Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable 2.7

Report on Advanced Resource Policies

Due date of deliverable: 2009-09-01 (T0+48)

Actual submission date: 2009-10-09

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **UEDIN**

Submitted version

Online repository revision code 7900

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary

This is Deliverable 2.7 of MOBIUS, an integrated project (FP6-015905) in the European Community Sixth Framework Programme. Full information about MOBIUS is available online at the project website `http://mobius.inria.fr`.

This deliverable reports on type systems and static analyses for advanced resource policies and analysis (Task 2.4: Advanced resource policies and analyses). The deliverable consists of 24 publications that have appeared elsewhere and includes contributions from all MOBIUS partners involved in Task 2.4, namely INRIA, LMU, UEDIN, and UPM. The results reported here build on previous work (Task 2.3: Types for basic resource policies).

This deliverable supersedes Deliverable 2.6, the Preliminary Report on Advanced Resource Policies. 11 of the 24 publications constituting this deliverable have already been included in Deliverable 2.6, 5 papers have been revised since, and 8 are new.

- Chapter 1 is a brief introduction outlining the space of type systems and program analyses covered by Task 2.4.
- Chapter 2 presents INRIA's implementation of an inter-procedural relational analysis for Java bytecode that can compute invariants used in the generation of resource certificates. The analysis is supplemented by a provably correct bytecode transformation to get rid of the operand stack.
- Chapter 3 reports on LMU's development of type inference algorithms for a type system which tracks amortised heap-space usage in Java programs, and LMU's development of a generic resource extension to the MOBIUS base logic. The extended logic serves as a target for translating type derivations into base logic proofs, as demonstrated by interpretations of type systems for constant heap space and for block booking resources.
- Chapter 4 presents UEDIN's approach to block booking: explicit accounting using resource managers. Resource safety is then enforced either dynamically by run-time monitoring, or statically by a type system. Additionally, preliminary results are reported about a static resource analysis for iterative Java code based on counting lattice points inside polytopes.
- Chapter 5 summarises UPM's work on resource usage analysis. It comprises a description of the different phases carried out to infer upper bounds from Java bytecode programs in the COSTA system. Also, it introduces a live heap space analysis which infers upper bounds on the peak of the heap usage along any execution of an input program.
- Appendix A (500+ pages) collates the 24 publications that constitute this deliverable.

Version Control History

Revisions

Version	Date	Purpose	Revision code
Draft D2.7	2009-08-07	First version, circulated for partner review	7645
Revised D2.7	2009-09-01	Revised version, prepared for lead site.	7827
Final D2.7	2009-10-09	Submitted to European Project Office	7900

Note: "Revision code" refers to the version control number assigned by the project online document repository. This uniquely identifies all historical versions of MOBIUS documents through drafting and editing.

Authors

Site	Contributed to Chapter
INRIA	2
LMU	3
UEDIN	1,4, editor
UPM	5

Contents

Chapter 1

Introduction

This deliverable reports the results of Task 2.4 on type systems and static analyses for advanced resource policies. The deliverable consists of copies of 24 publications — collated in the appendix for convenience — that have appeared elsewhere and includes contributions from all MOBIUS partners involved in Task 2.4, namely INRIA, LMU, UEDIN, and UPM, cf. following chapters for details.

This deliverable supersedes Deliverable 2.6, the Preliminary Report on Advanced Resource Policies. Of the 24 publications constituting this deliverable, 11 have already been included in Deliverable 2.6, 5 papers have been revised since, and 8 are new.

On mobile phones certain resources, like sending text messages, must be controlled tightly. As an example scenario take *bulk messaging* where the user wants to send a text message to a number of recipients. Because of the cost of sending text messages, the user must authorise each message explicitly. Java MIDP 2.0 implements this requirement by insisting on each message being authorised individually just before it is sent, thus bombarding the user with confirmation screens. A better way to fulfil this requirement would be collective authorisation of all messages in one go, also known as *block booking*. However, this requires tracking the flow of authorised resources from the points of authorisation to the points of use, to ensure that no more messages are sent than authorised.

The type systems and program analyses presented in this deliverable guarantee adherence to resource-related properties of mobile code, like soundness of block booking, for instance. Type systems are an enabling technology for the MOBIUS Proof-Carrying Code (PCC) architecture because they are intuitive, automatic and scalable. Hence improvements of program coverage and language coverage as well as of flexibility and scalability, as they are presented in this deliverable, are important steps to build the MOBIUS PCC-architecture.

The following chapters describe various type systems and program analyses for controlling resources, notably execution time, heap space, and access to external resources. The results reported here build on previous work (in Task 2.3) in various ways.

- Chapter 2 presents INRIA's implementation of an inter-procedural relational analysis for Java bytecode that can compute invariants used in the generation of resource certificates (e. g., for applications using block booking). The analysis relies on a stackless bytecode format (i. e., bytecode making trivial use of the operand stack only), which is produced automatically by a provably correct bytecode transformation.

- Chapter 3 reports on LMU's development of type inference algorithms for a type system which tracks amortised heap-space usage in Java programs, and LMU's development of a generic resource extension to the MOBIUS base logic. The extended logic serves as a target for translating type derivations into base logic proofs, as demonstrated by interpretations of type systems for constant heap space and for block booking resources.

- Chapter 4 presents UEDIN's approach to block booking: explicit accounting using resource managers. Resource safety is then enforced either dynamically by run-time monitoring, or statically by a type

system. Additionally, it reports early results about a static resource analysis for iterative Java code based on counting lattice points inside polytopes.

- Chapter 5 summarises UPM's work on resource usage analysis. It comprises a description of the different phases carried out to infer upper bounds from Java bytecode programs in the COSTA system. Also, it introduces a live heap space analysis which infers upper bounds on the peak of the heap usage along any execution of an input program.

Chapter 2

Polyhedral Analysis of Java Bytecode for Certificate Generation

In this deliverable, INRIA reports on the 2 publications listed below. The paper [1] is a revised version of the work reported last year in Deliverable 2.6, and [2] is new.

[1] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Research Report 6333, IRISA, September 2007. Revised August 2009.

[2] D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for Java bytecode. Research Report 7021, IRISA, 2009.

The main goal of this activity is the implementation of a relational (polyhedral) analysis for Java bytecode that can compute invariants used in the generation of resource certificates.

In [1], we have developed a relational (polyhedral-based) static analysis for Java bytecode that is capable of automatically computing invariants of the relations between program variables. By introducing variables that represent the amount of resources available (cf. the resource model for Java MIDP resources developed in the project), such an analysis can be used to prove that a program uses its resources correctly in the sense that it does not use more resources than it has been granted. To that end, the analysis infers invariants between these variables. It takes as input Java bytecode and infers for each bytecode instruction a relation between global variables, the parameters to a method, local variables and the numeric values stored on the stack. This relation is represented by elements of the abstract state underlying the analysis. Rather than fixing a particular abstract domain from the outset, we have specified the bytecode analysis with respect to an abstract numeric relational interface made up of a few central operations such as upper bounds, variable renaming and projection. These operations can then be instantiated with standard relational abstract domains such as polyhedra and octagons.

Compared to the work that was reported last year, we have developed a new implementation of the analysis. The modifications made on [1] are the following:

- The operand stack of Java bytecode can be costly to model fully in a relational analysis. In our previous work, we enriched the abstract domain to include symbolic expressions that represent the content of a stack element in terms of program variables. Our new implementation relies on the bytecode transformation, described in [2] that removes the operand stack manipulation. This work is described below.

- Inferring relations between the variables of a program allows to statically prove validity of array accesses. We use this safety policy as a case study to experimentally measure the precision of our relational analysis on real benchmarks. The revised version handles more benchmarks than before.

- We now treat some relations between reference fields, relying on the simple alias information given by the typing properties of the Java language.

- Our numerical abstract domain is now able to manage some non-linear reasoning and is sound with respect to the modulo arithmetic of Java.

In [2], we have investigated a semantic study of a bytecode transformation algorithm that allows to remove operand stack manipulations. We choose a transformation technique based on a symbolic execution of the bytecode, using a symbolic operand stack. We define a semantic preservation property and prove it is satisfied by our algorithm. We obtain hence a semantic proof that both bytecode programs and their respective stackless representation behave in the same way up to an observational semantic relation.

Such a provably correct transformation has many applications in static analysis where it has been used in several places but never been formalised and proven correct. The transformation is not trivial because of the special care that is taken not to generate too many temporary variables in the intermediate representation. It is also complicated by the fact that the analysis operates in one pass through the bytecode instructions of a program.

Future work should concern mechanisation of these results in the formal MOBIUS architecture. We intend to finish (a big part of it has already been achieved) a Coq soundness proof of our relational analyser in order to obtain a certified array bound checker for sequential Java bytecode programs. We also envisage mechanising our bytecode transformation to obtain a stackless layer of the Bicolano semantics.

Chapter 3

Improvements in the development of RAJA

In this deliverable, LMU reports on the 3 publications listed below. The papers [1, 3] were already reported in last year's Deliverable 2.6 (where [3] revises the paper titled *Implementing a type system for amortised heap-space analysis*), and [2] is new.

[1] L. Beringer, M. Hofmann, and M. Pavlova. Certification using the Mobius base logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects: Revised Lectures from the 6th International Symposium FMCO 2007*, Lecture Notes in Computer Science, volume 5382, pages 25–51. Springer-Verlag, 2008.

[2] M. Hofmann and D. Rodriguez. Membership checking in greatest fixpoints revisited. In *FICS: 6th Workshop on Fixed Points in Computer Science*, 2009.

[3] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In *CSL: 18th EACSL Annual Conference on Computer Science Logic*, Lecture Notes in Computer Science. Springer, 2009.

[1] describes a core component of MOBIUS' Trusted Code Base, the MOBIUS base logic. This program logic facilitates the transmission of certificates that are generated using logic- and type-based techniques and is formally justified w.r.t. the Bicolano operational model of the JVM. The paper motivates major design decisions, presents core proof rules, describes an extension for verifying intensional code properties, and considers applications concerning security policies for resource consumption and resource access.

Of particular relevance for the present deliverable are Section 4 and 5 of the named publication. In Section 4, a type system is presented that guarantees constant bound on heap space. In contrast to the type system discussed above, the system is phrased on bytecode, and has been formally justified with respect to the Bicolano operational semantics. Section 5 presents a solution to the block booking challenge, namely a bytecode-level type system for numeric correspondence assertions, where the authorisation request operation is parametric in a variable, such that the number of authorisations that are requested may depend dynamically on other data.

In [3] we have presented an improved typechecking algorithm for RAJA programs using the notion of *views* – refined types that describe the contribution of objects to the potential of data structures. We provide automatic type checking under relatively mild annotations. In particular, we automatically construct types arising from sharing and conditionals which had to be provided manually before. Finally, we prove soundness and completeness of the algorithm with respect to the declarative typing rules.

The notions of subtyping and sharing we use in [3] are slightly more flexible than the original ones from (Hofmann and Jost, ESOP 2006) and thus allow more examples to be typed. Semantic soundness of the improved system is a direct extension of the soundness proof in (Hofmann and Jost, ESOP 2006).[1] In

[1] `http://raja.tcs.ifi.lmu.de/download/files/rajaSoundProof.pdf`

particular, a sharing task can be reduced into a subtyping task in the presence of algorithmic views $(v_1 + v_2)$, i.e. a view that may be split into views v_1 and v_2.

Algorithmic views are automatically computed views based on user-defined views. We define many of them that are useful for the algorithm, for example: $(v_1 \vee v_2)$, $(v_1 \wedge v_2)$ which provide least upper bounds and greatest lower bounds for RAJA types respectively, or $(v \dot{-} \mathsf{n})$ which are views like v but with minus n units of potential. Future work should clarify the inference of view annotations in terms. We are currently working on an algorithmic typing system that collects subtyping constraints and tries to solve them.

In [2] we provide an algorithm for membership checking in greatest fixpoints, which we adapt for deciding subtyping for RAJA types. This algorithm extends a well-known algorithm for membership in greatest fixpoints of monotone operators of a special form called *invertible*[2] operators. The extended algorithm computes membership in the greatest fixpoint of arbitrary monotone operators. The algorithm has been proved correct by coinduction.

[2]Terminology due to B. C. Pierce. *Types and Programming languages.* MIT Press, 2002.

Chapter 4

Safety Guarantees from Explicit Resource Management

In this deliverable, UEDIN reports on the 4 publications listed below. The papers [1, 2] were already reported last year in Deliverable 2.6, and [3, 4] are new.

[1] D. Aspinall, P. Maier, and I. Stark. Monitoring external resources in Java MIDP. *Electronic Notes in Theoretical Computer Science*, 197(1):17–30, 2008.

[2] D. Aspinall, P. Maier, and I. Stark. Safety guarantees from explicit resource management. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects: Revised Lectures from the 6th International Symposium FMCO 2007*, Lecture Notes in Computer Science, volume 5382, pages 52–71. Springer-Verlag, 2008.

[3] K. MacKenzie. Resource analysis for iterative Java programs via lattice-point enumeration in polytopes. Technical Report EDI-INF-RR-1341, School of Informatics, University of Edinburgh, August 2009.

[4] P. Maier. Deciding extensions of the theories of vectors and bags. In N. D. Jones and M. Müller-Olm, editors, *Verification, Model Checking and Abstract Interpretation*, Lecture Notes in Computer Science, volume 5403, pages 245–259. Springer-Verlag, 2009.

The papers [1, 2, 4] focus on the compile- and run-time guarantees (e. g., static and dynamic soundness of block booking, cf. Chapter 1) of our approach to the explicit (i. e., manifest in code) management of resources such as text messages.

In [1], we present a Java library for MIDP devices which tracks and controls at run-time the use of potentially costly resources, such as sending text messages. The library supports block booking of resources while maintaining the security guarantee that attempted resource abuse is trapped. Tracking of resources is done by *resource managers*, special objects encapsulating multisets of authorised resources. This allows for fine-grained tracking; for instance, we are able to track not just the total number of text messages sent by an application, but the number of messages sent to each individual recipient. To reduce the run-time overhead of tracking multisets, resource managers can easily be erased without altering an application's behaviour if that application is *dynamically resource safe*, i. e., cannot be caught abusing resources. Additionally, the library introduces a flexible notion of *policy* for deciding which resources to grant.

The resource manager library for trapping attempts to abuse resources at run-time can be viewed as a language-based mechanism enforcing resource safety at run-time. In [2], we complement this with a type system for proving that a given program (in a functional language with resource managers) does not attempt to abuse resources. The type system derives logical constraints (in a generic logical constraint language) approximating the effects of evaluating program expressions. Typability of functions in the effect type system induces a notion of *static resource safety*, and in particular implies dynamic resource safety. As a consequence, resource managers can always be erased from well typed programs.

The decidability of type checking in the above type system rests on the decidability of satisfiability in the underlying constraint language. As examples like the bulk messaging application reveal, meaningful types require a constraint language able to express properties of multisets (to model resource managers) and container data structures (such as vectors and dictionaries). In [4], we present a decision procedure for such an expressive constraint language combining vectors and multisets.

The report [3] presents early results on a static resource analysis — based on methods from convex geometry — for determining tight bounds on the number of iterations of nested loops. Abstract interpretation over a domain of polyhedra (a technique due to Cousot and Halbwachs) is used to obtain linear constraints on control variables in iterative loop nests in Java programs. These constraints delimit a polyhedral region in some Euclidean space, and the points within this region which have integral coordinates correspond to the iterations of the loops. A technique of Barvinok is used to calculate a compact generating function from which it is easy to calculate the number of such points, and thus the number of times a particular program location is visited during execution. We have implemented a Java compiler which exploits these techniques to automatically determine resource bounds for realistic iterative Java programs.

Chapter 5

Static Resource Analysis of Java bytecode

To this deliverable, UPM contributes the 15 publications listed below. The papers [1, 4, 7, 9, 10, 12, 13, 14] were already reported last year in Deliverable 2.6, the papers [3, 5, 15] have been revised since, and [2, 6, 8, 11] are new.

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java bytecode. In *Formal Methods for Open Object-Based Distributed Systems: Proceedings of the 10th IFIP WG 6.1 International Conference FMOODS 2008, Oslo, Norway, June 4–6, 2008*, Lecture Notes in Computer Science 5051, pages 2–18. Springer-Verlag, 2008.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and cost analysis with COSTA and its user interfaces. In *Spanish Conference on Programming and Computer Languages*. Electronic Notes in Theoretical Computer Science, 2009. To appear.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper-bounds in static cost analysis. Submitted to the *Journal of Automated Reasoning*.

[4] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In M. Alpuente and G. Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, Lecture Notes in Computer Science 5079, pages 221–237. Springer-Verlag, 2008.

[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. Submitted to *ACM Transactions on Programming Languages and Systems*.

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource usage analysis and its application to resource certification. In *Foundations of Security Analysis and Design. FOSAD 2008/2009 Tutorial Lectures*, LNCS. Springer-Verlag, To appear. 2009.

[7] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap space analysis of Java bytecode. In *ISMM '07: Proceedings of the 6th International Symposium on Memory Management*, pages 105–116, New York, NY, USA, 2007. ACM Press.

[8] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *ISMM'09: Proceedings of the 8th International Symposium on Memory Management*, New York, NY, USA, June 2009. ACM Press.

[9] S. Genaim and F. Spoto. Constancy analysis. In M. Huisman, editor, *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.

[10] M. Marron, M. V. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In L. Hendren, editor, *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, Lecture Notes in Computer Science, volume 4959, pages 245–259. Springer-Verlag, 2008.

[11] M. Marron, D. Kapur, and M.Hermenegildo. Identification of logically related heap regions. In *ISMM'09: Proceedings of the 8th International Symposium on Memory Management*, New York, NY, USA, June 2009. ACM Press.

[12] M. Méndez-Lojo and M. V. Hermenegildo. Precise set sharing analysis for Java-style programs. In F. Logozzo, D. Peled, and L. D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, Lecture Notes in Computer Science, volume 4905, pages 172–187. Springer-Verlag, 2008.

[13] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards execution time estimation in abstract machine-based languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[14] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable resource usage nalysis for Java bytecode. Technical Report UNM TR-CS-2008-02 - CLIP1/2008.0, University of New Mexico, Department of Computer Science, UNM, January 2008.

[15] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-definable resource usage bounds analysis for Java bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2009.

For brevity, we describe only the new publications [2, 6, 8, 11] in this chapter; please check Deliverable 2.6 for summaries of the other publications.

In [6] we provide a high-level overview of the main techniques used in COSTA[1], a static analysis system which allows obtaining safe symbolic upper-bounds on the resource usage of Java bytecode (JBC for short). The results are *symbolic* in the sense that they do not refer to concrete, platform dependent, resources such as execution time, but rather they provide platform-independent information. This has the advantage that the results are applicable to any implementation of the Java Virtual Machine (JVM) on any particular hardware and the disadvantage that the information cannot refer to platform specific resources such as run-time. The fact that the analysis handles JBC represents that, at least in principle, it can deal with general-purpose programs written in a mainstream programming language such as Java and potentially other languages compiled to JBC. This high-level overview publication is of interest for dissemination of the research techniques developed in the project.

In [2] we describe the different *user interfaces* provided by COSTA, which greatly facilitate user interaction with the system. Such interfaces include: a classical *command line interface*; a *Web interface* which allows using COSTA from a remote location, without the need of installing it locally; and a recently developed Eclipse *plug-in*. The latter allows using the analyser during the development phase, in a widely used programming environment. This plugin allows programmers to analyse methods during the development process. As in the web interface, users can configure a large set of options by using the Eclipse preferences configuration window. Also, the user can choose either the automatic analysis or the expert mode which allows a more fine-grained customisation. By using this plugin, one can analyse one or several methods from a class or the whole class (by running the analysis on all its methods). The results of the analysis are shown using markers in the source code (see Fig. 5.1). Such markers are different depending on the cost model used for analysis. In addition, the plugin also shows all previous analysis results in an additional view, which we

[1] More information about COSTA can be found at `http://costa.ls.fi.upm.es`.

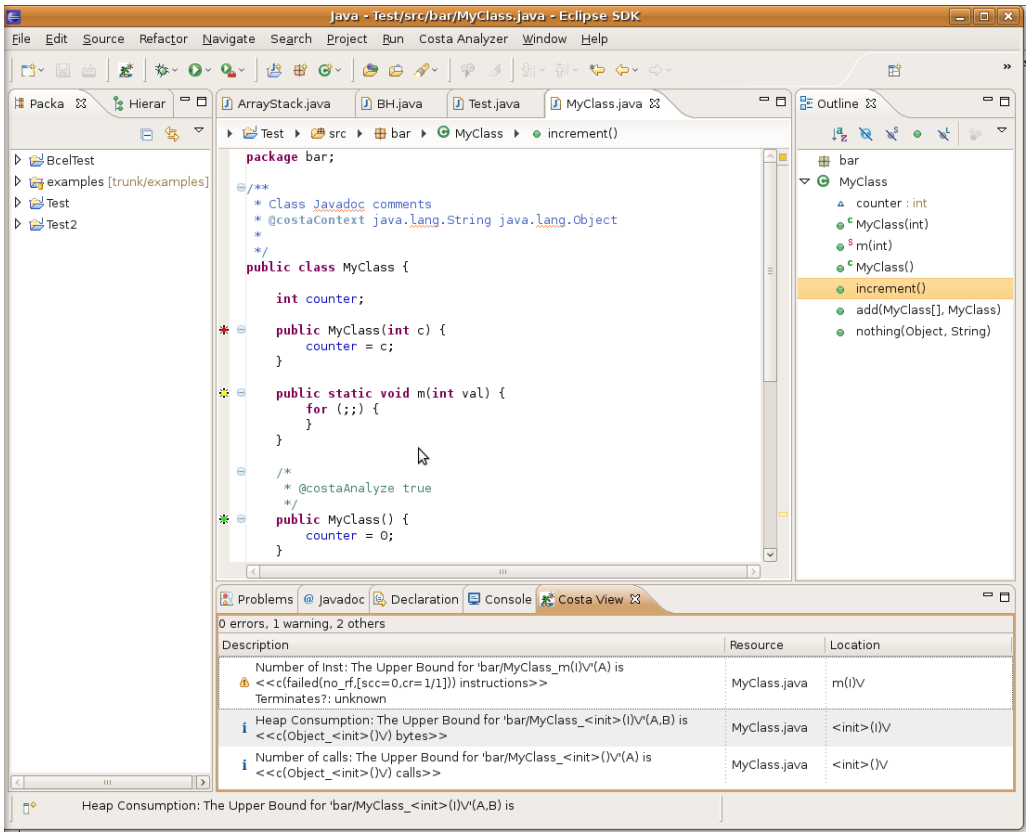


Figure 5.1: COSTA Plugin Markers and View

call "the COSTA view". The COSTA view also includes a warning icon for methods whose termination is not proved, in order to alert the programmer about potential problems. It can also read comments in the source code, written in Javadoc style, in order to set up analysis information.
COSTA can deal with almost full sequential Java, either in the *Standard edition* or the *Micro edition*. COSTA is able to read standard `.class` files and produce meaningful and reasonably precise results for non-trivial programs, possibly using Java *libraries*. Possible uses of such cost and termination results include:

- Helping the programmer in the *development* process, as obtained by including COSTA in Eclipse as a plugin.
- *Certification* of resource usage upper bounds and termination, thus providing guarantees to the code user, in the style of *Proof-carrying code*.
- Program *optimisation*, as for example, in concurrent systems.

Among all the above applications of resource analysis, we describe in [6] its application to *resource certification*, whereby programs are coupled with information about their resource usage. This information allows deciding whether the resources used by the program execution are acceptable or not *before* running the program.

The works [8, 11] are related to heap analysis. In the context of COSTA, we present a general framework to infer accurate bounds on the peak heap consumption of programs which improves the state-of-the-art in that it is not restricted to any complexity class and deals with all bytecode language features including recursion. To pursue our analysis, in [8] we characterise the behaviour of the underlying garbage collector. We assume a standard *scoped-memory* manager that reclaims memory when methods return. In this setting, our main contributions are:

1. *Escaped Memory Analysis.* We first develop an analysis to infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that is allocated during the execution of the method *and* which remains upon exit. The key idea is to infer first an upper bound for the total memory allocation of the method. Then, such bound can be manipulated, by relying on information computed by the *escape analysis*, to extract from it an upper bound on its escaped memory.

2. *Live Heap Space Analysis.* By relying on the upper bounds on the escaped memory, as our main contribution, we propose a novel form of *peak consumption cost relation* which captures the peak memory consumption over all program states along the execution for the considered scoped-memory manager. An essential feature of our CRs is that they can be solved by using existing tools for solving standard CRs.

3. *Ideal Garbage Collection.* An interesting, novel feature of our approach is that we can refine the analysis to accommodate other kinds of scope-based managers which are closer to an *ideal* garbage collector which collects objects as soon as they become unreachable.

4. *Implementation.* We report on a prototype implementation which is integrated in the COSTA system and experimentally evaluate it on the JOlden benchmark suite. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way.

The second paper on heap analysis [11] introduces a novel technique for identifying logically related sections of the heap such as recursive data structures, objects that are part of the same multi-component structure, and related groups of objects stored in the same collection/array. This information can be used in optimisations such as pool allocation, object co-location, static deallocation, and region-based garbage collection. This is illustrated using the Barnes-Hut benchmark from the JOlden suite.

Appendix A

Copies of Publications

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Result certification for relational program analysis

Frédéric Besson — Thomas Jensen — David Pichardie — Tiphaine Turpin

N° 6333 — version 2

initial version October 2007 — revised version August 2009

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6333--FR+ENG

Result certification for relational program analysis

Frédéric Besson*, Thomas Jensen† , David Pichardie*, Tiphaine Turpin‡

Thème SYM — Systèmes symboliques
Projet Lande

Rapport de recherche n° 6333 — version 2 — initial version October 2007 — revised version August 2009 — 29 pages

Abstract: We define a generic relational program analysis for an imperative, stack-oriented byte code language with procedures, arrays and global variables and instantiate it with an abstract domain of polyhedra. The analysis has automatic inference of loop invariants and method pre-/post-conditions, and efficient checking of analysis results by a simple checker. Invariants, which can be large, can be specialized for proving a safety policy using an automatic pruning technique which reduces their size. The result of the analysis can be checked efficiently by annotating the program with parts of the invariant together with certificates of polyhedral inclusions, which allow to avoid certain complex polyhedral computation such as the convex hull of two polyhedra. Small, easily checkable inclusion certificates are obtained using Farkas lemma for proving the absence of solutions to systems of linear inequalities. The resulting checker is sufficiently simple to be entirely certified within the Coq proof assistant.

Key-words: Static analysis, abstract interpretation, bytecode Java, Coq

This is a revised version. A section about describing the implementation of the analyser has been added.

* INRIA Rennes - Bretagne Atlantique/IRISA
† CNRS/IRISA
‡ Université Rennes I/IRISA

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Certification de résultat pour l'analyse de programme relationnelle

Résumé : Nous proposons une analyse générique de programme relationnelle pour un langage de bytecode impératif avec pile d'opérande, procédures, tableaux et variables globales. Cette analyse est instanciée avec un domaine abstrait de polyèdres. Elle propose une inférence automatique d'invariants de boucle et de préconditions/postconditions de procédures, ainsi qu'une vérification efficace du résultat de l'analyse par un vérificateur simple. Les invariants, qui peuvent être grands, peuvent être spécialisés pour prouver une propriété de sûreté en utilisant une technique automatique de compression de taille de certificat. Le résultat de l'analyse peut être vérifié efficacement en annotant le programme avec une partie des invariants et quelques certificats d'inclusion de polyèdre, qui permettent d'éviter certaines calculs polyédriques complexes comme le calcul de l'enveloppe convexe de deux polyèdres. Nous obtenons des certificats d'inclusion petits et facilement vérifiables grâce au lemme de Farkas pour prouver l'absence de solution dans un système d'inégalités linéaires. Le vérificateur ainsi obtenu est suffisamment simple pour être entièrement certifié avec l'assistant à la preuve Coq.

Mots-clés : Analyse statique, interprétation abstraite, bytecode Java, Coq

1 Introduction

Logic-based, static program verification, be it in form of abstract interpretation, symbolic model checking or interactive proving of programs, is used in a number of ways to improve the confidence in safety-critical systems and for protecting host machines from malicious code, as *e.g.*, done by the Java byte code verifier. As applications and the program logics grow in complexity, an automated technique for verifying program invariants based on a program logics should ideally meet all of the following three requirements:

- *Automatic Inference*: the complexity of both programs and the underlying logic can quickly make it burdensome to conduct program proofs manually. Automatic inference of program properties is necessary to obtain a technique that scales.
- *Result certification*: when inference is available, it often relies on advanced deductive methods for inferring an invariant whose size and complexity make it difficult to ascertain its validity manually. Efficient checking of the result of the inference or of any proposed invariant in general becomes important.
- *Small Trusted Computing Base (TCB)*: the result checker becomes the cornerstone of the reliability of the verification framework. In order to reduce the part of the code base that needs to be trusted without proof, the checker should be kept sufficiently simple and small in order to be able to verify the checking algorithmics mechanically.

Program verification based on general Hoare-style program logics may follow the Verification Condition Generator (VCGen) approach of *e.g.*, Extended Static Checking by Flanagan, Leino *et al.* [18] or use expressive type systems such as the dependent type systems of Xi and Pfenning [36] for proving properties of programs. The approaches based on VCGens are generally complete for partial correctness and will produce a set of verification conditions which, when satisfied, will allow to conclude that a given program property holds in the logic. Verification conditions often fall into fragments of logic that require them to be proved by dedicated decision procedures or theorem provers. VCGens and the type-based approaches are primarily concerned with invariant checking and discard part of the inference problem by relying on loop invariants and pre-post-condition of methods to be provided by the programmer. In terms of small TCB, the VCGens remain complex software which are hard to prove correct *in extenso*. The machine-checked formalizations e.g., by Nipkow, Wildmoser *et al.* [33, 34] show that this is indeed possible to certify an entire VCGen inside a proof assistant but also that this remains a major software certification challenge.

Another strand of program verification is based on abstract interpretation. Abstract interpretation is an automatic technique for inferring program properties in the form of fixpoints of monotone data flow functions. As a theory of proving programs it has strong semantic foundations. At the same time it should be noted that the algorithmics of the domains underlying the more advanced analyses such as polyhedral analysis (initially described by Cousot and Halbwachs [15]) is highly non-trivial. Checking an invariant is in theory simple as it only requires one more iteration to check that a property is indeed a fixpoint but, as said, this computation does in certain cases rely on non-trivial algorithmics that forms part of what must be trusted. In previous work [11, 29], some of the authors formalised the theory of abstract interpretation inside the proof assistant Coq and extracted Caml implementations of a variety of program analyses. This Certified Abstract Interpretation approach represents a systematic way of reducing the TCB of static analyzers and fulfills the three requirements listed above. However, a fully mechanised correctness proofs of more advanced program analysers such as an optimised, polyhedral-based analysis would require an enormous effort in terms of program certification.

The purpose of this paper is to demonstrate that by focusing on certifying the *result* of the analysis rather than the analysis itself, it is possible to develop a verification framework for advanced program properties that satisfies all of the three desired properties and, at the same time, requires a significantly smaller effort in order to be proved correct. This idea was previously used by Wildmoser *et al* [32] who use the result of an untrusted interval analysis in a VCGen for byte

```
  //     PRE:  0 ≤ |vec0|
static int bsearch(int key, int[] vec) {
  // (I1) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ |vec0|
   int low = 0, high = vec.length - 1;
  // (I2) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec0|
   while (0 < high-low) {
  // (I3) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
      int mid = (low + high) / 2;
  // (I4) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0| ∧ low + high − 1 ≤ 2 · mid ≤ low + high
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
   // (I5) key0 = key∧|vec0| = |vec|∧−2+3·low ≤ 2·high+mid∧−1+2·low ≤ high+2·mid∧−1+low ≤ mid ≤
1+high∧high ≤ low+mid∧1+high ≤ 2·low+mid∧1+low+mid ≤ |vec0|+high∧2 ≤ |vec0|∧2+high+mid ≤
|vec0| + low
   }
  // (I6) key0 = key ∧ |vec0| = |vec| ∧ low − 1 ≤ high ≤ low ∧ 0 ≤ low ∧ high < |vec0|
   return -1;
} //    POST:  −1 ≤ res < |vec0|
```

Figure 1: Binary search

code and by Leroy [23] in his certification of a compiler back-end where he, rather than certifying the complex graph-coloring algorithms for register allocation, proves the correctness of a checker that verifies a given coloring returned by an untrusted graph-coloring algorithms. Here, we generalise this idea by developing a relational analysis framework together with a certified checker. The basic observation is that an abstract interpretation can be decomposed into an abstract domain of properties, a generic program logic for reasoning about these properties and a fixpoint engine for solving recursive equations over the abstract domains. The inference does not need to use certified abstract domain operations and fixpoint engines, and the checking of invariants does not need to use a fixpoint engine at all. We take advantage of this to design a checker that re-uses the program logic but replaces the more complex domain operations with simpler ones, at the expense of providing some extra information in the certificate accompanying a program.

2 Overview

In the first part of this paper, we will develop a fully relational, interprocedural analyser which automatically infers an invariant for each control point in the program, a pre-condition that must hold at the point of calling a procedure and a post-condition that is guaranteed to hold when the procedure returns. Relational analyses are useful for finding loop invariants needed for proving program safety, e.g. when verifying the resource usage of programs or verifying safety properties related to safe memory access such as checking that all array accesses are within bounds. We will take Safe Array Access as an example safety policy and illustrate our approach with the Binary Search example given in Fig. 1, showing how the analysis will prove that the instruction that accesses the array `vec` with index `mid` will not index out of bounds.

We have annotated the code of Binary Search with the invariants that have been inferred automatically. Invariants refer to values of local and global variables and can also refer to the length of an array. For example, the invariant (I_3) asserts among other properties that when entering the while loop, the relation $0 \leq \texttt{low} < \texttt{high} < |\texttt{vec}|$ is satisfied. Similarly, the post-condition ensures that the result is a valid index into the array being searched, or -1, indicating that the element was not found. In addition, the analysis introduces a 0-indexed variable (such as *e.g.* $\texttt{key}_0$ in the example) for each parameter (and also for the global variables, of which there are none in the example) in order to refer to its value when entering the procedure. The effect of this is that the invariant on exit of the program defines a relation between the input and the output of the procedure, thus yielding a *summary relation* for the procedure.

2.1 Compressing invariants

Abstract interpretations may give you more information than you need for proving a particular property. In the case of the Binary Search example, if we are only interested in proving the validity of array accesses, there are a number of relations between variables in the invariants that can be forgotten. Reducing the number of constraints and the number of variables under consideration can lead to a significant gain in execution time when it comes to checking a proposed invariant. For example, pruning the invariants in Fig. 1 with respect to this property yields the simpler invariant shown in Fig. 2:

```
  //     PRE: True
static int bsearch(int key, int[] vec) {
  // (I'_1) |vec_0| = |vec| ∧ 0 ≤ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I'_2) |vec_0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec_0|
   while (0 < high-low) {
  // (I'_3) |vec_0| = |vec| ∧ 0 ≤ low < high < |vec_0|
      int mid = (low + high) / 2;
  // (I'_4) |vec| − |vec_0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
  //        2 · high − 2 · mid − 1 ≥ 0 ∧ |vec_0| − high − 1 ≥ 0
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
  // (I'_5) |vec_0| = |vec| ∧ −1 + low ≤ high ∧ 0 ≤ low ∧ 5 + 2 · high ≤ 2 · |vec|
   }
  // (I'_6) 0 ≤ |vec_0|
   return -1;
} //    POST: −1 ≤ res < |vec_0|
```

Figure 2: Binary search after invariant pruning

Notice that the inferred loop invariant I'_3 is close to what a specifying programmer of Binary Search might have come up with, but here produced automatically. We explain pruning of procedures in Section 7.

2.2 Analysing a stack-based language

Polyhedral analysis of While languages is well understood but we want our framework to be able to analyse byte code programs and not only source code. We could in theory avoid the problem by transforming the program into three-address code and treat each stack location as a local variable but this transformation is expensive from an algorithmic point of view, as it increases the number of times that the relation has to be updated. Instead, we achieve the effect of this transformation by defining an analysis for stack-oriented byte code that combines relational abstract interpretation with symbolic execution, following an idea previously used for analysing Java byte code by Xi and Xia [37] and Wildmoser *et al* [32]. This technique abstracts the environment of local variables by a relation (e.g., a polyhedron) and replace the operand stack with a stack of symbolic expressions used to "decompile" the operations on the operand stack. For example, the comparison of variables `low` and `high` will be compiled to the byte codes below, which are analysed in a state consisting of the relation I_2 as defined in Fig. 1 and an abstract stack that evolves as values are pushed onto the stack.

		[]	I_2
7 :	ipush 0	0	I_2
8 :	iload high	high :: 0	I_2
9 :	iload low	low :: high :: 0	I_2
10 :	isub	(high−low) :: 0	I_2
11 :	if_icmpge 56	[]	I_3

Before the comparison in instruction 11, the stack top contains the expression `high−low`, reflecting that in the real execution the stack top at this point will contain the value of this expression. When we learn from the test that the expression `high>low` evaluates to true in the state immediately following the comparison (and only then), we update the relation accordingly to obtain invariant I_3. Similarly, we have to update the relation when assigning a new value to a variable. For example, the instruction that assigns (`high+low`)/2 to `mid` is compiled and analysed as shown below. Again, the relation I_3 is only updated when the assignment to `mid` is done, to yield relation I_4.

		[]	I_3
14 :	iload low	`low`	I_3
15 :	iload high	`high` :: `low`	I_3
16 :	iadd	(`high+low`)	I_3
17 :	ipush 2	2 :: (`high+low`)	I_3
18 :	idiv	((`high+low`)/2)	I_3
19 :	istore mid	[]	I_4

More generally, with the abstract stack of expressions, only the comparisons and assignment to variables require updating the relation. In a polyhedron-based analysis this is a substantial saving.

2.3 Result checking with certificates

Checking an invariant obtained by computing a post-fixpoint of an abstract interpretation is in theory simple as it only requires one more iteration to check that it is indeed a post-fixpoint. In addition, only invariants at certain program points such as loop headers are required for rebuilding an entire invariant in one iteration. Lightweight Bytecode Verification by Rose [30] and the more general Abstraction-Carrying Code by Albert, Puebla and Hermenegildo [1] exploit this to construct efficient checkers for invariant-based program certificates. For the code in Fig. 2, only I'_2 is required.

The inference of invariants using our relational analysis uses an iterative fixpoint solver over an abstract domain of polyhedra and is in principle amenable to the same technique. However, despite efficient implementations of basic polyhedral operations, the algorithmic complexity of operations such a computing the least upper bound (*i.e.* the convex hull) of two polyhedra remains high, and certifying them in a proof assistant would be a major undertaking.

Instead, we propose an enriched certificate format which has the virtue of being simpler to check, at the cost of sending more information than in basic fixpoint reconstruction. We exploit that, for the checker, the only important property of the convex hull operators is that it produces an upper bound of two polyhedra and therefore can be replaced by inclusion checks with respect to an upper bound that is proposed by the certificates. Upper bounds are computed at join points so in Fig. 2 we would also supply I'_5.

Safety checks also reduces to inclusions of polyhedra as verifying the array access `vec[mid]` amounts to ensuring that I'_4 implies $0 \leq \texttt{mid} < |\texttt{vec}|$. By simple propositional reasoning, this reduces to proving that the linear systems of constraints $-\texttt{mid}-1 \geq 0 \wedge I'_4$ and $\texttt{mid}-|\texttt{vec}| \geq 0 \wedge I'_4$ have no solution. Due to a result by Farkas, such problems can be checked efficiently using certificates by a simple matrix computation. The key insight is that unsolvability follows from the existence of a *positive* combination of the constraints which yield a strict negative constant. This would lead to a contradiction because the sum and product of positive quantities cannot be strictly negative. The certificate is therefore a vector which records the coefficients of the positive combination. For example, the certificate $[\mathbf{2};\mathbf{2};\mathbf{0};\mathbf{0};\mathbf{1};\mathbf{2}]$ proves that the constraints $\texttt{mid} - |\texttt{vec}| \geq 0 \wedge I'_4$ are unsatisfiable, as the expression

$$\begin{array}{l}\mathbf{2}\cdot(\texttt{mid}-|\texttt{vec}|)+\mathbf{2}\cdot(|\texttt{vec}|-|\texttt{vec}_0|)+\mathbf{0}\cdot\cdots+\mathbf{0}\cdot\cdots+\\ \mathbf{1}\cdot(2\cdot\texttt{high}-2\cdot\texttt{mid}-1)+\mathbf{2}\cdot(|\texttt{vec}_0|-\texttt{high}-1)\end{array}$$

evaluates to -2. We explain these certificates in detail in Section 8.

2.4 Certified certificate checkers

The result checking technique explained above already drastically reduces the TCB of the analysis result which only rely on the result checker. To further reduce the TCB, we have machine-checked the result checker of our analysis in the Coq proof assistant. The main components of the formalisation are

1. a predicate `Safe:program→Prop` which models the safe programs with respects to the the semantics described in Section 4,
2. a function `checker:program→certificate→bool`, which checks the safety of a program using a certificate containing a (partial) result of an analysis and some inclusion certificates,
3. a machine checked proof establishing the correctness of the checker:

```
Theorem checker_correct :
  ∀ p cert, checker p cert = true → Safe p.
```

The Trusted Computed Base is hence reduced to the Coq type checker and the formal definition of program safety.

Once the certified result checker is verified (by the Coq type checker) and installed by the code consumer, two scenarios can be envisaged to verify the safety of programs sent by producers. In the first one, the consumer may use an efficient Ocaml version of the checker, extracted from the Coq version thanks to the Coq extraction mechanism. The other alternative is related to *proof by reflection*. For each program `p` and certificate `cert` the consumer may build a foundational Coq proof of `Safe p`. To do so he only has to check in Coq the term `checker_correct p cert refl_eqtrue` where `refl_eqtrue` denotes a proof of `true=true`. It is the role of the Coq reduction engine to verify during type checking if `true=true` is equivalent to `checker p cert = true` by running the checker inside Coq. In this way we combine two desirable features which are often difficult to reconcile in state-of-the art Proof Carrying Code: foundational proofs and small certificates.

3 Notations

Let A and B be sets. If A and B are disjoint then $A + B$ is the disjoint sum of A and B. We write $A_\bot$ the set $A + \{\bot\}$. For $f \in A \to B_\bot$, $dom(f) = \{a \in A \mid f(x) \neq \bot\}$. Let $f \in A \to B$, $f[x \mapsto v]$ is the function identical to f everywhere except for x for which it returns v. The notation $[x_1 \mapsto v_1; \ldots; x_n \to v_n]$ stands for a function f of domain $\{x_1, \ldots, x_n\}$ such that $f(x_i) = v_i$. A^* is the set of lists of elements of A. We write [] for the empty list and $a_0 :: \ldots :: a_{n-1}$ is a list l of length n ($|l| = n$) whose head (resp. tail) is a_0 (resp. a_{n-1}). $l[i]$ is the i-th element of l. We write a^i the list that is the repetition of a, i times. Let V, W be totally ordered sets. For $x \in V$, $\iota_V(x)$ is the index of x in set V and ι_V^{-1} is the inverse function. We abuse notations and identify $A^{|V|}$ with $V \to A$ *i.e.,* given a finite ordered set, $V = \{x_1, \ldots, x_n\}$ such that $x_1 < \ldots < x_n$, we identify the finite mapping $[x_1 \mapsto v_1, \ldots x_n \mapsto v_n]$ with the n-tuple $(v_1, \ldots, v_n)$. We will write A^V to denote both $A^{|V|}$ and $V \to A$. Let $\rho \in A^V$ and $V' \subseteq V$, $\rho_{|V'} \in A^{V'}$ is the restriction of e over the variables of V' such that for all $x \in V'$, $\rho_{|V'}(x) = e(x)$. Given V and W disjoint set of variables, $\rho_1 \in A^V$ and $e_2 \in A^W$, we write $\rho_1 \oplus \rho_2 \in A^{V+W}$ for the finite mapping such that $(\rho_1 \oplus \rho_2)_{|V} = \rho_1$ and $(\rho_1 \oplus \rho_2)_{|W} = \rho_2$. Let W and W' ordered sets of same cardinality. If $\rho \in A^{V+W}$, then $\rho_{W \to W'} \in A^{V+W'}$ is obtained by renaming the variables of W to the variables in W'. Formally, we have $\rho_{W \to W'}(x) = \rho(x)$ if $x \in V$ and $\rho_{W \to W'}(x) = \rho(\iota_W^{-1}(\iota_{W'}(x)))$ if $x \in W'$. To make the distinction clear between syntactic expressions and values, syntactic expressions are bracketed ($\llcorner \cdot \lrcorner$). For example, we write $\llcorner 1 + e \lrcorner$ a syntactic expression built by applying the $+$ operator to the constant 1 and the syntactic expression e.

4 A byte code language and its semantics

We use a simple stack-based byte code language to illustrate our ideas. Features include integers, dynamically created (unidimensional) array of integers, static methods (procedures) and static fields (global variables).

Programs are lists of methods and a method consists of a name, a number of arguments and a list of instructions. In the following, f ranges over the set S of static field names, r ranges over the set $R = \{r_0, \ldots, r_{|R|}\}$ of local variables and id ranges over the set $MethId$ of method names. Moreover, i and n range over $\mathbb{N}$ or $\mathbb{Z}$ depending on the context and p is used for control points.

$$
\begin{array}{rcrcl}
P & \in & Prog & = & Meth^* \\
m & \in & Meth & = & Sig \times Code \\
 & & Sig & = & MethId \times \mathbb{N} \\
c & \in & Code & = & Instr^* \\
instr & \in & Instr & & \\
instr & ::= & \multicolumn{3}{l}{Nop \mid Ipush\ n \mid Iinc\ r\ n \quad \text{where } n \in \mathbb{Z}} \\
 & & \multicolumn{3}{l}{Pop \mid Dup \mid Ineg \mid Iadd \mid Isub \mid Imult \mid Idiv} \\
 & & \multicolumn{3}{l}{Load\ r \mid Store\ r} \\
 & & \multicolumn{3}{l}{Getstatic\ f \mid Putstatic\ f} \\
 & & \multicolumn{3}{l}{Newarray \mid Arraylength \mid Iaload \mid Iastore} \\
 & & \multicolumn{3}{l}{Goto\ p \mid If\_icmp\ cond\ p} \\
 & & \multicolumn{3}{l}{\quad \text{where } cond \in \{=, \neq, <, \leq\}} \\
 & & \multicolumn{3}{l}{Invoke\ sig \quad \text{where } sig \in Sig} \\
 & & \multicolumn{3}{l}{Iinput \mid Return}
\end{array}
$$

The instruction set has operators for integer arithmetic and for manipulating local variable, static fields and an operand stack. Instructions on arrays permit to create, obtain the size of, access and update arrays. The flow of control can be modified unconditionally (with *Goto*), and conditionally with the family of conditional instructions *If_icmp cond* which compare the top elements of the run-time stack and branch according to the outcome. Input of data is modelled with the instruction *Iinput*. The inter-procedural layer of the language contains an instruction *Invoke* for invoking a method and an instruction *Return* which transfers control to the calling method, and, at the same time returns the top of the operand stack as result by pushing it onto the operand stack of the caller (see the operational semantics below).

A program state is composed of a frame stack, the value of static fields and a heap of arrays and has the form $<(m, p, s, l)^*, g, h>$. Each frame is a triple composed of a method m, a control point p to be executed next, an operand stack s local to a frame and l a partial mapping from local variables to values. The global heap h is used for storing allocated arrays and is modelled as a partial function from memory locations to arrays. A special error state *Error* models the run-time error arising from indexing an array outside its bounds.

$$
\begin{array}{rcrcl}
ref & \in & Location & & \\
v & \in & Val & = & \mathbb{Z} + Location \\
s & \in & Stack & = & Val^* \\
l & \in & LocVar & = & R \rightarrow Val \\
a & \in & Array & = & \mathbb{Z}^* \\
h & \in & Heap & = & Location \rightarrow Array_\perp \\
g & \in & Static & = & S \rightarrow Val \\
 & & Frame & = & Meth \times \mathbb{N} \times Stack \times LocVar \\
 & & State & = & Frame^* \times Static \times Heap \\
 & & & & + \{Error\}
\end{array}
$$

The byte code language is given an operational semantics via a transition relation $\rightarrow$ between states. Some of the rules of the definition of $\rightarrow$ are shown in Fig. 3. In the semantics, for a method $m = ((id, n), c)$, we write $m[p]$ for $c[p]$. Note that the language is untyped: registers and fields

may (and will) point successively to values of different types during execution. Instructions that require arguments with a certain type get stuck in case of error. Also, the number of registers $|R|$ is the same for all methods. Unused registers and uninitialised fields have the value 0. Finally, we only consider states $<st, g, h>$ such that every location appearing in st, g is in $dom(h)$, which is clearly preserved by the semantics in Fig. 3.

$$\frac{}{s,l,g,h \xrightarrow{Ipush\ n} n :: s,l,g,h} \qquad \frac{}{n_2 :: n_1 :: s,l,g,h \xrightarrow{Iadd} n_1 + n_2 :: s,l,g,h}$$

$$\frac{l(r) = n}{s,l,g,h \xrightarrow{Iinc\ r\ i} s,l[r \mapsto n+i],g,h} \qquad \frac{}{s,l,g,h \xrightarrow{Load\ r} l(r) :: s,l,g,h}$$

$$\frac{}{v :: s,l,g,h \xrightarrow{Store\ r} s,l[r \mapsto v],g,h} \qquad \frac{}{s,l,g,h \xrightarrow{Getstatic\ f} g(f) :: s,l,g,h}$$

$$\frac{h(ref) = \bot \quad n \geq 0}{n :: s,l,g,h \xrightarrow{Newarray} ref :: s,l,g,h[ref \mapsto 0^n]}$$

$$\frac{h(ref) = a \quad 0 \leq i < |a|}{i :: ref :: s,l,g,h \xrightarrow{Iaload} a[i] :: s,l,g,h} \qquad \frac{h(ref) = a \quad \neg\ 0 \leq i < |a|}{i :: ref :: s,l,g,h \xrightarrow{Iaload} Error}$$

$$\frac{m[p] = instr \quad s,l,g,h \xrightarrow{instr} s',l',g',h'}{<(m,p,s,l) :: st,g,h> \rightarrow_P <(m,p+1,s',l') :: st,g',h'>}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad n_1\ cond\ n_2}{<(m,p,n_2 :: n_1 :: s,l) :: st,g,h> \rightarrow_P <(m,p',s,l) :: st,g,h>}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad \neg\ n_1\ cond\ n_2}{<(m,p,n_2 :: n_1 :: s,l) :: st,g,h> \rightarrow_P <(m,p+1,s,l) :: st,g,h>}$$

$$\frac{m[p] = Invoke\ (mn,n) \quad m' = ((mn,n),c) \in P}{\begin{array}{l}<(m,p,(v_{n-1} :: \ldots :: v_0 :: s),l) :: st,g,h> \rightarrow_P \\ \quad <m',0,[],[r_0 \mapsto v_0;\ldots;r_{n-1} \mapsto v_{n-1};r_n \mapsto 0;\ldots;r_{|R|} \mapsto 0] :: (m,p,s,l) :: st,g,h>\end{array}}$$

$$\frac{m[p] = Return}{<(m,p,v :: s,l) :: (m',p',s',l') :: st,g,h> \rightarrow_P <(m',p'+1,v :: s',l') :: st,g,h>}$$

Figure 3: Operational semantics of the byte code language

5 Relational analysis of byte code

In this section, we describe a generic, relational analysis for byte code, parameterised with respect to a numeric relational domain used to abstract the values of the local and global variables of the program.

5.1 Symbolic analysis of the stack

Rather than treating each stack location as a new local variable and include this variable in the numeric abstraction describing the state, we integrate a *symbolic de-compilation* into the analysis that abstracts a stack location by a symbolic expression describing how the value at that stack location is computed from the values of the local variables. The operand stack is hence abstracted by a stack of symbolic expressions which represents relation between operands, static fields and local variables.

The following definition of expressions and guards has two purposes: they form the basis of the abstract domain for stacks (*Expr* only), which is specific to stack-based byte code, and they serve as the interface with the numeric relational domain, which is parametric. Note that those

two aspects of the analysis are completely independent apart from that.

$$\begin{array}{rll} \mathit{Expr}_V \ni e ::= & n \mid x \mid ? \mid e \diamond e & x \in V, \diamond \in \{+,-,\times,/\} \\ \mathit{Guard}_V \ni t ::= & e \bowtie e & \bowtie \in \{=,\neq,<,\leq,>,\geq\} \end{array}$$

The expression ? represents an unknown value and is responsible for the non-deterministic evaluation of expressions. Analyses will use this expression to model interactive inputs and abstract away numeric quantities not in the scope of the analysis. For instance, our analysis will not keep track of values stored in arrays.

The semantics $[\![e]\!]_\rho$ and $[\![t]\!]_\rho$ of expressions and guards with respect to an environment $\rho \in \mathbb{V} \to \mathbb{Z}$ are given below.

$$\begin{array}{rcl} [\![n]\!]_\rho = \{n\} \qquad [\![x]\!]_\rho = \{\rho(x)\} \qquad [\![?]\!]_\rho = \mathbb{Z} & & \\ [\![e_1 \diamond e_2]\!]_\rho & = & \{n_1 \diamond n_2 \mid n_1 \in [\![e_1]\!],\ n_2 \in [\![e_2]\!]\} \\ [\![e_1 \bowtie e_2]\!]_\rho & \iff & \exists\ n_1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho \quad n_1 \bowtie n_2 \end{array}$$

Note that this is not the whole concretisation function for symbolic expressions, which is described later (see Fig. 4).

Symbolic stacks Concrete operand stacks are abstracted by lists of symbolic expressions. To deal correctly with values which are returned after a method call we use auxiliary variables in a given set A, so the symbolic abstract domain for stacks is Expr^*_{R+S+A}.

5.2 Numeric relational domain specification

Apart from symbolic expressions stacks, the byte code analysis is specified with respect to an abstract numeric relational interface (defined below) that can be instantiated with standard relational abstract domains. We thus assume a domain $\mathbb{D}$ parameterised over a (finite) totally ordered set of variables V.

Language independent operators An abstract element is mapped to a set of environments in $\mathbb{Z}^V$ by the concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(\mathbb{Z}^V)$. To manage sets of variables, $\mathbb{D}$ is equipped with a projection operator $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$, an extension operator $\mathbb{E}_{V'} : \mathbb{D}_V \to \mathbb{D}_{V+V'}$ and a renaming operator $\cdot_{W \to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$.The abstract domain is also equipped with a partial order $\sqsubseteq \subseteq \mathbb{D}_V \times \mathbb{D}_V$ and meet and upper bound operators $\sqcap, \sqcup : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$. These components are language-independent.

Language dependent operators The abstract assignment of an expression $e \in \mathit{Expr}_V$ to a variable $x \in V$ is modelled by the operator $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$. A guard $t \in \mathit{Guard}_V$ may be abstracted by two operators $\mathit{assume}^\sharp(t), \mathit{ensure}^\sharp(t) : \mathbb{D}_v$: the $\mathit{assume}^\sharp$ operator computes an over-approximation of the guard, while $\mathit{ensure}^\sharp$ computes an under-approximation.

Definition 5.1 states formally the requirements over the operators of abstract domain $\mathbb{D}_V$.

Definition 5.1. *An abstract domain $\mathbb{D}$ is a family of sets $\mathbb{D}_V$ with:*

- *a concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(\mathbb{Z}^V)$,*
- *a decidable ordering relation $\sqsubseteq \subseteq \mathbb{D}_V \times \mathbb{D}_V$ such that*

$$d \sqsubseteq d' \Rightarrow \gamma(d) \subseteq \gamma(d'),$$

- *a projection* $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$*, an extension* $\mathbb{E}_{V'} : \mathbb{D}_V \to \mathbb{D}_{V+V'}$ *and a renaming* $\cdot_{W \to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$ *operators such that:*

$$\begin{aligned}\gamma(\exists_{V'}(d)) &= \{\rho_{|_V} \mid \rho \in \gamma(d)\}\\ \gamma(\mathbb{E}_{V'}(d)) &= \{\rho \mid \rho_{|_V} \in \gamma(d)\}\\ \gamma(d_{W \to W'}) &= \{\rho_{W \to W'} \mid \rho \in \gamma(d)\},\end{aligned}$$

- *a meet operator* $\sqcap : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$ *such that*

$$\gamma(d \sqcap d') = \gamma(d) \cap \gamma(d'),$$

- *an upper bound operator* $\sqcup : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$ *such that*

$$\gamma(d \sqcup d') \supseteq \gamma(d) \cup \gamma(d'),$$

- *an abstract assignment operator* $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$ s.t.

$$\gamma([\![x := e]\!]^\sharp(d)) \supseteq \{\rho[x \mapsto v] \mid \rho \in \gamma(d) \wedge v \in [\![e]\!]_\rho\},$$

- $assume^\sharp, ensure^\sharp : Guard_V \to \mathbb{D}_V$ *such that*

$$\gamma(ensure^\sharp(t)) \subseteq \{\rho \mid [\![t]\!]_\rho\} \subseteq \gamma(assume^\sharp(t)).$$

With the operator $assume^\#$ of the numerical domain we define the abstract test $[\![t]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$ of a guard $t \in Guard_V$ by:

$$\begin{aligned}[\![e \bowtie e']\!]^\sharp(l^\sharp) &= assume^\sharp(e \bowtie e') \sqcap l^\sharp \quad \text{if } \bowtie \in \{=, <, \leq, >, \geq\}\\ [\![e \neq e']\!]^\sharp(l^\sharp) &= (assume^\sharp(e' < e) \sqcap l^\sharp) \sqcup (assume^\sharp(e < e') \sqcap l^\sharp)\end{aligned}$$

The specific rule for $\neq$ is necessary to ensure a good precision with convex polyhedra.

5.3 Analysis specification

The byte code analysis is defined by specifying for each byte code an abstract transfer function which maps abstract states to abstract states (for non-jumping intraprocedural instruction at least). The abstract states are pairs of the form $(s^\sharp, l^\sharp)$ where $l^\sharp$ is a relation between local, global and auxiliary variables and $s^\sharp$ is an abstract stack whose elements are symbolic expressions built from these variables. More precisely, the analysis manipulates the following sets of variables:

R: set of local variables $r_0, \ldots, r_{|L|-1}$ of methods,

R_0: set of old local variables $r_0^{old}, \ldots, r_{|P|-1}^{old}$ of methods, representing their initial values t the beginning of method execution,

S: set of static fields $f_0, \ldots, f_{|S|-1}$ of the program

S_0: set of old static fields $f_0^{old}, \ldots, f_{|S|-1}^{old}$ of the program used to model values of static fields at the beginning of method execution

A: set of auxiliary variable $aux_0, \ldots, aux_{|A|-1}$ used to keep track of results of methods in the symbolic operand stack

Moreover, we use a "primed" version X' of the variable set X for renaming purposes. For each method the analysis computes a signature $Pre \to Post$ whose meaning is

> if the method is called with in a context where its arguments and the static fields satisfy the property *Pre* then if the method returns, then its result, its arguments, and the initial and final values of static fields satisfy the property *Post*.

instr	F_{instr}
Nop	$(s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$
Ipush n	$(s^\sharp, l^\sharp) \to \big(n :: s^\sharp, l^\sharp\big)$
Pop	$(e :: s^\sharp, l^\sharp) \to \big(s^\sharp, l^\sharp\big)$
Dup	$(e :: s^\sharp, l^\sharp) \to \big(e :: e :: s^\sharp, l^\sharp\big)$
Iadd	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e_2 + e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Isub	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e_2 - e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Imult	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e_2 \times e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Idiv	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e_2 / e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Ineg	$(e :: s^\sharp, l^\sharp) \to \big(\llcorner 0 - e \lrcorner :: s^\sharp, l^\sharp\big)$
Iinput	$(s^\sharp, l^\sharp) \to (? :: s^\sharp, l^\sharp)$
Load r	$(s^\sharp, l^\sharp) \to \big(\llcorner r \lrcorner :: s^\sharp, l^\sharp\big)$
Store r	$(e :: s^\sharp, l^\sharp) \to \big(s^\sharp[?/r], [\![r := e]\!]^\sharp(l^\sharp)\big)$
Getstatic f	$(s^\sharp, l^\sharp) \to \big(\llcorner f \lrcorner :: s^\sharp, l^\sharp\big)$
Putstatic f	$(e :: s^\sharp, l^\sharp) \to \big(s^\sharp[?/f], [\![f := e]\!]^\sharp(l^\sharp)\big)$
Iinc r n	$(s^\sharp, l^\sharp) \to \big(s^\sharp[\llcorner r - n \lrcorner/r], [\![r := r + n]\!]^\sharp(l^\sharp)\big)$
Newarray	$(e :: s^\sharp, l^\sharp) \to \big(e :: s^\sharp, l^\sharp\big)$
Arraylength	$(e :: s^\sharp, l^\sharp) \to \big(e :: s^\sharp, l^\sharp\big)$
Iaload	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(? :: s^\sharp, l^\sharp\big)$
Iastore	$(e_3 :: e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(s^\sharp, l^\sharp\big)$

$$\frac{m[p] = instr \notin \{Goto\ p', If\_icmp\ cond\ p', Invoke\ sig, Return\}}{F_{instr}(Loc(m,p)) \sqsubseteq Loc(m, p+1)}$$

$$\frac{m[p] = Goto\ p}{Loc(m,p) \sqsubseteq Loc(m,p)}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{\big(s^\sharp, [\![e_1\ cond\ e_2]\!]^\sharp(l^\sharp)\big) \sqsubseteq Loc(m,p')}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{\Big(s^\sharp, [\![e_1\ \overline{cond}\ e_2]\!]^\sharp(l^\sharp)\Big) \sqsubseteq Loc(m, p+1)}$$

$$\frac{m[p] = Invoke\ (mn,n) \quad ((m',n),c') \in P \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp)}{\Big(\exists_{R+S_0+A}\Big(\textstyle\prod_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\Big)\Big)_{S \to S_0} \sqsubseteq Pre((mn,n),c')}$$

$$\frac{m[p] = Invoke\ sig \quad (sig,c') \in P \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp)}{\left(\begin{array}{l} \llcorner aux_j \lrcorner :: s^\sharp[?/aux_j], \\ \exists_{S'+\{res\}} [\![aux_j := res]\!] \left(\begin{array}{l} l^\sharp_{S \to S'} \\ \sqcap\ \exists_{R_0}\left(\begin{array}{l} \prod_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old})_{S \to S'} \\ \sqcap\ Post(sig,c')_{S_0 \to S'} \end{array}\right) \end{array}\right) \end{array}\right) \sqsubseteq Loc(m, p+1)}$$

where p is the index of the $j-$th *Invoke* in m

$$\frac{m[p] = Return \quad Loc(m,p) = (e :: s^\sharp, l^\sharp)}{\exists_{R+A}([\![res := e]\!]^\sharp(l^\sharp)) \sqsubseteq Post(m)}$$

$$\frac{(m,n,c) \in P}{\prod_{i=0}^{|S|-1} assume^\sharp(f_i = f_i^{old}) \prod_{i=0}^{n-1} assume^\sharp(r_i^{old} = r_i) \sqcap Pre(m) \sqsubseteq Loc(m,0)}$$

$$\frac{((\text{main},0),c) \in P}{\top \sqsubseteq Pre((\text{main},0),c)}$$

Figure 4: Relational byte code analysis with stack de-compilation

Preconditions are actually chosen by over-approximating the context in which each method may actually be invoked. Additionally the analysis computes at each control point of each method a local invariant between the current (R) and initial (R_0) values of local variables, the current (S) and initial (S_0) values of static fields, and some auxiliary variables (A) which are used temporarily to remember results of method calls which are still on the stack

Definition 5.2 (Abstract domain)**.** *The abstract value for a program P is described by an element $(Pre, Post, Loc)$ of the lattice*

$$\begin{aligned} State^\# = \quad & Meth \to \mathbb{D}_{R_0+S_0} \\ & \times\ Meth \to \mathbb{D}_{R_0+S_0+S+\{res\}} \\ & \times\ Meth \times \mathbb{N} \to \big(Expr_{R+S+A^\star} \times \mathbb{D}_{R_0+S_0+R+S+A}\big)_\bot \end{aligned}$$

The analysis is specified as a solution of a constraint (inequation) system associated to each program. The constraint system is formally defined in Fig 4. Note that extensions are left implicit. For non-jumping intraprocedural instructions, the constraint is defined via a transfer function in $Expr^\star \times \mathbb{D}_{R_0+S_0+R+S+A} \to (Expr^\star \times \mathbb{D}_{R_0+S_0+R+S+A})_\perp$. We (ab)use notation and write $(e :: s^\sharp, l^\sharp) \to \left(e :: e :: s^\sharp, l^\sharp\right)$ for the function that maps a state of the form $(e :: s^\sharp, l^\sharp)$ to the resulting state $\left(e :: e :: s^\sharp, l^\sharp\right)$ and other states to $\perp$. The analysis maintains a symbolic version of the operand stack and most of the transfer functions are defined as symbolic executions. The transfer functions for the stack operations *Nop*, *Pop* and *Dup* mimic the semantics of those operations so *e.g.*, *Dup* will duplicate the expression on top of the (abstract) operand stack and hence is abstracted by the function $(e :: s^\sharp, l^\sharp) \to \left(e :: e :: s^\sharp, l^\sharp\right)$. The abstraction of the instruction *Load* r for fetching the value of local variable r just pushes the expression $\llcorner r \lrcorner$ onto the abstract stack (rather than projecting an abstract value of r from the relation describing the local variables). Similarly, the abstraction of the addition operation *Iadd* pops the two topmost expressions e_1 and e_2 from the abstract stack and replaces them with the symbolic expression $\llcorner e_2 + e_1 \lrcorner$.

The transfer function for the *Store* r operation updates the abstract environment of local variables with the constraint that r is now equal to the value given by the expression e on top of the abstract stack top. Formally, this is done using the operation $[\![x := e]\!]^\sharp$ provided by the interface of the relational domain. By the same token, all occurrences of the sub-expression $\llcorner x \lrcorner$ in the abstract stack become invalid, as r now (potentially) has changed value, and are replaced by the "don't know" expression $\llcorner ? \lrcorner$. The analysis abstracts arrays references by the length of the referenced array, so the transfer functions for *Newarray*(which takes the length as argument and returns a reference to the created array) becomes the identity function. Similarly for *Arraylength*.

For all non-jumping instructions, we generate a constraint saying that the state following the instruction should include the result of applying the transfer function of the instruction to the state preceding the instruction. For the conditional *If_icmp cond p'*, we use the abstract tests provided by the relational domain to take the outcome of the test into account, so *e.g.*, at program point p' we know that the condition *cond* holds between the two top elements of the stack. If these are given by expressions e_1 and e_2 then we know that the symbolic expression $\llcorner e_1 \ cond \ e_2 \lrcorner$ evaluates to true in the current environment. The expression $[\![e_1 \ cond \ e_2]\!]^\sharp(l^\sharp)$ in the rule for conditionals updates the environment of local variables ($l^\sharp$ to take this information into account. A similar constraint is generated for the program point $p+1$ using this time the negation $\overline{cond}$ of the condition *cond*.

The analysis of method calls is the most complicated part. The complications partly arise because we have several kinds of variables (static fields, local and auxiliary variables) whose different scope must be catered for. The analysis gives rise to two constraints: one that relates the state before the call to the pre-condition of the method and one that registers the impact of the call on the state immediately following the call site.

When invoking a method m' from method m, we compute an abstract state that holds before starting executing m' and which constrains the $Pre(m')$ component of the abstract element describing m'. This state registers that the n topmost expressions $e_1, \ldots, e_n$ on the abstract stack corresponds to the actual arguments that will be bound to the local variables of the callee m', by injecting the constraints $e_i = r_i^{old}$ into the relational domain and adding them to the current state as given by $l^\sharp$. Care must be exercised not to confound the parameters R_0 of the caller with the parameters of the callee, hence the projecting out of R_0 before joining the constraints. Furthermore, the local variables R, the initial values of static fields S_0 and the auxiliary variables A of method m have a different meaning in the context of method m' and are removed from the abstract state at the start of m' too. Finally, the current value of static fields S in m at the point of the method call becomes the initial value of the static fields when analysing m', hence the renaming of S into S_0. The entire start state for m' is thus described by the expression

$$\left(\exists_{R+S_0+A}\left(\prod_i assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\right)\right)_{S \to S_0}$$

The second rule for *Invoke* describes the impact of the method call on its successor state. We use an auxiliary variable aux_j (chosen to be free in $s^\#$) to name the result of a method call which is pushed onto the stack. This variable is constrained to be equal to the variable *res* which receives the value returned by m'. The rest of the left-hand side expression of the constraint

$$l^\sharp_{S\to S'} \sqcap \exists_{R_0} \left(\prod_i assume^\sharp(e_i = r_i^{old})_{S\to S'} \sqcap Post(m')_{S_0\to S'} \right)$$

serves to link the post-condition $Post(m')$ of the method with the state $l^\sharp$ of the call site. These are linked via the local variables x_i constrained to be equal to the argument expressions e_i and via the global static fields S. Again, some renaming and hiding of variables is required: *e.g.*, the initial values of the static fields in m', referred to by S_0, correspond to the values of the static fields before the call in the state $l^\sharp$ and in the expressions e_i, referred to by S. The renamings $S_0 \to S'$ and $S \to S'$, respectively, ensures that these values are identified.

Two rules are used to initiate the analysis of a method (constraint on $Loc(m, 0)$) and of the entire program (constraint on $Pre((\text{main}, n), c)$). To initialise the analysis of a method m, the precondition $Pre(m)$ is conjoined with the constraints linking the variable f_i^{old} to the current value of the static field f_i and linking the parameters r_i^{old} with the local variables r_i, in accordance with how parameters are handled in *e.g.* Java byte code. The analysis of the main method starts in the completely unconstrained state $\top$.

5.4 Inference

The constraint system presented in the previous section can be turned into a post-fixpoint problem by standard techniques. Consequently, the solutions of the system can be characterised as the set of post-fixpoints $\{x \mid F^\sharp(x) \sqsubseteq x\}$ of a suitable monotone operator $F^\sharp \in State^\sharp \to State^\sharp$ operating on the global abstract domain $State^\sharp$ of the analysis. Assuming that $State^\sharp$ is a complete lattice[1] we know that the least solution $lfpF^\sharp$ of this problem exists and can be over-approximated by any post-fixpoint of $F^\sharp$. Computing such a post-fixpoint is the role of chaotic iterations [14] which operate on the equation system associated with the constraint system and choose a suitable iteration strategy [9]. Iteration is sped up by using widening on well-chosen control points. Neither the iteration strategy nor the widening operators belong to the TCB since the validity of the result can be checked with a post-fixpoint test.

5.5 Safety checks

Once the analysis has inferred correct invariants, this information is used to check if they enforce the suitable safety policy. In a context of array bound checking we must check that each array access is within the bounds of the array. As a consequence, for each occurence of an instruction *Iaload* or *Iastore* at a program point (m, pc), we test if the local invariant $Loc(m, pc)$ computed by the analysis ensures a safe array access.

Definition 5.3 (Abstract safety checks)**.** *We say a set of local invariant* $Loc \in (\mathbb{N} \to (Expr^\star \times \mathbb{D}_{P+S_0+L+S+A})_\bot)$ *verifies all safety checks of a program if and only if*

$$\begin{array}{l}
\forall m \in P,\ pc \in \mathbb{N}, \\
\quad m[p] = Iaload \ \Rightarrow \\
\qquad Loc(m, pc) = (e_2 :: e_1 :: s^\sharp, l^\sharp) \Rightarrow \\
\qquad l^\sharp \sqsubseteq ensure^\sharp(\lfloor 0 \le e_2 \rfloor) \ \wedge \ l^\sharp \sqsubseteq ensure^\sharp(\lfloor e_2 < e_1 \rfloor) \\
\quad \wedge \\
\quad m[p] = Iastore \ \Rightarrow \\
\qquad Loc(m, pc) = (e_3 :: e_2 :: e_1 :: s^\sharp, l^\sharp) \Rightarrow \\
\qquad l^\sharp \sqsubseteq ensure^\sharp(\lfloor 0 \le e_2 \rfloor) \ \wedge \ l^\sharp \sqsubseteq ensure^\sharp(\lfloor e_2 < e_1 \rfloor)
\end{array}$$

[1] For the polyhedra abstract domain this assumption is too strong but we can relax it by considering a complete lattice containing $State^\sharp$ and all its upper bounds [15].

$$
\begin{array}{rrcl}
\beta_{\mathbb{V}} : & \mathit{Heap} \times (\mathbb{V} \to \mathit{Val}) & \to & (\mathbb{V} \to \mathbb{Z}_\bot) \\
& (h, z) & \mapsto & \lambda x. \left| \begin{array}{ll} z(x) & \text{if } z(x) \in \mathbb{Z} \\ |h(z(x))| & \text{if } z(x) \in \mathit{dom}(h) \end{array} \right. \\
\gamma^{\mathit{expr}}_{h,g,l,a} : & \mathit{Expr}_{R+S+A} & \to & \mathcal{P}(\mathit{Val}), \quad h \in \mathit{Heap}, g \in \mathit{Static},\ l \in \mathit{LocVar} \\
& & & \qquad \text{and } a \in \mathit{Val} \to \mathbb{Z} \\
& e & \mapsto & [\![e]\!]_{\beta_{R+S+A}(h, l \oplus g \oplus a)} \\
& & & \cup \left\{ \mathit{ref} \in \mathit{dom}(h) \;\middle|\; |h(\mathit{ref})| \in [\![e]\!]_{\beta_{R+S+A}(h, l \oplus g \oplus a)} \right\}
\end{array}
$$

$$
\begin{array}{rrcl}
\gamma_{\mathit{Pre}} : & \mathbb{D}_{R_0+S_0} & \to & \mathcal{P}(\mathit{Static} \times \mathit{Heap} \times \mathit{LocVar}) \\
& \mathit{pre} & \mapsto & \{\ (g_0, h_0, l_0) \mid \beta_{R_0+S_0}(h_0, l_0 \oplus g_0) \in \gamma(\mathit{pre})\ \} \\
\gamma_{\mathit{Post}} : & \mathbb{D}_{R_0+S_0+S+\{\mathit{res}\}} & \to & \mathcal{P}((\mathit{Static} \times \mathit{Heap} \times \mathit{LocVar}) \times (\mathit{Static} \times \mathit{Heap} \times \mathit{Val})) \\
& \mathit{post} & \mapsto & \left\{ \begin{array}{l} ((g_0, h_0, l_0), (g, h, v)) \mid \\ \quad \beta_{S_0+R_0+S+\{\mathit{res}\}}(h_0, g_0 \oplus l_0 \oplus g \oplus [\mathit{res} \mapsto v]) \in \gamma(\mathit{post}) \end{array} \right\} \\
\gamma_{\mathit{Loc}} : & \mathit{Expr}^\star \times \mathbb{D}_{R_0+S_0+R+S+A} & \to & \mathcal{P}\left(\begin{array}{l} (\mathit{Static} \times \mathit{Heap} \times \mathit{LocVar}) \\ \times\ (\mathit{Static} \times \mathit{Heap} \times \mathit{Stack} \times \mathit{LocVar}) \end{array} \right) \\
& (e_1 :: \cdots :: e_n, \mathit{loc}) & \mapsto & \left\{ \begin{array}{l} ((g_0, h_0, l_0), (g, h, v_1 :: \cdots :: v_n, l)) \mid \\ \quad \exists a \in A \to \mathbb{Z},\ \forall i \in [\![1, n]\!],\ v_i \in \gamma^{\mathit{expr}}_{h,g,l,a}(e_i) \wedge \\ \quad\quad \beta_{S_0+R_0+S+R+A}(h_0, g_0 \oplus l_0 \oplus g \oplus l \oplus a) \in \gamma(\mathit{loc}) \end{array} \right\}
\end{array}
$$

Figure 5: Concretisation functions

5.6 Soundness of the analysis

Fig. 5 gives the concretisation functions for the abstract domains. The auxiliary abstraction function β maps everything to an integer, abstracting arrays by their length. γ^{expr} defines concretisation of a symbolic expression with respect to an environment. γ^{Pre} maps pre-conditions to sets of calling contexts, γ^{Post} maps post-conditions to relations between calling contexts and return contexts, and γ^{Loc} maps local invariants to relations between calling contexts and local program states. Note that concretisations contain only states such that all locations that are being referenced are defined in the heap.

Definition 5.4 (Reachable states)**.** *For a method m in a program P, a heap h, a static heap g, a set of local variables l, a frame stack st, the set $[\![P]\!]^m_{h,g,l,st}$ of reachable state from an execution of m starting in an initial configuration (h, g, l, st) is defined by*

$$[\![P]\!]^m_{h,g,l,st} = \left\{ s \mid <(m, 0, [], l') :: st, g, h> \xrightarrow{\geq st}{}^*_P\ s \right\}$$

where $\xrightarrow{\geq st}{}^*_P$ is the reflexive transitive closure of $\to_P$ restricted to states who have a form $< \ldots :: (m, \ldots) :: st, \ldots >$.

The purpose of $\xrightarrow{\geq st}{}^*_P$ is to collect only the states in between the start and the end of the execution of a particular stack frame.

Definition 5.5 (Safe method)**.** *A method m in a program P is said to be* safe *wrt. a precondition $Pre \subseteq Heap \times Static \times LocVar$ if for all stack frames st and all $(h, g, l) \in Pre$, $Error \notin [\![P]\!]^m_{h,g,l,st}$.*

Theorem 5.6 (Correctness)**.**
Let P be a program and $(Pre, Post, Loc)$ a solution of the constraint system associated with P. If Loc satisfies all safety checks then every method m in P is safe wrt. to $Pre(m)$. In particular,

$$<(((main, n), c), 0, [], \lambda r.0) :: [], \lambda f.0, \lambda ref.\bot > \not\to_P Error$$

Proof. The proof is divided into two parts. We first prove that each reachable intermediate state at a point (m, p) satisfies the property $\gamma_{Loc}(Loc(m, p))$, that each method m is called in a context satisfying $\gamma_{Pre}(Pre(m))$ and that its return value (if it exists) satisfies $\gamma_{Post}(Loc(m))$. In the second part we prove that if a state at some point (m, p) satisfies $\gamma_{Loc}(Loc(m, p))$ as well as the abstract safety check associated with this point, then no error happens in the next semantic step. To deal with the steps corresponding to procedure calls, the proof makes use of an intermediate big-step operational semantics. Details are omitted for lack of space. □

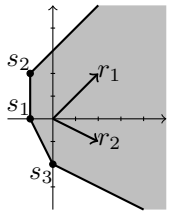


Figure 6: Dual representation of polyhedra

6 Polyhedral analysis

We now instantiate the relational analysis framework using linear relations in the form of convex polyhedra. Polyhedral program analysis has a well-established theory [15] with several implementations [4, 21]. Here, we recall the basics of this theory.

Definition 6.1. *Convex polyhedra of dimension n ($\mathbb{P}_n \subseteq \mathbb{Q}^n$) are (convex) subsets of $\mathbb{Q}^n$ that can be expressed as a finite intersection of half-planes of $\mathbb{Q}^n$.*

Polyhedra can be represented as sets of linear constraints. It is desirable to keep these sets in normal form *i.e.,* without redundant constraints. For this purpose, polyhedra libraries maintain a dual representation of polyhedra based on *generators* in which a convex polyhedron is the convex hull of a (finite) set of *vertices*, *rays* and *lines*. Vertices, rays and lines are respectively extremal points, infinite directions and bi-directional infinite directions of the polyhedron. Fig. 6 shows a a polyhedron with four constraints whose dual representation is made of three *vertices* (s_1,s_2,s_3) and two *rays* (r_1,r_2).

The efficiency of the algorithm that maintains the normal form of the double description is of crucial importance. For this task, state-of-the-art polyhedral libraries [4, 21] use Chernikova's algorithm [13]. In the worst case, the number of generators is exponential in the number of constraints (and *vice-versa*) but, in practise, the double description offers a good performance. To alleviate further the cost of normalising polyhedra, these libraries switch lazily from one representation to the other.

Polyhedral cannot directly handle expressions that fall outside the linear fragment. It would be sound but unsatisfactory to abstract those expressions towards an arbitrary value *i.e.*, the ? expression. More information can be retained by *linearising* expressions [26]. For instance, the precise analysis of Binary Search (Fig. 1) requires a precise model of euclidean divisions. Given an integer constant n, the guard $y = x/n$ is abstracted by the linear guards $0 \leq x - n \cdot y < n$. Multiplications can also be linearised by using the range of variables.

We now briefly explain how polyhedral algorithms implement the abstract numeric relational domain specified in Definition 5.1. To be implemented efficiently, the double description of polyhedra is needed, using Chernikova's algorithm to reconstruct the coherence of the double representation.

The convex polyhedron can directly be cast into an abstract numeric domain by mapping variables of the domain to dimensions of the polyhedron. Hence, we get $\mathbb{D}_V = \mathbb{P}_{|V|}$ and the concretisation:

$$\gamma(P) = \{\rho \in \mathbb{Z}^V \mid \rho \in P \cap \mathbb{Z}^V\}$$

Renaming of variables consists in applying a permutation to the dimensions of polyhedron. The **extension** operation which add new variables consists in inserting new unconstrained dimensions at the relevant indexes.
Projections can be efficiently performed on the *generator* description of polyhedra in linear time. Each generator is projected by erasing the now irrelevant dimensions.
Intersections are computed by taking the union of the constraints of each polyhedron.
The **convex hull**, *i.e.*, least upper bound, is computed by taking the union of the generators of both polyhedra.

An **assignment** $[\![x := e]\!]^\sharp$ is modelled by the linear transformation (if e is linear) that keeps all the variables unchanged except x which is mapped to e. The transformation is applied to the generators.
Inclusion tests are using both representation at once. Checking the containment of two polyhedra ($P \sqsubseteq Q$) amounts to verifying that the generators of P satisfy the constraints of Q.
Widening operators are used by the fixpoint iterator to ensure convergence. For convex polyhedra, there exist various widening operators [15, 3].
Assume and ensure operators are responsible for interpreting guards of the target language. If the guard t is linear, a polyhedron is built from it and no abstraction takes place. Otherwise, t has to be linearised. In the worst case, universal (resp. empty) polyhedra can be used as sound (though very imprecise) fallbacks.

7 Fixpoint pruning

The result of the polyhedral byte code analysis will be a fixpoint of the transfer functions, representing an invariant of the program under analysis. This invariant will often contain more information than necessary for proving a particular safety policy such as absence of indexing outside array bounds. In the following we show how to *prune* an invariant with respect to a given safety policy, resulting in an invariant that is smaller and cheaper to verify.

7.1 Witnesses and pruning

We have applied the technique described in [7] for pruning constraint-based invariants, with some adaptations allowing to handle our interprocedural polyhedral analysis on byte code better. First we recall the definition of witnesses for this particular analysis.

Definition 7.1. *A witness for a program P is a solution (Pre, Post, Loc) to the constraint system associated with P that satisfies the safety checks of P (see Definition 5.3).*

We use this as the basis for building certificates, relying on the fact that if there exists a witness for P then P is safe (see Theorem 5.6). Part of the witness is sent to the checker in the constraint representation only (see Section 6), so we aim at extracting a weaker witness with fewer linear constraints than the one produced by the inference algorithm of Section 5.4 (if the analysis is accurate enough for the program). Pruning leaves the symbolic expression stacks of the witness unchanged because the checker recomputes them (and hence nothing is transmitted about this part).

It is easy to see that there is generally no unique weakest witness nor a unique witness with the minimum number of constraints (because the analysis is not *distributive*). Also, the idea of starting from the safety requirements to compute backward a witness that satisfies them cannot achieve the same precision as a forward analysis, because intuitively it would have to guess the invariants that a forward analysis naturally discovers. For these reasons we use a technique of pruning that removes as many linear constraints as possible from a given witness.

7.2 Abstract algorithm

We use a variation of the greedy heuristic presented in [7]. In the following we identify polyhedra with sets of constraints. We use

$$\begin{aligned} Var = \quad & \{\mathrm{pre}_m \mid m \in P\} \cup \{\mathrm{post}_m \mid m \in P\} \\ & \cup \{\mathrm{loc}_{m,p} \mid m \in P,\ m = ((mn, n), c),\ p < |c|\} \end{aligned}$$

to denote the set of unknowns of the constraint system associated with P. For an abstract element $x = (Pre, Post, Loc)$ we define the set of linear constraints of x:

```
prune(w) :=
  let w' = ∅
  while w' is not a witness do
    choose a constraint C and k ∈ w'_|dep(C) s.t. w', {k} ⊬ C
      (or a check C such that w' ⊬ C)
    choose x ⊆ (w \ w')_|dep(C) such that w' ∪ x, {k} ⊢ C
      (respectively, w' ∪ x ⊢ C)
    w' := w' ∪ x
  done
  return w'
```

Figure 7: Witness pruning algorithm

$$\overline{x} = \bigcup_{m \in P} \begin{array}{l} \{(\mathrm{loc}_{m,p}, k) \mid Loc(m,p) = (s^\#, l^\#),\ k \in l^\#\} \\ \cup \{(\mathrm{pre}_m, k) \mid k \in Pre(m)\} \\ \cup \{(\mathrm{post}_m, k) \mid k \in Post(m)\} \end{array}$$

For $V \subseteq Var$ we define $\overline{x}_{|V} = \{(var, k) \in \overline{x} \mid var \in V\}$ and $x_{|V}$ is defined accordingly.

Recall that the constraint system for P is a set of constraints of the form $F(x) \sqsubseteq x_{|\{v\}}$ where $v \in Var$. For a constraint c we note $\overline{x}, \overline{y} \vdash C$ if $F(x) \sqsubseteq y_{|\{v\}}$ (we can do so since the expression stacks are fixed) and $\overline{x} \vdash C$ for $\overline{x}, \overline{x} \vdash C$. We will overload the notation and write also $\overline{x} \vdash C$ if x satisfies the safety check C. Then, for every such constraint C, we define a set $dep(C) \subseteq Var$ that represents the dependencies of this constraint, in the sense that if $\overline{x}, \overline{y} \vdash C$ then $\overline{x}_{|dep(C)}, \overline{y} \vdash C$. The definition of dep is straightforward. For example, if C is the constraint $F_{instr}(Loc(m,p)) \sqsubseteq Loc(m,p+1)$ corresponding to an non-jumping intraprocedural instruction (see the first part of Fig. 4), then $dep(C) = \{\mathrm{loc}_{m,p}\}$. For the constraint $\ldots \sqsubseteq Loc(m,p+1)$ of an *Invoke sig* instruction, $dep(C) = \{\mathrm{loc}_{m,p}, \mathrm{post}_{(sig,c)}\}$ where $(sig, c) \in P$.

The pruning algorithm is shown in Fig. 7. The main issue in this non-deterministic algorithm is the choice of the subset $\overline{x}$: we obviously want a minimal one in the sense of set inclusion (achievable in reasonable time by monotonicity), but it is not unique.

7.3 Efficient pruning for polyhedral byte code analysis

Our strategy is to take a minimal such $\overline{x}$ that almost minimizes a cost function taking into account the number of linear constraints, the number of non-null coefficients in them, and, for *Invoke*, the number of post constraints (as opposed to loc). This allows us to obtain a witness with simpler invariants and signatures. The heuristic blindly applies the definition of $\vdash$ while labelling (part of) the search space. The dependency function *dep* helps by reducing the number of linear constraints to be considered at each step.

Finally, we face a problem specific to the polyhedra domain when pruning an invariant: in order to keep things small, the polyhedra are usually represented in a minimal form in which the relation between a set of dimensions does not necessarily appear as a dedicated linear constraint, but often as a consequence of several other relations. For example, the constraint $x \leq z$ is implicit in $x \leq y \leq z$. For the purpose of finding a small invariant, we may benefit from being able to include such constraints. Our solution is to add some implicit constraints to the invariant before pruning it. More precisely, for a polyhedron in $\mathbb{D}_V$, we add all the projections $\exists_{V \setminus V'}$ (see Section 6) where V' is a subset of V of cardinality at most n. For the maximal number n of dimensions in the implicit constraints to be generated, ∞ seems too costly for non-trivial programs, and unnecessary. It turns out that 3 is enough for all of our examples, which is not surprising because very few correctness proofs actually rely on linear invariants involving more than three variables.

8 Result checking of polyhedral analysis

A result checker for abstract interpretation based static analysis can be reduced to an (optimised) fixpoint checker [1], with the downside that the abstract domains are still part of the TCB. Formally certifying optimised polyhedral libraries [4, 21] is feasible but would require an enormous certification effort. Instead, we propose a lightweight verifier of polyhedral analyses using a result checking methodology which has two advantages: i) the TCB is small, and ii) the checking time is optimised.

8.1 The polyhedral domain revisited

Chernikova's algorithm is at the origin of the computational complexity of convex polyhedra operations, so a first approach would be to design a result checker for Chernikova's algorithm *i*.e., a normal form checker. This has the inconvenience that most of the polyhedral operations would be annotated with their result together with a certificate attesting that it is in normal form. Instead, we develop a checker which only uses the constraint representation of polyhedra and which never need to normalise. Moreover, projections are not computed but delayed using a set of extra *existential* variables. More precisely, our polyhedra are represented by a list of linear expression over two disjoint sets of variables V and E. Variables in $v \in V$ are genuine variables while $e \in E$ are (existential) variables that represent dimensions which have been projected out.

Definition 8.1. *Let V and E be disjoint sets of variables.*

$$\mathbb{P}_V = Lin^*_{V+E}$$

where

$$Lin_{V+E} = \{\lfloor c_1 \times x_1 + \cdots + c_n \times x_n \rfloor \mid c_i \in \mathbb{Z} \wedge x_i \in V + E\}.$$

Given $es \in \mathbb{P}_V$, the concretisation function is defined by

$$\gamma_V(es) = \{\rho_{|V} \mid \forall k \in es, [\![k \geq 0]\!]_\rho\}$$

In the following, we show how to implement the polyhedral operations using (only) polyhedra in constraint form.

Renaming simply consists in applying the renaming to the expressions within the polyhedron. Because the existential variables belong to a disjoint set, no capture can occur. In addition, for this encoding, **extension** is a no-op because unused variables have no impact on the internal representation.

$$es \in \mathbb{P}_V \Rightarrow \forall W \supseteq V, es \in \mathbb{P}_W$$

Using Fourier-Motzkin elimination (see *e.g.*), [31], **projections** can be computed directly over the constraint representation of polyhedra However, in the worst case, the number of constraints grows exponentially in the number of variables to project. To solve this problem, we delay the projection and simply register them as existentially quantified. This is done by renaming these variables to fresh variables.

To compute **intersections**, care must be taken not to mix up the existential variables. To avoid capture, existentially variables are renamed to variables that are fresh for both polyhedra. Thereafter, the intersection is implemented by taking the union of the expressions.

To implement the **assume** and **ensure** operators, the involved expressions are first linearised and the obtained linear inequality is put into the form $e \geq 0$ which now belongs to the set Lin defined above.

For convex polyhedra, **assignment** is efficiently implemented as an atomic operation. However, it can be expressed in terms of the previous operators: given x' a fresh variable, an assignment can be defined as follows.

$$[\![x := e]\!]^\sharp(P) = \left(\exists_{\{x\}} \left(P \sqcap assume^\sharp(x' = e)\right)\right)_{\{x'\} \to \{x\}}$$

It is this latter definition that we use.
Widening operators are only used during the fixpoint iteration, and are not needed at checking time.
Convex Hull is the typical operation that is straightforward to implement using the generator representation of polyhedra. Using a relaxation technique, it is possible to express the convex hull as the projection of a polyhedron of higher dimension [2] but since this requires to compute projections this does not scale. Even with our delaying of projections, the size of the polyhedron doubles. Instead of computing a convex hull, we follow the result certification methodology and provide a certificate polyhedron that is the result of the convex hull computation. Furthermore, our result checker need not check that the result is exactly the convex hull but only that it is an upper bound by doing a double inclusion test.

$$isUpperBound(P, Q, UB) \equiv P \sqsubseteq UB \wedge Q \sqsubseteq UB$$

To implement **inclusion tests**, we push the result certification methodology further and use inclusion certificates. The form of certificates and their generation are described below.

8.2 Result certification for polyhedral inclusion

Farkas lemma (Lemma 8.2) is a theorem of linear programming (see for instance [31]) which gives a notion of *emptiness* certificate for polyhedra. In this part, we show how this result can be i) lifted to obtain an inclusion checker; ii) extended further to deal with existential variables. Our inclusion checker $\sqsubseteq_{check}$ takes as input a pair of polyhedra (P, Q) and an inclusion certificate. It will only return true if the certificate allows to conclude that P is indeed included in Q ($P \sqsubseteq Q$).

Lemma 8.2 (Farkas Lemma)**.** *Let $A \in \mathbb{Q}^{m\times n}$ and $b \in \mathbb{Q}^n$. The following statements are equivalent:*

- *For all $x \in \mathbb{Q}^n$, $\neg(A \cdot x \geq b)$*
- *There exists $ic \in \mathbb{Q}^m$ satisfying $A^t \cdot ic = \bar{0}$ and $b^t \cdot ic > 0$.*

The soundness ($\Leftarrow$) of certificates is the easy part and is all that is needed in the machine-checked proof. It follows that the existence of a certificate ensures the infeasibility of the linear constraints and therefore that the polyhedron made of these constraints is empty.

Thus, an *inclusion certificate* ic is a vector of $\mathbb{Q}^m$ and checking a certificate consists of 1) computing a matrix-vector product ($A^t \cdot ic$) 2) verifying that the result is a null vector; 3) computing a scalar product ($b^t \cdot ic$); and 4) verifying that the result is strictly positive. All in all, the certificate checker runs in quadratic-time in terms of arithmetic operations.

Certificates generation can be recast as a linear programming problem that can be efficiently solved by either the Simplex or interior point methods. The set of certificates is characterised by the convex polyhedron

$$Cert = \left\{ ic \,\middle|\, ic \geq \bar{0} \wedge b^t \cdot ic > 0 \wedge A^t \cdot ic = \bar{0} \right\}$$

As a result, finding an *extremal* certificate amounts to solving a linear optimisation problem. For instance, the solution of the linear program $min\{c^t \cdot \bar{1} \mid c \in Cert\}$ minimises the sum of the coefficients of the certificate. In theory, such a minimisation might not yield a compact certificate because the optimisation is done over the rationals – there are very small rationals that require many bits. However, in practise, the technique is sufficiently efficient.

From emptiness to inclusion Lemma 8.3 states that in the absence of existential variables an inclusion check amounts to emptiness checks.

Lemma 8.3. *Given $P, P' \in \mathbb{P}_{V+E}$, we have*

$$\forall e' \in P', \gamma_{V+E}(-e'-1 :: P) = \emptyset$$

if and only if

$$\gamma_{V+E}(P) \subseteq \gamma_{V+E}(P').$$

Proof. By construction, polyhedra in $\mathbb{P}_{V+E}$ do not have existential variables. Hence, we have $\gamma_{V+E}(P') = \bigcap_{e' \in P'} \gamma_{V+E}(e'::[])$. Moreover, the complement of a linear constraint $e' \geq 0$ is $-e' - 1 \geq 0$. These facts allow to reduce inclusion to a set of emptiness tests. □

Lemma 8.4 states that to do an inclusion test, it is sound to drop existential variables.

Lemma 8.4. *Let P and P' be polyhedra in constraint form.*

$$\gamma_{V+E}(P) \subseteq \gamma_{V+E}(P') \Rightarrow \gamma_V(P) \subseteq \gamma_V(P')$$

Proof. The Lemma follows from the definition of γ and the fact that the restriction operator on environments is monotone. □

Together, Lemma 8.3 and Lemma 8.4 allow the design of a sound result checker for inclusion tests of form $P \subseteq P'$. In general, the checker is incomplete but this only shows up in cases where P' has existential variables. However, inclusions only need to be certified when P' is a polyhedron computed by the analyser and such a P' *does not* contain existential variables, so the inclusion checker is always used in a context where it is complete.

9 Implementation and Experiments

The relational bytecode analysis has been implemented in Caml and instantiated with the efficient NewPolka polyhedral library [21] as its relational abstract domain. The programs we analyse are genuine Java programs where unsupported instructions have been automatically replaced by conservative numerical instructions *e.g.*, *Getfield* replaces the top-most element of the stack by an arbitrary value. The analyser then computes a solution to the constraint system generated from a program. From these invariants, loop headers and join points are extracted. Inclusion certificates required by the checker are generated using the GNU Linear Programming Toolkit [24] which features a Simplex computing in exact rational arithmetic. Loop headers and join point invariants constitute (the part of) the analyser result that is sent to the checker. The certificate is made of the inclusion certificates.

As invariants computed by static analysers often contain more information than necessary for proving a particular safety policy *i.e.,* the absence of array out-of-bounds accesses, it is interesting to *prune* the analysis result and eliminate invariants that are useless for proving a given safety property. The advantages are twofold: invariants to check are smaller and their verification cheaper. We have adapted the technique described in [7] for pruning constraint-based invariants, thus allowing to handle our interprocedural polyhedral analysis (Section 7). For our benchmarks, pruning can halve the number of constraints to verify. This reduction can sometimes but not always produce a similar reduction in checking time. The reduction is especially visible when the analyser tends to generate huge invariants which cannot be exploited. This is *e.g.*, the case for FFT where the analyser approximates an exponential with a complex polyhedron without any positive effect on the number of successful safety checks.

For each program we provide the checking time with after fixpoint pruning, using either an extracted checker (Caml) or the checker running in Coq. In the first approach the Coq result checker is automatically transformed into a Caml program by the Coq extraction mechanism. In the second approach, the result checker is directly run inside the reduction engine of Coq to compute a foundational proof of safety of the program. Fig. 8 presents our experimental results. The benchmarks are relatively modest in size and do not use that many variables and it is well

known that full-blown polyhedral analyses have scalability problems. Our analyser will not avoid this but can be instantiated with simpler relational domains such as *e.g.*, octagons, without having to change the checker. The programs and the analysis results can be found on-line [35] and replayed in Coq or with an extracted Caml checker. We consider two families of programs. The first one consists of benchmarks used by Xi to demonstrate the dependent type system for Xanadu [36]. For this family we automatically prove the absence of out-of-bound accesses. The second is taken from the Java benchmark suite SciMark for scientific and numerical computing where our polyhedral analysis prove safety for array accesses except for the more intricate multi-dimensional arrays representing matrices. This explains why certain scores are below 100%. When the analyser cannot prove all the array accesses safe, we obtain a certificate by using a refined version of the safety property where all but a designated subset of array accesses are required to be correct.

Program	size	score	#variables	certificate size	checking time (Caml/ Coq)
BSearch	80	100%	6	131	1.4 / 11.6
HeapSort	143	100%	9	334	3.7 / 35.5
QuickSort	276	100%	9	462	128.7 / 974.0
Random	883	83%	8	390	8.0 / 44.3
Jacobi	135	50%	19	132	1.7 / 9.2
LU	559	45%	16	997	17.4 / 91.5
SparseCompRow	90	33%	15	72	1.1 / 6.1
FFT	591	78%	30	645	22.7 / 193.8

Figure 8: Size in number of instructions, score in ratio succeeded checks / total checks, certificates in bytes, checking time in milliseconds

The checking time is very small (less than one second), which is especially noteworthy given that the checker is run in Coq. We clearly benefit here from our efficient implementation and the optimised reduction engine of Coq [19]. Compared to the extracted version, the Coq checker is at most 10 times slower.

10 Towards a Certified Lightweight Array Bound Checker for Java Bytecode

The work we have reported in the previous sections demonstrates the feasibility of efficiently checking, in a foundational way, the result of a relational static analyser. Our aim is now to scale this approach on a more realistic fragment of Java (mainly its full sequential part) for a competitive array bound checker.

To do so, we have designed a new static analyser of Java bytecode programs with several new features. The prototype is written in OCaml. In a second time we will develop a certified result checker in Coq. In this section we present the main characteristics of the analyser. All components are schematically presented in Figure 9.

10.1 Parsing of `.class` files

We rely on the Javalib Ocaml library[2] that gives us a factorized representation of bytecode instructions with full inlining of constant pool indirections.

10.2 Removing operand stack manipulation

The JVM is a stack-based virtual machine. This intensive use of the operand stack make it difficult to adapt standard static analysis techniques that have been first designed for more standard

[2] `http://javalib.gforge.inria.fr/`

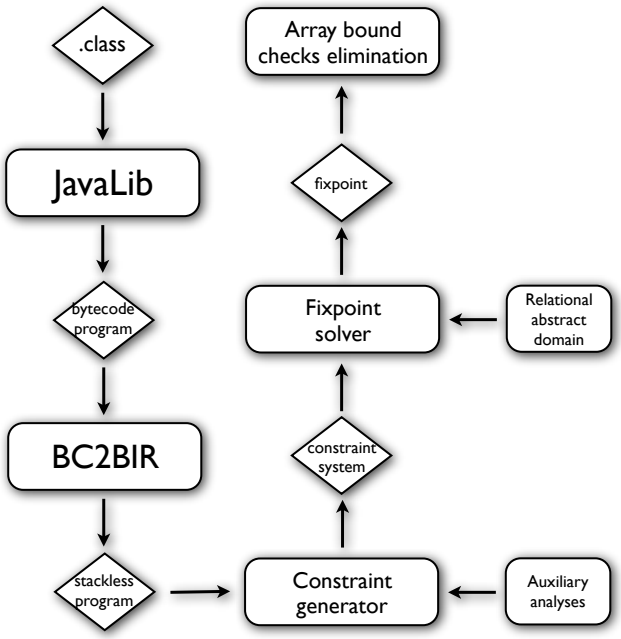


Figure 9: Architecture of the array bound analyser

(variable-based) 3-address codes. A naive translation from a stack-based code to 3-address code may result in an explosion of temporary variables, which in turn may complicate analyses of relational program analyses.

In Section 5.1 we rely on a stack of symbolic expressions to represents relation between operands, static fields and local variables. We adapt here this idea to transform a bytecode program into a stack-less representation. Instead of doing this transformation during the analysis, we perform it once for all before the analysis, instead of re-transforming at each iteration of the analysis.

The transformation algorithm is described and proved correct in a separate research report [17]. We give just give here an example of its result. Figure 10 presents a simple source program, its bytecode and its stackless representation and finally the symbolic operand stack that is computed during transformation. In this example the variable `x` is denoted by number `1` in the bytecode representation and `r1` its counterpart in the stackless representation. The `iload 1` instruction generates a `nop` instruction but pushes the symbol `r1` on top of the symbolic operand stack. The `ifne 8` instruction uses the symbolic stack to recover the original guard (`x==0`) of the program. It generates a conditional jump to line `8` and pops the first (and only) element of the symbolic operand stack. The next instruction `iconst 1` generates a `nop` instruction and pushes the symbol `1` on top of the symbolic operand stack. The effect of the next instruction `goto 9` is more subtle: since the target of the jump is a branching point, the transformation takes care to generate the same symbolic stack from both predecessors of line `9`. To do so, it generates a fresh variable `b9` and generates the necessary assignment before the jump. The next instruction `iconst -1` takes the same precaution. Finally the instruction `ireturn` pop the top of the symbolic stack to return the corresponding expression.

Thanks to this transformation our static analysis just need to reason on a simple language with expression trees. Note that we do not expect the result verifier to rely on the same preliminary transformation: we plan to keep a symbolic manipulation similar to Section 5.1 during fixpoint checking. This is possible because the transformation algorithm operates mainly in one pass on each methods.

```
int f(int x) {return (x==0) ? 1 : -1;}
```

(a) source program

```
int f(int);
 0: iload  1
 1: ifne   8
 4: iconst 1
 5: goto   9

 8: iconst -1
 9: ireturn
```

(b) bytecode program

```
int f(int);
 0: nop;
 1: if r1 != 0 goto 8;
 4: nop;
 5: b9 := 1;
    goto 9;
 8: b9 := -1;
 9: return b9;
```

(c) stackless program

```
0: []
1: [r1]
4: []
5: [1]

8: []
9: [b9]
```

(d) symbolic stack

Figure 10: Example of bytecode transformation

10.3 Constraint generation

For each method of a program we generate a set of numerical symbolic constraints. Figure 11 presents a Java method (binary search), its stackless representation and the constraint system that is generated for this method. On the left part of the constraint system, we note the corresponding line number where the constraint has been generated. The system constrains three kind of variables: *Pre*, *Post* and $Loc(i)$ where i is a line number. Words in italic mode correspond to reserve words. The first constraint binds the values of the variables `key` and `vec` with the formal parameters of the method. The constraint generator only keeps expressions that can be expressed in a simple numeric language with variable, constants and numeric operations. Hence, at line 12, it keeps the expression $((\texttt{high} - \texttt{low})/2) + \texttt{low}$ but forgets $(\texttt{key} \neq \texttt{vec}[\texttt{mid}])$ at line 32 because it contains an array expression. The special operator $[local :=?]$ projects all local variables (here `low`, `mid`, `high`, `key` and $\texttt{vec}.length$). It is used to constrain the post-condition *Post* which only deals with parameters and final result.

Auxiliary analyses are necessary to generate some of these constraints. It is for example necessary to recover array types in order to predict which expression will definitively handle rectangular arrays, with which dimensions. In this example, it allows us to predict that `vec` handles an array of one dimension.

10.4 Fixpoint solving

Each constraint system is given to a generic fixpoint solver [20] that over-approximates its fixpoints. More precisely, for any value of the precondition variable *Pre*, it computes a value for the postcondition *Post*. This technique allows us to obtain a context-sensitive analysis which has the same level of precision that a full inlining of methods. In case of recursive calls, we iter the fixpoint resolution between methods, using widenings to ensure convergence. The technique is taken and adapted from the last chapter of Patrick Cousot's PhD thesis [16].

The constraints are interpreted on top of any abstract domain of the Apron library [21], as octagons [25] or polyhedra [15]. Java arithmetic overflow is taken into account by systematically proving that no overflow/underflow occurs. As a consequence, in the binary search example of Figure 11, the expression $((\texttt{high} - \texttt{low})/2) + \texttt{low}$ is proved to be safe with respect to overflow/underflow, while $(\texttt{high} + \texttt{low})/2$ would have lead to a true alarm. When such a case occurs, the analysis over-approximates the value of the expression by $\top$.

```
static int bsearch(int key, int[] vec) {
  int low = 0;
  int high = vec.length - 1;
  while (high > low) {
    int mid = (high -low) / 2 + low;
    if (key == vec[mid]) return mid;
    else if (key < vec[mid])
      high = mid - 1;
    else low = mid + 1;
  }
  return -1;
}
```

(a) source program

```
static int bsearch(int key, int[] vec)
   0. low := 0
   2. high := vec.length-1
   7. if (high <= low) goto 56
  12. mid := ((high-low)/2)+low
  21. if (key != vec[mid]) goto 32
  29. return mid
  32. if (key >= vec[mid]) goto 48
  40. high := mid-1
  45. goto 53
  48. low := mid+1
  53. goto 7
  56. return -1
```

(b) stackless bytecode representation

$$
\begin{array}{rrcl}
 & [\texttt{key} = param0][\texttt{vec}.length = param1.length]Pre & \sqsubseteq & Loc(0) \\
0. & [\texttt{low} := 0]Loc(0) & \sqsubseteq & Loc(2) \\
2. & [\texttt{high} := \texttt{vec}.length - 1]Loc(2) & \sqsubseteq & Loc(7) \\
7. & [\texttt{high} \leq \texttt{low}]Loc(7) & \sqsubseteq & Loc(56) \\
 & [\texttt{high} > \texttt{low}]Loc(7) & \sqsubseteq & Loc(12) \\
12. & [\texttt{mid} := ((\texttt{high} - \texttt{low})/2) + \texttt{low}]Loc(12) & \sqsubseteq & Loc(21) \\
21. & Loc(21) & \sqsubseteq & Loc(32) \\
 & Loc(21) & \sqsubseteq & Loc(29) \\
29. & [local :=?][result := \texttt{mid}]Loc(29) & \sqsubseteq & Post \\
32. & Loc(32) & \sqsubseteq & Loc(48) \\
 & Loc(32) & \sqsubseteq & Loc(40) \\
40. & [\texttt{high} := \texttt{mid} - 1]Loc(40) & \sqsubseteq & Loc(45) \\
45. & Loc(45) & \sqsubseteq & Loc(53) \\
48. & [\texttt{low} := \texttt{mid} + 1]Loc(48) & \sqsubseteq & Loc(53) \\
53. & Loc(53) & \sqsubseteq & Loc(7) \\
56. & [local :=?][result := -1]Loc(56) & \sqsubseteq & Post
\end{array}
$$

(c) constraint system

Figure 11: Example of constraint generation

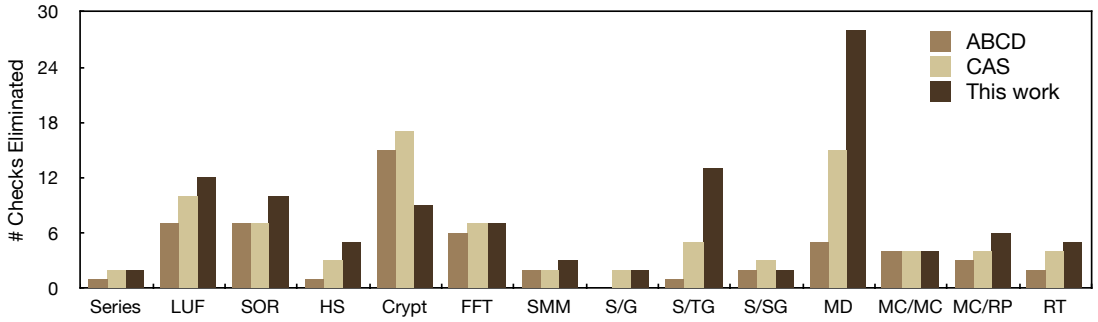


Figure 12: Evaluation of the precision of the intra-procedural part of the analyser on the Java Grande Forum benchmarks

10.5 Preliminary experiments

The goal of this implementation is to obtain an array bound checker that with a state-of-the-art precision and then design a certified result checker for it. In order to achieve the first goal we have run the analyser on the same benchmark suite, Java Grande Forum benchmarks [10], as Niedzielski et al. [28]. The benchmarks were modified to express to cope with their intra-procedural analysis. They have also run teh ABCD analyser [8] on the same programs. We have bridle our analysis in order to not propagate out of method calls and run our analysis on the same benchmark suite (with the same modifications, kindly provided by the authors of [28]). The result is shown in Figure. 12. For all benchs (except one) we obtain a similar or better precision. This results should however be interpreted with precaution because some of the checks eliminated in [28] may not have been counted here.

11 Related work

A number of relational abstract domains (octagons [25], convex polyhedra [15], polynomial equalities [27]) have been proposed with various trade-offs between precision and efficiency, and intra-procedural relational abstract interpretation for high-level imperative languages is by now a mature analysis technique. However, to the best of our knowledge the present work is the first extension of this to an inter-procedural analysis for byte code. Dependent type systems for Java-style byte code for removing array bounds checks have been proposed by Xi and Xia [37]. The analysis of the stack uses singleton types to track the values of stack elements, achieving the same as our symbolic stack expressions. The analysis is intra-procedural and does not consider methods (they are added in a later work [36] which also adds a richer set of types). The type checking relies on loop invariants. We have run our analysis on the example Xanadu programs given by Xi and have been able to infer the invariants necessary for verifying safe array access automatically.

The area of certified program verifiers has been an active field recently. Wildmoser, Nipkow *et al.* [33] were the first to develop a fully certified VCGen within Isabelle/HOL for verifying arithmetic overflow in Java byte code. The certification of abstract interpreters has been developed by Cachera, Pichardie *et al.* [11, 29]. for a variety of analyses including class analysis of Java byte code and interval analysis. Lee *et al.* [22] have certified the type analysis of a language close to Standard ML in LF and Leroy [23] has certified some of the data flow analyses of a compiler back-end. Leroy also observes that for certain, more involved analyses such as the register allocation, it is simpler and sufficient to certify a checker of the result than the analysis itself. The same idea is used by Wildmoser *et al.* [32] who certifies a VCGen that uses untrusted interval analysis for producing invariants and that relies on Isabelle/HOL decision procedures to check the verification conditions generated with the help of these invariants. Their technique for analysing byte code is close to ours in that they also use symbolic expressions to analyse the operand stack and the main

contribution of the work reported here with respect to theirs is to develop this result checking approach for a fully relational analysis.

The idea of removing useless parts from an invariant was developed independently by Besson *et al.* [7] and by Yang *et al.* [38] who call it abstract value slicing. Both works deal with intra-procedural invariants and both are based on a dependency computation that selects, for every constraint $F(X) \sqsubseteq Y$ of the constraint system of P and every subset of an abstract state Y, a sufficient subset of X that satisfies the constraint. The two methods differ in the way that this choice is done but both have been shown viable for intra-procedural pruning of relational invariants. The present work is an extension of the principles underlying the non-deterministic algorithm in [7] to handle the pre-/post-conditions arising from the interprocedural analysis. Finally, it should be noted that the fixpoint compression is orthogonal to and compatible with the optimisation of iteration strategies for fixpoint checking underlying Lightweight Bytecode Verification [30] and the more general abstraction-carrying code [1, 6]. Our checker combines both techniques.

12 Conclusions and future work

This paper demonstrates the feasibility of an interprocedural relational analysis which automatically infers polyhedral loop invariants and pre-/post-condition for programs in an imperative byte code language. The machine-generated invariants can be pruned wrt. a particular safety policy to yield compact program certificates. To simplify the checking of these certificates, we have devised a result checker for polyhedra which uses inclusion certificates (issued from a result due to Farkas) instead of computing convex hulls of polyhedra at join points. This checker is much simpler to prove correct mechanically than the polyhedral analyser and provides a means of building a foundational proof carrying code that can make use of industrial strength relational program analysis.

Future work concerns extensions to incorporate richer domains of properties such as disjunctive completion of polyhedra or non-linear (polynomial) invariants. The certificate format and the result checker can accommodate the disjunctive completions, the inclusion certificates from Section 8.2 can be generalised to deal with non-linear inequalities as well [5]. However, the analyses for *inferring* such properties are in their infancy. On a language level, the challenge is to extend the analysis to cover the object oriented aspects of Java byte code. The inclusion of static fields and arrays in our framework provides a first step in that direction but a full extension would notably require an additional analysis to keep track of aliases between objects.

A promising domain of application for our relational analysis technique is to verify the dynamic allocation and consumption of resources and in particular to ensure statically that a program always acquires a necessary amount of resources before consuming them. The approach of Chander *et al.* [12] relies on the programmer to provide loop invariants and pre- and post-conditions for methods in order to link program variables to the amount of resources available and perform powerful transformations such as hoisting resource allocations out of loops. Our inter-procedural byte code analyser could infer the necessary invariants and pre-/post-conditions and in the same vein provide the checker for integrating this into a mobile code resource certification scheme.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code. In *Proc. of the 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning*, Springer LNAI vol. 3452, pages 380–397, 2004.

[2] B. De Backer and H. Beringer. A clp language handling disjunctions of linear constraints. In *Proc. of the 10th Int. Conf. on Logic Programming (ICLP'93)*, pages 550–563. MIT Press, 1993.

[3] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. of 10th Int. Static Analysis Symposium*, pages 337–354. Springer LNCS vol. 2694, 2003.

[4] R. Bagnara, P.M Hill, and E. Zaffanella. The Parma polyhedral library user's manual, 2006.

[5] F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 48–62. Springer, 2006.

[6] F. Besson, T. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

[7] F. Besson, T. Jensen, and T. Turpin. Small witnesses for abstract interpretation based proofs. In *Proc. of 16th Europ. Symp. on Programming (ESOP 2007)*, pages 268–283. Springer LNCS vol. 4421, 2007.

[8] R. Bodík, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *Proc. of PLDI'00*, pages 321–333. ACM Press, 2000.

[9] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications*, pages 128–141. Springer LNCS vol. 735, 1993.

[10] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, and R.A. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[11] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *Proc. of 13th Europ. Symp. on Programming (ESOP'04)*, pages 385–400. Springer LNCS vol. 2986, 2004.

[12] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of the 14th European Symposium on Programming (ESOP 2005)*, pages 311–325. Springer LNCS vol. 3444, 2005.

[13] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *U.S.S.R Comp. Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of 5th ACM Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[16] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. PhD thesis, Thèse d'état ès sciences mathématiques, Université scientifique et médicale de Grenoble, France, 1978. In french.

[17] D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for java bytecode. Research Report xxxx, IRISA, 2009.

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[19] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proc. of the 7th ACM International Conference on Functional Programming (ICFP'02)*, pages 235–246. ACM Press, 2002.

[20] B. Jeannet. Fixpoint: generic fixpoint solving library, 2009.

[21] B. Jeannet and the Apron team. The Apron library, 2007.

[22] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ml. In *Proc. of 34th ACM Symp. on Principles of Programming Languages (POPL'07)*, pages 173–184. ACM Press, 2007.

[23] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of the 33rd ACM Symp. on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[24] A. Makhorin. Glpk (gnu linear programming kit) version 4.28, 2008.

[25] A. Miné. The octagon abstract domain. In *Proc. of Working Conf. on Reverse Engineering 2001*, IEEE, pages 310–319. IEEE Computer Society, October 2001.

[26] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI'06*, volume 3855 of *LNCS*, pages 348–363. Springer, 2002.

[27] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. of 31st ACM Symp. on Principles of Programming Languages (POPL'04)*, pages 330–341. ACM Press, 2004.

[28] D. Niedzielski, J. von Ronne, A. Gampe, and K. Psarris. A verifiable, control flow aware constraint analyzer for bounds check elimination. In *Proc. of the 16th International Static Analysis Symposium (SAS'09)*, pages 137–153. Springer LNCS vol. 5673, 2009.

[29] D. Pichardie. *Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université de Rennes 1, 2005.

[30] E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[31] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[32] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of 1st Workshop on Bytecode Semantics, Verification and Transformation, ENTCS*, 2005.

[33] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In *Proc. of the 15th European Symp. on Programming (ESOP'05)*, 2005.

[34] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Exploring New Frontiers of Theoretical Informatics, TC1 3rd Int. Conf. on Theoretical Computer Science (TCS2004)*, pages 333–347. Kluwer, 2004.

[35] Coq development of the certified checker. `http://www.irisa.fr/lande/polycert/`.

[36] H. Xi. Imperative Programming with Dependent Types. In *Proc. of 15th IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 375–387. IEEE, 2000.

[37] Hongwei Xi and Songtao Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proc. of CASCOON '99*, pages 110–125, 1999.

[38] H. Yang, S. Seo, K. Yi, and T. Han. Goal-directed weakening of abstract interpretation results. Submitted for publication.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Provably Correct Stackless Intermediate Representation for Java Bytecode

Delphine Demange — Thomas Jensen — David Pichardie

N° ????

Juillet 2009

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche RENNES - BRETAGNE ATLANTIQUE

A Provably Correct Stackless Intermediate Representation for Java Bytecode

Delphine Demange*, Thomas Jensen†, David Pichardie‡

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet Celtique

Rapport de recherche n° ???? — Juillet 2009 — 53 pages

Abstract: The Java virtual machine executes stack-based bytecode. The intensive use of an operand stack has been identified as a major obstacle for static analysis and it is now common for static analysis tools to manipulate a stackless intermediate representation (IR) of bytecode programs. Several algorithms have been proposed to achieve such a transformation, whereas only little attention has been paid to their formal semantic properties. This paper specifies such a bytecode transformation and provides the semantic foundations for proving that an initial bytecode program and its IR behave similarly, in particular with respect to object creation and throwing of exceptions. The transformation is based on a symbolic execution of the bytecode, using a symbolic operand stack. Each bytecode instruction modifies the abstract stack and gives rise to the generation of IR instructions. We formalize a notion of semantics preservation: an initial program and its IR form have similar execution traces but since the transformation does not preserve the order in which objects are allocated, the similarity between traces is defined using an equivalence relation over the two heaps. Finally, we prove the correctness of this transformation with respect to this semantic criterion.

Key-words: Program Analysis, Bytecode languages

Work partially supported by EU project MOBIUS

* Université de Rennes 1, Rennes, France
† CNRS, Rennes, France
‡ INRIA, Centre Rennes - Bretagne Atlantique, Rennes

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Une Représentation Intermédiaire Basée Registre Prouvée Correcte pour le Bytecode Java

Résumé : La machine virtuelle Java execute des programmes bytecodes en utilisant une pile d'opérande. Cet usage intensif d'une pile d'opérande a été identifié comme un obstacle majeur pour l'analyse statique. Il est désormais courant que les outils d'analyses statiques modernes aient recours a une transformation préliminaire qui retire cet usage. Plusieurs algorithmes ont été proposés pour réaliser cette transformation, mais très peu d'attention a été porté jusque là à leurs propriétés sémantiques. Ce travail spécifie une telle transformation et propose les foundation sémantiques pour prouver qu'un programme bytecode initial et sa représentation intermédiaire se comporte de façons similaires.

Mots-clés : Analyse statique, languages de bytecode

Chapter 1

Introduction

Static analysers of Java programs often work at bytecode level, for several reasons. The security of a Java virtual machine (JVM) is enforced at byte-code level, hence security-related analyses (notably the Java byte code verifier) operates at this level. Also, the Java byte code language is sometimes considered simpler than its source counterpart (no inner class, no generics), and hence simpler to analyse.

However four important features still complicates the static analysis of a bytecode program. First, the JVM is a stack-based virtual machine. This **intensive use of the operand stack** may make it difficult to adapt standard static analysis techniques that have been first designed for more standard (variable-based) 3-address codes. As noticed by Logozzo and Fähndrich [LF08], a naive translation from a stack-based code to 3-address code may result in an explosion of temporary variables, which in turn may complicate analyses of relational program analyses. Second, the Java execution model intensively relies on dynamic check to ensure part of its intrinsic properties like absence of null-pointer dereferences, out-of-bounds array accesses, etc... The consequence is that many instructions of the Java bytecode language may rise different kind of **exceptions**. Any sound static analysis must take this mechanism into account without permuting the throwing of exceptions that could arise during the execution of a program. The Java bytecode verifier not only enforces type safety of bytecode programs but also a complex **object initialisation** property: an object can't be used before an adequate constructor has been called and terminated correctly. References to uninitialised objects are frequently duplicated on the operand stack, which make difficult for an analysis to recover the sequence of actions i) allocation of raw object, ii) call to a constructor, iii) store reference to initialised object in a local variable. The bytecode verifier has this ability (at least for the two first actions), tracking alias of uninitialised object in the operand stack, but this information is lost for the other static analyses. Finally, Java bytecode is unstructured: a bytecode program is a list of instructions, the control flow is affected by conditional and unconditional jumps.

These complications are well-known to the Java analysis community. As a consequence, several Java bytecode optimization and analysis tools work on an **intermediate representations** (IR) of bytecode that makes analysis simpler. This is for example the case with Soot [VRCG+99] and its IR (Baf, Jimple and Grimp) and with the Jalapeno Dynamic Compiler developed at IBM [BCF+99]. Using such transformations may simplify the work of the analyser but its overall correctness now becomes dependent on the semantics-preserving properties of the transformation. Surprisingly, the semantic foundations of these byte code transformations has received less attention. This paper provides a semantically sound, provably correct specification of a transformation of byte code into an intermediate representations (IR) of bytecode that i) removes the use of the operand stack and re-

builds tree expressions, ii) makes more explicit the throwing of exception and takes care of preserving their order, iii) rebuild the initialisation chain of an object with a dedicated instruction `x := new C(arg1,arg2,...)`.

The Figure 1.1 presents an example of Java program (Figure 1.1(a)) which raises some transformation challenges. This example illustrates the fact that transforming bytecode in order to reconstruct side-effect-free expressions and folded object initialisations raises several challenges in the algorithm, the observational equivalence and the correctness proof. Its corresponding bytecode version (Figure 1.1(c)) illustrates the standard object initialisation scheme. An expression like `new A()` is traditionally compiled into the sequence of lines `[4;5;6]`. A new object of class `A` is first allocated in the heap and stored at an address that is pushed on top of the operand stack. The address is then duplicated on the stack by the instruction `dup` and the non-virtual method `A()` is called, consuming the top of the stack. The previous copy is now on the top of the stack and represents from now an *initialized* object. This initialization by side-effect, is particularly challenging for the bytecode verifier [FM99] which has to keep track of the alias between not-yet-initialized references on the stack. Using a similar approach we are able to stick again allocation and instance initializer calls. Figure 1.1(b) shows the result of such a folding. In this example we take care to not use expression with side-effects. It is nevertheless wrong for multiple reasons. First, it does not respect the allocation order. This fact is unavoidable if we want to keep side-effect free expression and still fold object constructions. It is not so a serious problem because before the call to its constructor, an uninitialised reference is unusable. This however complexify the correctness proof. In Bytecode Java programs, the allocation order may nevertheless have a functional impact because of the static initializer `A.<clinit>` that may be called when reaching an instruction `new A`. In Figure 1.1(b) this order is not preserved since `A.<clinit>` may be call before `B.<clinit>` while the bytecode program follows an inverse order. The program in Figure 1.1(d) solves this problem using a specific instruction `mayinit(A)` that make explicit the potential call to a static initializer. The second major semantic problem of Figure 1.1(b) is that the program does not respect the exception launching order of the bytecode version. In Figure 1.1(b) the method call to `A()` may appear before the `NullPointer` exception that may be raised during the evaluation of `x.f` if the value of `x` is null. The program in Figure 1.1(d) solves this problem using a specific instruction `notnull(x)` that explicitly checks that `x` is non-null and raises a `NullPointer` exception otherwise.

```
B f(X x) {
  return new B(x.f,new A());
}
```

(a) source function

```
B f(x);
  0: t1 = new A();
  1: t2 = new B(x.f,t1);
  2: vreturn t2;
```

(b) BIR function (not semantic preserving)

```
B f(x);
  0: new B
  1: dup
  2: aload x
  3: getfield f
  4: new A
  5: dup
  6: constructor A
  7: constructor B
  8: vreturn
```

(c) BC function

```
B f(x);
  0: mayinit(B);
  1: nop;
  2: nop;
  3: notnull(x);
  4: mayinit(A);
  5: nop;
  6: t1 := new A();
  7: t2 := new B(x.f,t1);
  8: vreturn t2
```

(d) BIR function (semantic preserving)

Figure 1.1: Motivating example

Chapter 2

Related work

Many Java bytecode optimization and analysis tools work on an intermediate representations (IR) of bytecode that make its analysis much simpler. Soot [VRCG+99] is a Java bytecode optimization framework originally providing three IR: Baf, Jimple and Grimp. Optimizing Java bytecode consists in successively translating bytecode into Baf, Jimple, and Grimp, and then back to bytecode, while performing diverse optimizations on each IR. Baf is a stack-based code that abstracts the *constant pool*, a table in each class containing the value of all constants (numerical constants, references to strings, class methods...), that is tedious to handle in static analyses. Each Baf instruction is fully typed. Jimple is a typed stackless 3-address code. Expressions are made more explicit (operands cannot be as separated as they could be in the stack). Jimple instructions are typed. Grimp is a stackless code with tree expressions. It is obtained by simple collapsing of 3-address Jimple instruction. Costa [AAG+07] is a static analyser that infer cost information on Java bytecode programs. It relies on an IR similar to Jimple by removing explicit uses of the operand stack using additional local variables. Contrary to Soot, it does not address the problem of generating types instructions. Type inference of Jimple instructions has been adressed in [GHM00]. The transformation algorithm studied in this paper follows instead The symbolic evaluation technique used by Whaley [Wha99] for the High intermediate representation of the Jalapeño Optimizing Compiler [BCF+99]. This paper pushes the technique further, generating boolean tree expressions in conditionnal branchings and folding constructor calls with allocation statements. All these previous works have been mainly concerned with the construction of effective and powerfull tools but, as far as we know, little attention has been paid to the formal semantic properties that are ensured by these transformations.

The use of a symbolic evalatution of the operand stack to recover some tree expressions in a bytecode program has been employed in several context of Java Bytecode analysis. The technique was already used in one of the first Sun Just-In-Time compiler [CFM+97] for direct translation of bytecode to machine instruction. Xi and Xia propose a dependent type system for array bound check elimination that uses symbolic expressions to type operand stacks with *singleton* types in order to recover relations between length of arrays and index expression that are used to access them. Besson *et al* [BJP06], and independently Wildmoser *et al* [WCN05], propose an extended interval analysis that verifies there is no out-of-bound array accesses in certified programs using symbolic decompilation. Besson *et al* give an example that shows how the precision of the standard interval analysis is enhanced by including syntactic expressions in the abstract domain.

Chapter 3

The source language: BC

In our work, we consider a restricted version of the Java Bytecode language. Difficulties that are inherent to the object oriented paradigm (e.g. object initialization) have to be addressed before considering e.g. multi-threading. In this chapter, we present the formalization of the subset of Java Bytecode language we consider: BC. We first describe its syntax in Section 3.1. Then, we propose an observational, operational semantics for BC.

3.1 Syntax of BC

The set of bytecodes we consider is given in Figure 3.1. We describe each of them briefly and justify our choices before further formalizing their semantics in Section 3.2.

var	::=		*variables* :
		$\mathtt{x} \mid \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots$	
		`this`	
oper	::=		*operands* :
		$\mathtt{c} \mid \mathtt{c}' \ldots \mid \mathtt{null}$	*constant*
		var	*variable*
		$\mathtt{pc} \mid \mathtt{pc}' \ldots$	*program counter*
		$\mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \ldots$	*class name*
		$\mathtt{f} \mid \mathtt{f}' \mid \ldots$	*field name*
		$\mathtt{m} \mid \mathtt{m}' \mid \ldots$	*method name*

instr	::=	*instructions* :
		`nop` | `push c` | `pop` | `dup` | `add` | `div`
		| `load x` | `store x`
		| `new C` | `constructor C`
		| `getfield f` | `putfield f`
		| `invokevirtual C.m`
		| `if pc` | `goto pc`
		| `vreturn` | `return`

Figure 3.1: Operands and instructions of BC

BC provides simple stack operations: `push c` pushes the constant `c` (who might be `null`) onto the stack. `pop` pops the top element off the stack. `dup` duplicates the top element of the stack. All operands are the same size: we hence avoid the problem of typing `dup` and `pop`. The `swap` bytecode is not considered as it would be treated similarly to `dup` or `pop`. Only two binary arithmetic operators over values is available, the addition `add` and the division `div`: other operators (i.e. substraction, multiplication) would be handled similarely. We also restrict branching instructions to only one, `if pc`: the control flow jumps to the label `pc` if the top element of the stack is zero. Others jumps (e.g.

mcode	::=		*method code* :
		`pc : return`	*method end*
		`pc : vreturn`	*method end*
		`1 :` *instr mcode*	*instr list*
r	::=		*return type* :
		`void | v`	*void/non void*
msig	::=		*signature* :
		`C` *r* `m` : *var* ... *var* : *var* ... *var*	
method	::=		BC *method* :
		msig { *mcode* }	

class	::=		BC *class* :
		`C` {	
		$\mathtt{f}_1 \ldots \mathtt{f}_n$	
		$method_1$	
		...	
		$method_m$ }	
prog	::=		BC *program* :
		$class_1$ ...	
		$class_p$	

Figure 3.2: BC syntax: methods, classes, programs

`ifle pc`) would be treated similarely. For the same reason, we choose not to include switch tables in BC.

The value of a local variable `x` can be pushed onto the stack with the instruction `load x`. The special variable `this` denotes the current object (within a method). We do not distinguish between loading an integer or reference variable, as it is in the real bytecode language: we suppose the bytecode passes the BCV, hence variables are correctly used. The same applies to bytecode `store x` that stores the value of the stack top element in the local variable `x`.

A new object of class `C` is allocated in the heap by the instruction `new C`. Then, it has to be initialized by calling its constructor `constructor B`, where `B` is either the declared class of the object (`C`) or one of its super class. We will see in Section 5.1 that both cases have to be distinguished during the transformation. The constructor of a class is supposed to be unique. In real Java bytecode, `constructor B` corresponds to `invokespecial B.<init>`, but instruction `invokespecial` is used for many other cases. Our dedicated bytecode focus on its role for object constructors. We do not consider static fields or static methods. Class fields are read and assigned with `getfield f` and `putfield f` (we suppose the resolution of field class has been done). A method `m` is called on an object with `invokevirtual C.m`. Finally, methods can either return a value (`vreturn`) or not (`return`). Real bytecode provides one instruction per return value type, but for the reason given above, BC does not. For sake of simplicity, we do not use any constant pool. Hence constants, variables, classes, fields and method identifiers will be denoted by strings – their potential identifier at Java source level (every identifier is unique).

Figure 3.2 gives the syntax of BC. A BC method is made of its signature (class name, method name, value or void return, formal paramaters and local variables) together with its code, a list of BC instructions, indexed by a program counter `pc` starting from 1. In the following, $instrAt_P(\mathtt{m}, \mathtt{pc})$ denotes the instruction at `pc` in the method `m` of the program P. A BC class is made of its name, its fields names and its methods. Finally, a BC program is a set of BC classes. In the next section, we present the operational, observational semantics we defined for BC. Here is an example of BC program. The method `main` computes

```
Test {
  f1 f2
  Test v main: x y : z {
     1:   load x
     2:   load y
     3:   add
     4:   store z
     5:   push 1
     6:   load z
     7:   add
     8:   vreturn
  }
}
```

the sum of its two arguments, stores the result in the local variable `z`, and returns the sum of 1 and `z`.

3.2 Semantics of BC

3.2.1 Semantic domains

Semantic domains are given in Figure 3.3. A value is either an integer, a reference or the special value *Null*.

$$\begin{array}{rcl} \mathit{Value} & = & (\mathit{Num}\ n),\ n \in \mathbb{Z} \\ & & (\mathit{Ref}\ r),\ r \in \mathit{Reference} \\ & & \mathit{Null} \\ \mathit{Stack} & = & \mathit{Value}^* \\ \mathit{Env} & = & \mathit{var} \hookrightarrow \mathit{Value} \\ \mathit{InitTag} & = & \mathit{ClassName} \cup \widetilde{\mathit{ClassName}} \cup \widetilde{\mathit{ClassName}}_{\mathit{PCount}} \\ \mathit{Object} & = & (\mathit{FieldName} \rightarrow \mathit{Value})_{\mathit{InitTag}} \\ \mathit{Heap} & = & \mathit{Reference} \hookrightarrow \mathit{Object} \\ \mathit{Frame} & = & \mathit{MethodName} \times \mathit{PCount} \times \mathit{Env} \times \mathit{Stack} \\ \mathit{State} & = & (\mathit{Heap} \times \mathit{Frame}) \cup (\mathit{Heap} \times (\mathit{Value} \cup \{\mathit{Void}\})) \cup \Omega^{\mathit{NP}}_{\mathit{PCount}} \cup \Omega^{\mathit{DZ}}_{\mathit{PCount}} \end{array}$$

Figure 3.3: The BC's semantic domains

The operand stack is a list of elements of *Value*. Given the set of variable identifiers *var*, that includes the special identifier *this* (denoting the current object), an environment is a partial function from *var* to *Value*. To lighten the notations, we assume in the following that when the variable `x` is accessed in an environment l, written l(`x`), `x` is as required in the domain of l.

An object is represented as a total function from its fields names to values. An initialization status is also attached to every object. Initialization tags were first introduced by Freund and Mitchell [FM03] in order to formalize the object initialization verification pass performed by the Bytecode Verifier. They provide a type system ensuring, amoung other things, that (i) every newly allocated object is initialized before being used and (ii) every constructor, from the declared class of the object up to the `Object` class is called before considering the object as initialized. We further explain initialization tags in a dedicated paragraph below.

The heap is a partial function from non-null references to objects. The special reference *Null* does not point to any object. Each time a new object is allocated in the heap, the partial function is extended accordingly. Object allocation is a deterministic function, given the current state of the heap and the size of the new object to allocate in it. We do not model any garbage collector and the heap is considered arbitrarily large, or at least sufficiently large for the program execution not to raise `outOfMemory` errors.

Initialization tags Object initialization is a key point in our work. So let us describe what it means for an object to be initialized. In Figure 3.4 is given the life cycle of an object, from its creation to its use. Suppose the direct super class of `C` is class `B` and the direct super class of `B` is `Object` (see class hierarchy on the right). A new object of class `C` is allocated in the heap with `new C` at program counter `pc` in the main method. It is allocated in the heap but yet uninitialized. No operation is allowed on this object (no method call on it, nor field access or modification). At some point, the constructor of

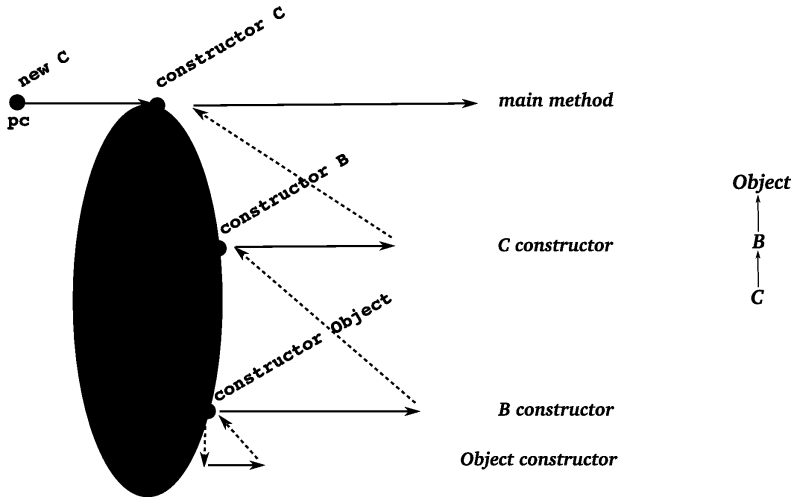


Figure 3.4: Object initialization process: example

class C is invoked. The initialization process has begun (the object is *being initialized*), and in the JVM specification, from this point, its fields can be written. To simplify, we consider that no field modification is allowed yet (this simplification is also done in [FM03]). The only operation that is allowed on it is to call a super constructor of this object (in the JVM, another constructor of class C can be invoked, but in our work constructors are supposed to be unique). Every super constructor in the class hierarchy has to be called, up to the constructor of the `Object` class. As soon as the `Object` constructor is called, the object is considered as initialized: methods can be called on it, and its fields can be accessed or modified. In Figure 3.4, the grey area denotes the part of the execution where the created object is considered as being initialized. Before entering this area, the object is said to be uninitialized. After escaping this aera, the object is considered as initialized.

In our work, we suppose that the bytecode passes the Bytecode Verifier. Hence, we know that the constructors chain is correctly called and that each object is initialized before being used. We use the initialization tags introduced in [FM03] to a different end: the semantics preservation needs to distinguish objects that are uninitialized from the other (see Section 5.2). We hence use a similar but simplified version of Freund and Mitchell's initialization tags:

- An object that is just allocated (no constructor has been called yet) has initialization tag $\widetilde{\mathsf{C}}_{\mathtt{pc}}$, meaning that the object was allocated at line `pc` by the instruction `new C`. Keeping track of `pc` strictly identifies uninitialized objects (thanks to the BCV pass, we are ensured that no uninitialized reference is in the stack at backward branchings) and is used to ensure the correctness of the substitutions on abstract stacks during the transformation (see Section 5.1 for further details)
- If the object is being initialized, its tag is $\widetilde{\mathsf{C}}$. The declared class of the object is C, and the current constructor is of class C or above ([FM03] keeps track of it, but we do not need to)
- As soon as the constructor of the `Object` class is called on a yet unitialized object of class C, the initialization tag is updated to C and the object is considered as initialized

We follow the work of [FM03] for the definition of function *Blank*(*t*), where $t \in \mathit{InitTag}$. This function creates a new object of the class indicated in the tag and whose fields are set to the default values (i.e. to zero for integers and *Null* for references).

Execution states In the BC semantics, there are two kinds of execution states: normal execution states and error states. In the first case, the state is made of the current heap and, either an execution frame (the method has not returned yet) or a returned value (possibly *Void*). An execution frame is made of the current method $m \in MethodName$, environnement $l \in Env$, operand stack $s \in Stack$ and the next instruction to execute is at label $pc \in PCount$. Note that we do not define the semantics of BC with the traditional call stack.

BC semantics also includes error states where the execution flows whenever a division by zero (Ω^{DZ}_{pc}) has occured or a null pointer has been dereferenced (Ω^{NP}_{pc}) at program point `pc`. We denote by Ω^{k}_{pc} the error state whose kind k is either DZ or NP. The reason why error states are introduced is twofold. First, it makes the semantics able to know when the execution gets stuck because of a violation of syntactic or structural constraints verified by the BCV or because of an execution error. Second, it makes it possible to state the semantics preservation not only for normal executions. Error states are parametrized by the kind of error and the program point of the faulty instruction: whereas exceptions are not included in BC, this mechanism can be considered as equivalent. In Section 5.2, we state that both the kind of error and the faulty program point are preserved.

To lighten the notations, in the rest of this report, *Value* is ranged over by v when clear from the context. In the next section, we present formally the semantics we give to BC.

3.2.2 Semantics

We define the semantics of BC with a view to capturing as much of the program behaviour as possible, so that the semantics preservation property can fits the need of most of static analyses. We hence formalize the BC semantics in terms of a labelled transition system: labels keep track amoung other things of memory modifications, method calls and returns. Hence, the program behaviour is defined in a more observationnal way, and more information than the classical input/output relation is made available.

Basic semantics Before going into more details about labels and all the kinds of transitions we need, let us first present the rules given in Figures 3.5, 3.6 and 3.7 without considering neither transition labels, nor transitions indices. Transitions relate states in *State* as defined in Figure 3.3. We first describe normal execution rules (Figures 3.5 and 3.6). The general form of a semantic rule is

$$\frac{\begin{array}{c} instrAt_P(\mathtt{m}, \mathtt{pc}) = instr \\ \text{other conditions} \end{array}}{\langle s \rangle \rightarrow \langle s' \rangle}$$

where s and s' are execution states, related through the transition relation, which is defined by case analysis on the instruction at the current program point. When needed, other conditions that have to be satisfied (conditions about the content of the stack, of the heap...) are specified below the instruction.

Rules for `nop`, `push c`, `pop`, `dup`, `add`, `div` and `load x` are rather simple, and we do not make further comments about them. In rule for `store x`, we allow to explicitly store a reference in a local register only if it points to a initialized object. We need this restriction in the transformation algorithm: no valid BIR instruction sequence would match. Once again this restriction appears to be relatively minor: even if it would fit the JVM specification, such a bytecode sequence seems to be rarely used in practice: no such case were found in the analysis we made on the Soot distribution and the required libraries.

$$\frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{nop}}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle} \qquad \frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{push}\ c \\ v = (\mathit{Num}\ c) \Leftrightarrow c \neq \mathtt{null} \\ v = \mathit{Null} \Leftrightarrow c = \mathtt{null}\end{array}}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v :: s\rangle}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{pop}}{\langle h, m, pc, l, v :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle} \qquad \frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{dup}}{\langle h, m, pc, l, v :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v :: v :: s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{add} \\ v_1 = (\mathit{Num}\ n_1) \quad v_2 = (\mathit{Num}\ n_2) \\ v' = (\mathit{Num}\ (n_1 + n_2))\end{array}}{\langle h, m, pc, l, v_1 :: v_2 :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v' :: s\rangle} \qquad \frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{div} \\ v_1 = (\mathit{Num}\ n_1) \quad v_2 = (\mathit{Num}\ n_2) \\ n_2 \neq 0 \quad v' = (\mathit{Num}\ (n_1/n_2))\end{array}}{\langle h, m, pc, l, v_1 :: v_2 :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v' :: s\rangle}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{load}\ x}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, l(x) :: s\rangle} \qquad \frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{store}\ x \\ v = (\mathit{Ref}\ r) \Rightarrow h(r) = o_C\end{array}}{\langle h, m, pc, l, v :: s\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, pc+1, l[x \mapsto v], s\rangle}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{if}\ pc'}{\langle h, m, pc, l, (\mathit{Num}\ 0) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle} \qquad \frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{if}\ pc' \qquad n \neq 0}{\langle h, m, pc, l, (\mathit{Num}\ n) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{goto}\ pc'}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{vreturn}}{\langle h, m, pc, l, v :: s\rangle \xrightarrow{[\mathit{return}(v)]}_0 \langle h, v\rangle} \qquad \frac{\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{return}}{\langle h, m, pc, l, s\rangle \xrightarrow{[\mathit{return}(\mathit{Void})]}_0 \langle h, \mathit{Void}\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{new}\ C \\ (h', (\mathit{Ref}\ r)) = \mathit{newObject}(C, h) \qquad h' = h[r \mapsto \mathit{Blank}(\widetilde{\mathsf{C}_{\mathtt{pc}}})]\end{array}}{\langle h, m, pc, l, s\rangle \xrightarrow{[\mathit{mayinit}(\mathsf{C})]}_0 \langle h', m, pc+1, l, (\mathit{Ref}\ r) :: s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{putfield}\ f \\ h(r) = o_C \quad o' = o[f \mapsto v]\end{array}}{\langle h, m, pc, l, v :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r.f \leftarrow v]}_0 \langle h[r \mapsto o'], m, pc+1, l, s\rangle} \qquad \frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{getfield}\ f \\ h(r) = o_C\end{array}}{\langle h, m, pc, l, (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, o(f) :: s\rangle}$$

Figure 3.5: BC transition system

$$\forall \mathtt{m} \in \mathit{MethodName}, \mathit{InitLocalState}(\mathtt{m}) = \mathtt{m}, 1, [\mathit{this} \mapsto (\mathit{Ref}\ r), \mathtt{x_1} \mapsto v_1 \ldots \mathtt{x_n} \mapsto v_n], \varepsilon$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{invokevirtual\ C.m'} \\ h(r) = o_{C'} \qquad rv \neq \mathit{Void} \quad V = v_1 :: \ldots :: v_n \\ \mathit{Lookup}(m', \mathtt{C'}) = \mathtt{mc} \quad \langle h, \mathit{InitLocalState}(\mathtt{mc})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', rv\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r.C.mc(V)].\vec{\lambda}_h}_{n+1} \langle h', m, pc+1, l, rv :: s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{invokevirtual\ C.m'} \\ h(r) = o'_C \quad V = v_1 :: \ldots :: v_n \\ \mathit{Lookup}(m', \mathtt{C'}) = \mathtt{mc} \quad \langle h, \mathit{InitLocalState}(\mathtt{mc})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r.C.mc(V)].\vec{\lambda}_h}_{n+1} \langle h', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{constructor}\ C \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}}_{\mathtt{j}} \quad C \neq \mathtt{Object} \\ h' = h[r \mapsto \mathit{upInit}(o, \widetilde{\mathtt{C}})] \quad V = v_1 :: \ldots :: v_n \\ \langle h', \mathit{InitLocalState}(\mathtt{C.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{[r \leftarrow C.init(V)].\vec{\lambda}_h}_{n+1} \langle h'', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{constructor\ Object} \\ h(r) = o_t \quad t = \widetilde{\mathtt{Object}}_{\mathtt{j}} \\ h' = h[r \mapsto \mathit{upInit}(o, \mathtt{Object})] \quad V = v_1 :: \ldots :: v_n \\ \langle h', \mathit{InitLocalState}(\mathtt{Object.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{[r \leftarrow \mathtt{Object}.init(V)].\vec{\lambda}_h}_{n+1} \langle h'', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{constructor}\ C' \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}} \quad C \subset C' \neq \mathtt{Object} \\ V = v_1 :: \ldots :: v_n \quad \langle h, \mathit{InitLocalState}(\mathtt{C'.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r \leftarrow C'.init(V)].\vec{\lambda}_h}_{n+1} \langle h', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c}\mathit{instrAt}_P(\mathtt{m}, \mathtt{pc}) = \mathtt{constructor\ Object} \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}} \quad h' = h[r \mapsto \mathit{upInit}(o, \mathtt{C})] \\ V = v_1 :: \ldots :: v_n \\ \langle h', \mathit{InitLocalState}(\mathtt{Object.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle\end{array}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r \leftarrow \mathtt{Object}.init(V)].\vec{\lambda}_h}_{n+1} \langle h'', m, pc+1, l, s\rangle}$$

Figure 3.6: BC transition system : object initialization and method calls

In rule for the bytecode `new C`, $newObject(C, h)$ allocates a new object of class C in the heap h, and returns the new heap h' together with the new reference (*Ref r*). All fields of the object are set to their default value by the function *Blank*() and its initialization tag is $\widetilde{\mathtt{C}}_{\mathtt{pc}}$. As already said, this tag will be updated when calling a constructor of class C (or above) on (*Ref r*). One can access an object field with `getfield`, or modify it using `putfield`: the object must be initialized. Modifying an object field does not change its initialization status.

Dynamic methods can only be called on initialized objects (rule for `invokevirtual` in Figure 3.6). The method resolution returns the right method to execute (function *Lookup(m,C')*). The current object (pointed to by the reference (*Ref r*)) is passed to the called method `m'` as an argument using the special local variable `this`. Other arguments are passed to the method in variables $\mathtt{x}_1$ to $\mathtt{x}_\mathtt{n}$, assuming these are the variables identifiers found in the signature of `m'`.

Let us now describe the semantic rules of constructor calls (Figure 3.6). The first two rules are used when calling the first constructor on a object: the object is not initialized yet . The constructor is called with the reference to the object in its `this` register. At the beginning of the constructor, the object initialization status is updated (using function $upInit(o, \mathtt{t})$ that changes the initialization tag of the object o for t). It changes for $\widetilde{\mathtt{C}}$ if the declared class of the object is not `Object`, and the object is considered as initialized otherwise (second rule). The last two rules of Figure 3.6 are used when calling a constructor on an object whose initialization is ongoing: the initialization tag of the object is $\widetilde{\mathtt{C}}$. The object is initialized only at the beginning of the constructor of class `Object`.

Let us now describe how execution error are handled (Figure 3.7). The instruction `div` might cause a division by zero if the second top element of the stack is (*Num 0*). In this case, the execution goes into the error state Ω^{DZ}_{pc}, meaning that the division by zero (*DZ*) arose at point `pc`. Similarely, reading or writing a field might dereference a null pointer (the kind of error is here *NP*). Finally, concerning method and constructor calls, there are two cases: either the error is raised by the call itself (leading to Ω^{NP}_{pc}), or the error arises during the execution of the callee, at a given program point `pc'` (other side conditions are equal to the normal case). In this case, the error state is propagated to the caller: it ends in the error state of the same kind (*NP* or *DZ*) but parametrised by program point `pc`, that is the program point of the faulty instruction, from the point of view of the caller. This mechanism is very similar to Java Bytecode exception handlers.

Up to now, we described a rather classical bytecode semantics. We use the notion of initialization status introduced by [FM03], that has been simplified: we know the object initialization is correct, because we assume our bytecode passes the BCV. We only have to keep track of three initialization status, parametrised by the declared class of the object (uninitialized, being initialized or initialized). Now, we go into further details, describing the particularities of our semantics.

Observational semantics As can be seen in semantic rules, transitions are labelled. Labels are intented to keep track of the most of information about the program behaviour (e.g. memory effects or variable modifications...). Every (relevant) preserved elementary action should be made observable. Program behaviour aspects that are not preserved by the transformation are defined in terms of silent transitions, written $\langle s \rangle \xrightarrow{\tau} \langle s' \rangle$ (see e.g rules for `nop`, `dup` or `load x`). From now on, we use λ to denote either the silent event τ or any observable event. Observable events fall into one of these three categories:

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{div}}{\langle h, m, pc, l, (\mathit{Num}\ n) :: (\mathit{Num}\ 0) :: s\rangle \xrightarrow{\tau}_0 \Omega^{DZ}_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{putfield}\ f}{\langle h, m, pc, l, v :: \mathit{Null} :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}} \qquad \frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{getfield}\ f}{\langle h, m, pc, l, \mathit{Null} :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{invokevirtual}\ \mathtt{C.m'} \quad V = v_1 :: \ldots :: v_n}{\langle h, m, pc, l, V :: \mathit{Null} :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{invokevirtual}\ \mathtt{C.m'} \quad h(r) = o_{C'} \quad \mathit{Lookup}(m', C') = \mathtt{mc} \quad V = v_1 :: \ldots :: v_n \quad \langle h, \mathit{InitLocalState}(\mathtt{mc})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r.C.mc(V)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{constructor}\ C \quad V = v_1 :: \ldots :: v_n}{\langle h, m, pc, l, V :: \mathit{Null} :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{constructor}\ C \quad h(r) = o_t \quad t = \widetilde{\mathtt{C}}_{\mathtt{j}} \quad C \neq \mathtt{Object} \quad h' = h[r \mapsto \mathit{upInit}(o, \widetilde{\mathtt{C}})] \quad V = v_1 :: \ldots :: v_n \quad \langle h', \mathit{InitLocalState}(\mathtt{C.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{[r \leftarrow C.init(V)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{constructor}\ \mathtt{Object} \quad h(r) = o_t \quad t = \widetilde{\mathtt{Object}}_{\mathtt{j}} \quad h' = h[r \mapsto \mathit{upInit}(o, \mathtt{Object})] \quad V = v_1 :: \ldots :: v_n \quad \langle h', \mathit{InitLocalState}(\mathtt{Object.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{[r \leftarrow \mathtt{Object}.init(V)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{constructor}\ C' \quad h(r) = o_t \quad t = \widetilde{\mathtt{C}} \quad C \subset C' \neq \mathtt{Object} \quad V = v_1 :: \ldots :: v_n \quad \langle h, \mathit{InitLocalState}(\mathtt{C'.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r \leftarrow C'.init(V)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\mathit{instrAt}_P(\mathtt{m},\mathtt{pc}) = \mathtt{constructor}\ \mathtt{Object} \quad h(r) = o_t \quad t = \widetilde{\mathtt{C}} \quad h' = h[r \mapsto \mathit{upInit}(o, \mathtt{C})] \quad V = v_1 :: \ldots :: v_n \quad \langle h', \mathit{InitLocalState}(\mathtt{Object.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'}}{\langle h, m, pc, l, V :: (\mathit{Ref}\ r) :: s\rangle \xrightarrow{\tau.[r \leftarrow \mathtt{Object}.init(V)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

Figure 3.7: BC transition system : error handling

EventStore	$x \leftarrow v$	Local assignment
EventHeap	$r.\mathtt{f} \leftarrow v$	Field assignment
	mayinit(C)	Potential class initialization
	$r \leftarrow \mathtt{C.init}(v_1, \ldots, v_n)$	Object initialization
	$r.\mathtt{C.m}(v_1, \ldots, v_n)$	Dynamic method call
EventReturn	*return*(v)	Method return
	return(*Void*)	

Assigning the top element v of the stack to the local variable `x` (rule for `store x`) gives rise to the observable event $[x \leftarrow v]$. When assigning the value v to the field `f` of the object pointed to by the reference (*Ref r*) (rule for `putfield f`), the transition is labelled by the sequence of events $\tau.[r.\mathtt{f} \leftarrow v]$ (the τ event is introduced in order to match the execution of the BIR assertion generated when transforming this instruction – more detail in Chapter 5).

When returning from a method (rules for `return` and `vreturn`), *Void* or the return value is made observable. We do not observe the "object allocation" event – the memory effect of the instruction `new Cl`. In fact, as will be seen in Section 5.2, the transformation does not preserve the order in which objects are allocated. Both allocation orders could be related using trace languages, but this would make the trace equivalence statement far too complex in relation to the gain of information: as long as no constructor has been called on a reference, no operation is allowed on it, appart from basic stack operations (e.g `dup`), and passing it as a constructor argument. Object allocation order is not preserved by the transformation.

However, we would like the class initialization order to be preserved. In the JVM, classes are initialized at the time of their first use: object creation, static method invocation, or a static field access. A class initialization consists in executing its static initializer, and its static fields initializers. BC does not include static methods or fields. Hence, the class initialization only happens when the first object of a class is created. This is the role of the label *mayinit*(C) in the rule for `new C`. In our work, for sake of clarity, we do not deal with class initialization. However, we introduce all the required material that makes this extension feasible.

Transitions Let us now describe the different kinds of transitions used in the semantics. First, a single transition can give rise to several observable events (see e.g. rule for `putfield` in Figure 3.5). To this end, we use *multi-label transitions*.

Definition 1 (Multi-label transition)**.** *A multi-label transition* $\langle s_1 \rangle \xrightarrow{\vec{\lambda}} \langle s_2 \rangle$ *between state* s_1 *and* s_2 *is a transition labelled by a (finite) sequence of events* $\vec{\lambda} = \lambda_1.\lambda_2 \ldots \lambda_n$.

In rules for method or constructor calls, the execution of the callee has to be considered on its whole from its starting states to its return (or error) state. We thus define *multi-step transitions* as being the transitive closure of transitions: several steps are performed between two states s_1 and s_n but intermediate states of the computation are not distinguished.

Definition 2 (Multi-step transition)**.** *There is a multi-step transition* $\langle s_1 \rangle \overset{\vec{\lambda}}{\Rightarrow} \langle s_n \rangle$ *between states* s_1 *and* s_n *if there exist states* s_2 *up to* s_n *and multi-labels* $\vec{\lambda_1}, \ldots, \vec{\lambda_{n-1}}$ *such that* $\vec{\lambda} = \vec{\lambda_1}.\vec{\lambda_2} \ldots \vec{\lambda_{n-1}}$ *and* $\langle s_1 \rangle \xrightarrow{\vec{\lambda_1}} \langle s_2 \rangle \xrightarrow{\vec{\lambda_2}} \ldots \xrightarrow{\vec{\lambda_{n-2}}} \langle s_{n-1} \rangle \xrightarrow{\vec{\lambda_{n-1}}} \langle s_n \rangle$.

Note that definition of multi-step and multi-label transitions are mutually recursive, from the above definition and the semantic rules for method and constructor calls.

In rule for `invokespecial C.m'` (Figure 3.6), the whole method `m'` is executed and terminates in state $\langle h'', rv\rangle$, using a multi-step transition. This execution produces an event trace $\vec{\lambda}$. This trace contains events related to the local variables of the method, its method calls, some modifications of the heap, and the final return event. While events in *EventStore* and *EventReturn* only concerns `m'` (they are irrelevant for `m`), events related to the heap should be seen outside the method (i.e. from each caller) as well, since the heap is shared between all methods. We hence define the filtering $\vec{\lambda}_c$ of an event trace $\vec{\lambda}$ to a category $c \in \{EventStore, EventHeap, EventReturn\}$ of events as the maximal subtrace of $\vec{\lambda}$ that contains only events in c. Finally, if the method terminates, then the caller `m` makes a multi-label step, the trace $\vec{\lambda}_{EventHeap}$ (written $\vec{\lambda}_h$ in the rules) being exported from the callee `m'`. Constructor calls rules are based on the same idea. Note the semantics does not distinguishes between executions that are blocked and non-terminating.

Finally, each transition of our system is parametrized by a positive integer n representing the *call-depth* of the transition, the number of method calls that arise within this computation step. Concerning *one-step* transitions, this index is incremented when calling a method or a constructor. For multi-step transitions, we define the call-depth index by the following two rules:

$$\frac{\langle s_1\rangle \xrightarrow{\vec{\lambda}}_n \langle s_2\rangle}{\langle s_1\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle s_2\rangle} \qquad \frac{\langle s_1\rangle \overset{\vec{\lambda_1}}{\Rightarrow}_{n_1} \langle s_2\rangle \qquad \langle s_2\rangle \xrightarrow{\vec{\lambda_2}}_{n_2} \langle s_3\rangle}{\langle s_1\rangle \overset{\vec{\lambda_1}.\vec{\lambda_2}}{\Longrightarrow}_{n_1+n_2} \langle s_3\rangle}$$

The call-depth index is mainly introduced for technical reasons. We show the semantics preservation theorem in Chapter 5 by strong induction on the call-depth of the step.

In this chapter, we have defined the source language we consider. BC is a sequential subset of the Java bytecode language, providing object oriented features. We do not include exceptions, but the way we handle execution errors is very similar. The BC semantics is defined in terms of a labelled transition system. Labels are used to keep track of the most possible of behavioural aspects preserved by the transformation. In the next chapter, we define the target language BIR of our tansformation. Its semantics is intentionally very similar to the BC semantics: it is based on the same ideas and uses the same kinds of transitions, so as to easily formulate the semantics preservation in terms of a simulation property.

Chapter 4

The target language: BIR

In this chapter, we describe the intermediate representation language we propose. The BIR language provides a language for expressions, and folds method and constructor calls. In the first section, we describe its syntax and motivate our choices. The second section presents the operational semantics we give to BIR which is very similar to the BC semantics (Section 3.2).

4.1 Syntax of BIR

We already saw how expression trees could increase the precision of static analyses. Many analyses first reconstruct expressions before analysing the bytecode. Hence, the BIR language provides a language for expressions. The language of BIR expressions and instructions is given in Figure 4.1. BIR distinguishes between two kinds of variables: local variables in *var* are identifiers that are also used in the initial BC program, while local variables in *tvar* are fresh variables that are introduced in the BIR. We need to distinguish them in the semantics (see Section 4.2). An expression is either a constant (integers or `null`), a variable, an addition or division of two expressions, or an access to a field of an arbitrary expression (e.g `x.f.g`).

In BIR a variable or the field of a given expression can be assigned with x := *expr* and *expr*.`f` := *expr*. We do not aim at folding control structures: BIR is unstructured and provides conditional and unconditional jumps to a given program point `pc`. Constructors are fold in BIR: new objects are created with the instruction `new C(`*expr*, ... *expr*), and are directly stored in a variable. The reason for folding method and constructor calls is twofold: first, to ease the analyses that often need to relate the allocated reference and the corresponding constructor. Then, as no operation is permitted on an unitialised object, there would be no need to keep the unitialised reference available in a variable. In the constructor of class `C`, the constructor of the super class has to be called. This is done with the instruction *expr*`.super(C',`*expr*,...,*expr*`)`, where `C'` is the super class of `C`. We need to pass `C'` as an argument (unlike in Java source) in order to identify which constructor has to be called (we simplified the initializations tags). The same remarks applies for method calls. Every method ends with a return instruction: `vreturn` *expr* or `return` depending on whether the method returns a value or not (*Void*). When calling a method on an object, the result, if any, must be directly stored in a local variable (as there is no stack in BIR anymore). If the method returns *Void*, the instruction *expr*`.m(C,`*expr*, ..., *expr*) is used.

BIR includes the instruction `mayinit(C)` intended to initialize the class `C` whenever it is required. As already mentioned in the previous chapter, we do not deal with proper class initialization. Hence, this instruction semantically behaves as `nop`. A possible extension taking into account class initializa-

var	::=		*local variables* :
		$\mathtt{x} \mid \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots$	
		`this`	
tvar	::=		*temporary variables* :
		$\mathtt{t} \mid \mathtt{t}_1 \mid \mathtt{t}_2 \mid \ldots$	
x	::=		*variables* :
		var | *tvar*	
oper	::=		*operands*
		$\mathtt{pc} \mid \mathtt{pc}' \mid \ldots$	*program counter*
		$\mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \ldots$	*class name*
		$\mathtt{f} \mid \mathtt{f}' \mid \ldots$	*field name*
		$\mathtt{m} \mid \mathtt{m}' \mid \ldots$	*method name*

expr	::=		*expressions* :
		`c` | `null`	*constants*
		x	*variables*
		expr + *expr*	*addition*
		expr.`f`	*field access*
instr	::=		*instructions* :
		`nop`	
		| `notnull`(*expr*) | `notzero`(*expr*)	
		| `mayinit`(`C`)	
		| *x* := *expr*	
		| *expr*.`f` := *expr*	
		| *x* := `new` `C`(*expr*, ..., *expr*)	
		| *expr*.`super` (`C`, *expr*, ..., *expr*)	
		| *x* := *expr*.`m`(`C`, *expr*, ..., *expr*)	
		| *expr*.`m`(`C`, *expr*, ..., *expr*)	
		| `if` *expr* `pc` | `goto` `pc`	
		| `vreturn` *expr* | `return`	

Figure 4.1: Expressions and instructions of BIR

tion would consist in giving the appropriate semantics to this instruction, as well to the BC instruction `new C`, and to match both executions in the theorem.

Finally, BIR provides two assertions: `notzero`(*expr*) and `notnull`(*expr*). They respectively check whether the value of the expression is zero and null. If this is the case, then the execution of the program continues. Otherwise, the execution goes in a corresponding error state. The benefit brought by these assertions is threefold: it ensures that (i) both BC and BIR error states are reached, in that event, *at the same program point*, (ii) each time a BIR expression is evaluated, its semantics is defined and (iii) the execution of a BIR program never gets stuck but in a BIR error state which is only reached through these assertions.

Figure 4.2 gives the syntax of BIR programs, which is very similar (apart from instructions) from the BC syntax. Like in BC, a BIR method is made of its signature together with its code. The code of a BIR method can be seen as a list of lists of BIR instructions: the transformation algorithm can generate several instructions for a single BC instruction. To make the semantic preservation statement more easy to state and prove, we keep track of this instruction mapping: BIR program instructions are grouped into lists which are indexed by a program counter `pc`: `pc` is the label of the initial BC instruction.

In the following, we denote by $instrsAt_P(\mathtt{m}, \mathtt{pc})$ the list of instructions at label `pc` in the method `m` of program `P`. A BIR class is made of its name, its fields names and its methods. Finally, a BIR program is a set of BIR classes. Here is the BIR version of our example program of Section 3.1. Expressions `x+y` and `z+1` are reconstructed.

```
Test {
  f1 f2
  Test v main: x y : z {
     4: z := x + y
     8: vreturn 1+z
  }
}
```

$$
\begin{array}{lll}
\textit{mcode} & ::= & \textit{method code}: \\
 & \texttt{pc}: \texttt{vreturn}\ \textit{expr} & \textit{method end} \\
 & \texttt{pc}: \texttt{return} & \textit{method end} \\
 & \texttt{pc}: (\textit{instr}\backslash\{\texttt{return}, \texttt{vreturn}\ \textit{expr}\})^* & \\
 & \quad \textit{mcode} & \textit{instr list} \\
r & ::= & \textit{return type}: \\
 & \texttt{void} \mid \texttt{v} & \textit{void/non void} \\
\textit{msig} & ::= & \textit{signature}: \\
 & \texttt{C}\ r\ \texttt{m}: \textit{var} \ldots \textit{var} : \textit{var} \ldots \textit{var} &
\end{array}
$$

$$
\begin{array}{lll}
\textit{method} & ::= & \text{BIR } \textit{method}: \\
 & \textit{msig}\ \{ & \\
 & \quad \texttt{1}: \textit{instr}^* & \\
 & \quad \textit{mcode} & \\
 & \} &
\end{array}
\qquad
\begin{array}{lll}
\textit{class} & ::= & \text{BIR } \textit{class}: \\
 & \texttt{C}\ \{ & \\
 & \texttt{f}_1 \ldots \texttt{f}_n & \\
 & \textit{method}_1 \ldots \textit{method}_m\ \} & \\
\textit{prog} & ::= & \text{BIR } \textit{program}: \\
 & \textit{class}_1 \ldots \textit{class}_p &
\end{array}
$$

Figure 4.2: BIR syntax: methods, classes, programs

4.2 Semantics of BIR

4.2.1 Semantic domains

Semantic domains of BIR are rather similar to those of BC, except that BIR is stackless. They are given in Figure 4.3. The *Value* domain is the same as in BC. An object in BIR also needs to make its initialization status explicit. It is defined similarely than in BIR.

$$
\begin{array}{rcl}
\textit{Value} & = & (\textit{Num}\ n),\ n \in \mathbb{Z} \\
 & & (\textit{Ref}\ r),\ r \in \textit{Reference} \\
 & & \textit{Null} \\
\textit{Env} & = & \textit{var} \cup \textit{tvar} \hookrightarrow \textit{Value} \\
\textit{InitTag} & = & \textit{ClassName} \cup \widetilde{\textit{ClassName}}_{\textit{PCount}} \\
\textit{Object} & = & (\textit{FieldName} \rightarrow \textit{Value})_{\textit{InitTag}} \\
\textit{Heap} & = & \textit{Reference} \hookrightarrow \textit{Object} \\
\textit{State} & = & (\textit{Heap} \times \textit{MethodName} \times (\textit{PCount} \times \textit{instr}^*) \times \textit{Env}) \\
 & & \cup (\textit{Heap} \times \textit{Value} \cup \{\textit{Void}\}) \cup \Omega^{\textit{NP}}_{\textit{PCount}} \cup \Omega^{\textit{DZ}}_{\textit{PCount}}
\end{array}
$$

Figure 4.3: The BIR's semantic domains

As already mentionned, BIR program instructions are organized into lists. Hence, the program counter does not index a single instruction. In a BIR semantic state, the current program point is defined as a pair $(\texttt{pc}, \ell) \in \textit{PCount} \times \textit{instr}^*$ where pc is the program counter and ℓ is the list of instructions being executed. The head element of the list defines the next instruction to execute. More

details is given in the semantic rules about the way the execution flows from one instruction to its successor.

Note that error states are defined as in BC. Still, the *PCount* parameter uniquely determines the faulty program point. As will be seen in the next chapter, at most one assertion is generated per instruction list, and it is always the first instruction of the list.

4.2.2 Semantics of expressions

The semantics of expressions is defined as is standard by induction of the structure of the expression, given an environment and a heap. Expressions are intended to be side-effect free, as it makes easier their treatment in static analyses. As it is clear from the context, we use the same symbols + and / for both syntaxic and semantic versions of the addition and division operators. The semantics of expression is defined by the relation defined on $Heap \times Env \times expr \times Value$:

$$\frac{}{h, l \vDash \mathtt{c} \Downarrow (Num\ c)} \qquad \frac{}{h, l \vDash \mathtt{null} \Downarrow Null} \qquad \frac{x \in dom(l)}{h, l \vDash x \Downarrow l(x)}$$

$$\frac{h, l \vDash e_i \Downarrow (Num\ n_i) \text{ for } i = 1, 2}{h, l \vDash e_1 + e_2 \Downarrow (Num\ (n_1 + n_2))} \qquad \frac{h, l \vDash e_i \Downarrow (Num\ n_i) \text{ for } i = 1, 2 \qquad n_2 \neq 0}{h, l \vDash e_1 / e_2 \Downarrow (Num\ (n_1 / n_2))}$$

$$\frac{h, l \vDash e \Downarrow (Ref\ r),\ r \in Reference \quad r \in dom(h) \quad h(r) = o_{\mathsf{C}} \quad f \in dom(o_{\mathsf{C}})}{h, l \vDash e.f \Downarrow o_{\mathsf{C}}(f)}$$

Figure 4.4: Semantics of BIR expressions

4.2.3 Semantics of instructions

It is very similar to the semantics of BC: it is an observational, operational semantics. We model observational events the same way than in BC. Although they are not relevant in the trace equivalence statement, temporary variables modifications are made observable: we need to distinguish them from the τ event in order to be able to match execution traces. We thus split events in *EventStore* into two event subsets:

$$\begin{array}{rcl} EventStore & = & EventLocalStore \cup EventTempStore \\ & = & \{x \leftarrow v \mid x \in var\} \cup \{x \leftarrow v \mid x \in tvar\} \end{array}$$

Transition rules are given in Figures 4.5, 4.6 and 4.7. Let us now explain how the flow of execution goes from one instruction to the other. Suppose the instruction list being executed is $\ell = i; \ell'$. As can be seen in Figures 4.5 and 4.6, the first instruction $i = hd(\ell)$ of ℓ is first executed. If the flow of control does not jump, then we use the function *next* defined as follows:

$$next(pc, i; \ell') = \begin{cases} (pc + 1, instrsAt_P(\mathtt{m}, \mathtt{pc{+}1})) & \text{if } \ell' = nil \\ (pc, \ell') & otherwise \end{cases}$$

A program P is initially run on $instrsAt_P(\mathtt{main}, 1)$.

As will be seen in the next chapter, the generated BIR instruction lists are never empty. Hence, the function *next* is well defined. Moreover, when the control flow jumps, the instruction list to execute is directly identified by the label of the jump target (e.g. rule for `goto`).

In the rule for object creation (Figure 4.6), note that the freshly created object is directly considered as being initialized, thanks to the constructor folding: as soon as the object has been allocated, its constructor is called thus its status updated. No instruction can be executed between the object allocation and its constructor call.

Semantic rules for assertions are also rather intuitive: either the assertion passes, and the execution goes on, or it fails and the execution of the program is aborted in the corresponding error state.

Concerning rules for handling execution errors, notice that the BIR semantics suggests more blocking states than BC. For instance, no semantic rule can be applied when trying to execute a method call on a null pointer. Here, we do not need to take into account this case: the transformation algorithm generates an assertion when translating method call instruction. This assertion catches the null pointer dereferencing attempt.

Apart from these points, rules of BIR use the same principles as BC rules. Hence, we do not further comment them. Syntaxes and semantics of BC and BIR are now defined. The next chapter defines the transformation algorithm *BC*2*BIR*, and shows that the semantics of BC is preserved by the transformation.

$$\frac{hd(\ell) = \texttt{nop}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, next(pc, \ell), l\rangle} \qquad \frac{hd(\ell) = x := expr \quad h, l \vDash expr \Downarrow v}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, next(pc, \ell), l[x \mapsto v]\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = expr.\texttt{f} := expr' \\ h, l \vDash expr \Downarrow (Ref\ r) \quad h(r) = o_C \quad h, l \vDash expr' \Downarrow v \quad o' = o[f \mapsto v] \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.f \leftarrow v]}_0 \langle h[r \mapsto o'], m, next(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{if}\ expr\ \texttt{pc'} \\ h, l \vDash expr \Downarrow (Num\ 0) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, (pc', instrsAt_P(\texttt{m}, \texttt{pc'})), l\rangle} \qquad \frac{\begin{array}{c} hd(\ell) = \texttt{if}\ expr\ \texttt{pc'} \\ h, l \vDash expr \Downarrow (Num\ n) \quad n \neq 0 \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, next(pc, next(pc, \ell)), l\rangle}$$

$$\frac{hd(\ell) = \texttt{goto pc'}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, (pc', instrsAt_P(\texttt{m}, \texttt{pc'})), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{vreturn}\ expr \\ h, l \vDash expr \Downarrow v \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[return(v)]}_0 \langle h, v\rangle} \qquad \frac{hd(\ell) = \texttt{return}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[return(Void)]}_0 \langle h, Void\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notnull}(expr) \\ h, l \vDash expr \Downarrow (Ref\ r) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, next(pc, \ell), l\rangle} \qquad \frac{\begin{array}{c} hd(\ell) = \texttt{notnull}(expr) \\ h, l \vDash expr \Downarrow Null \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notzero}(expr) \\ h, l \vDash expr \Downarrow (Num\ n) \quad n \neq 0 \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, next(pc, \ell), l\rangle} \qquad \frac{\begin{array}{c} hd(\ell) = \texttt{notzero}(expr) \\ h, l \vDash expr \Downarrow (Num\ 0) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \Omega^{DZ}_{pc}}$$

$$\frac{hd(\ell) = \texttt{mayinit}(\texttt{C})}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[mayinit(\texttt{C})]}_0 \langle h, next(pc, \ell), l\rangle}$$

Figure 4.5: BIR transition system

$$\forall \mathtt{m} \in \mathit{MethodName}, \mathit{InitLocalState}(\mathtt{m}) = \mathtt{m}, (1, \mathit{instrsAt}_P(\mathtt{m}, 1)), [\mathit{this} \mapsto (\mathit{Ref}\ r), \mathtt{x}_1 \mapsto v_1 \ldots \mathtt{x}_n \mapsto v_n]$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = x{:=}\mathtt{new\ C}\ (e_1, \ldots, e_n) \\ h, l \vDash e_i \Downarrow v_i \quad (h_0, (\mathit{Ref}\ r)) = \mathit{newObject}(C, h) \\ h' = h_0[r \mapsto o_t] \quad t = \widetilde{\mathtt{C}} \\ \langle h', \mathit{InitLocalState}(\mathtt{C.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow C.\mathit{init}(v_1,\ldots,v_n)].\vec{\lambda}_h.[x \leftarrow (\mathit{Ref}\ r)]}_{n+1} \langle h'', (m, \mathit{next}(pc, \ell), l[x \mapsto (\mathit{Ref}\ r)]\rangle}$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = \mathtt{e.super(C, e_1, \ldots, e_n)} \\ h, l \vDash e \Downarrow (\mathit{Ref}\ r) \quad h, l \vDash e_i \Downarrow v_i \\ C' \subseteq C \neq \mathtt{Object} \quad h(r) = o_t \quad t = \widetilde{\mathtt{C}'} \\ \langle h, \mathit{InitLocalState}(\mathtt{C.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', \mathit{Void}\rangle \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow C.\mathit{init}(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \langle h', m, \mathit{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = e.\mathtt{super(Object, e_1, \ldots, e_n)} \\ h, l \vDash e \Downarrow (\mathit{Ref}\ r) \quad h, l \vDash e_i \Downarrow v_i \\ h(r) = o_t \quad t = \widetilde{\mathtt{C}} \quad h' = h[r \mapsto \mathit{upInit}(o, \mathtt{C})] \\ \langle h', \mathit{InitLocalState}(\mathtt{Object.init})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h'', \mathit{Void}\rangle \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow \mathit{Object.init}(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \langle h'', m, \mathit{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = y := \ \mathtt{e.m'(C, e_1, \ldots, e_n)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (\mathit{Ref}\ r) \\ h(r) = o_{\mathtt{C}'} \qquad \mathit{Lookup}(m', \mathtt{C}') = mc \\ \langle h, \mathit{InitLocalState}(\mathtt{mc})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', v\rangle \quad v \neq \mathit{Void} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda}_h.[y \leftarrow v]}_{n+1} \langle h', m, \mathit{next}(pc, \ell), l[y \mapsto v]\rangle}$$

$$\frac{\begin{array}{c} \mathit{hd}(\ell) = \mathtt{e.m'(C, e_1, \ldots, e_n)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (\mathit{Ref}\ r) \\ h(r) = o_C \qquad \mathit{Lookup}(m', \mathtt{C}') = \mathtt{mc} \\ \langle h, \mathit{InitLocalState}(\mathtt{mc})\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', \mathit{Void}\rangle \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \langle h', m, \mathit{next}(pc, \ell), l\rangle}$$

Figure 4.6: BIR transition system: object initialisation and method calls

$$\frac{\begin{array}{c} hd(\ell) = x{:=}\texttt{new C}\ (e_1, \ldots, e_n) \\ h, l \vDash e_i \Downarrow v_i \quad (h_0, (\mathit{Ref}\ r)) = \mathit{newObject}(C, h) \\ h' = h_0[r \mapsto o_t] \quad t = \widetilde{\texttt{C}} \\ \langle h', \mathit{InitLocalState}(\texttt{C.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow C.init(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{e.super(C, e}_1, \ldots, \texttt{e}_n\texttt{)} \\ h, l \vDash e \Downarrow (\mathit{Ref}\ r) \quad h, l \vDash e_i \Downarrow v_i \\ C' \subseteq C \neq \texttt{Object} \quad h(r) = o_t \quad t = \widetilde{\texttt{C}'} \\ \langle h, \mathit{InitLocalState}(\texttt{C.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow C.init(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\begin{array}{c} hd(\ell) = e\texttt{.super(Object, e}_1, \ldots, \texttt{e}_n\texttt{)} \\ h, l \vDash e \Downarrow (\mathit{Ref}\ r) \quad h, l \vDash e_i \Downarrow v_i \\ h(r) = o_t \quad t = \widetilde{\texttt{C}} \quad h' = h[r \mapsto \mathit{upInit}(o, \texttt{C})] \\ \langle h', \mathit{InitLocalState}(\texttt{Object.init})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r \leftarrow Object.init(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\begin{array}{c} hd(\ell) = y := \ \texttt{e.m}'\texttt{(C, e}_1, \ldots, \texttt{e}_n\texttt{)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (\mathit{Ref}\ r) \\ h(r) = o_{C'} \qquad \mathit{Lookup}(m', \texttt{C}') = mc \\ \langle h, \mathit{InitLocalState}(\texttt{mc})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{e.m}'\texttt{(C, e}_1, \ldots, \texttt{e}_n\texttt{)} \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (\mathit{Ref}\ r) \\ h(r) = o_C \qquad \mathit{Lookup}(m', \texttt{C}') = \texttt{mc} \\ \langle h, \mathit{InitLocalState}(\texttt{mc})\rangle \vec{\Rightarrow}_n \Omega^k_{pc'} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\vec{\lambda}_h}_{n+1} \Omega^k_{pc}}$$

Figure 4.7: BIR transition system: error cases

Chapter 5

BC2BIR is semantics-preserving

In this chapter, we present the transformation algorithm from BC to BIR and the properties it satisfies. It is based on a symbolic execution of the bytecode, using a symbolic operand stack. Each bytecode instruction modifies the abstract stack and gives rise to the generation of BIR instructions. It is very similar to the BC2IR algorithm of the Jalapeño Optimizing Compiler [Wha99]. Many other similar algorithms exist (see e.g [CFM+97], [XX99] or [WCN05]). Our contribution mainly lies in its formalization (Section 5.1) and especially the proof of its semantics preservation property (Section 5.2).

5.1 The BC2BIR algorithm

This algorithm transforms BC programs into BIR programs. It is based on a symbolic execution of the bytecode that uses an abstract stack, containing symbolic expressions, as was used in the interval analysis of Besson et al [BJP06]. Whereas this work performs the expression decompilation during the analysis, the BC2BIR algorithm can be thought of as a preprocessing pass of the static analysis: expression decompilation and constructor folding are done once and for all.

As the transformation symbolically executes the bytecode, it uses a stack of symbolic expressions:

Definition 3 (*AbstrStack*)**.**
Abstract stacks are defined as $AbstrStack = SymbExpr^*$*, where* $SymbExpr = expr \cup ExprUninit(\mathsf{C}, \mathsf{pc})$.

Expressions in *expr* are decompiled expressions of BC, while $ExprUninit(\mathsf{C}, \mathsf{pc})$ is used as a placeholder for freshly allocated references in order to fold constructor calls. It denotes a reference pointing to a yet uninitialized object allocated in the heap by the instruction `new C` at program point `pc`:

$$h, l \vDash ExprUninit(\mathsf{C}, \mathsf{pc}) \Downarrow Dummy(\mathsf{C}, \mathsf{pc})$$

We need to keep the class (`C`) and program counter (`pc`) information in the value of this placeholder. In fact, this information make consistant the substitution operation on abstract stacks used in the transformation when folding constructors. Semantic domains as defined in Chapters 3 and 4 have more or less the same name. In the following, we use language subscripts (BC or BIR) to desambiguate domains.

The heart of the algorithm is a transformation, $BC2BIR_i$, that converts a BC instruction into a list of BIR instructions. This basic transformation is then somewhat iterated on the whole code of a method. $BC2BIR_i$ is defined as a function

$$BC2BIR_i : PCount \times instr_{BC} \times AbstrStack \rightarrow \left(instr^*_{BIR} \times AbstrStack\right) \cup Fail$$

Given an abstract stack, $BC2BIR_i$ modifies it according to the BC instruction at program point pc, and returns a list of BIR instructions. *Fail* is returned when the transformation does not succeed. We will need the program counter parameter for object creation and initialization. $BC2BIR_i$ is given Figure 5.1, where the $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ denote fresh temporary variables introduced at point pc. They are used to keep the abstract stack elements coherent with the value they should represent. Every undescribed case (e.g. the stack height or content mismatches) yields *Fail*.

$$
\begin{array}{ll}
BC2BIR_i(\mathtt{pc}, \mathtt{nop}, as) & = ([\mathtt{nop}], as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{push\ c}, as) & = ([\mathtt{nop}], \mathtt{c} :: as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{pop}, \mathtt{e} :: as) & = ([\mathtt{nop}], as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{dup}, \mathtt{e} :: as) & = ([\mathtt{nop}], \mathtt{e} :: \mathtt{e} :: as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{load\ x}, as) & = ([\mathtt{nop}], \mathtt{x} :: as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{if\ pc'}, \mathtt{e} :: as) & = ([\mathtt{if\ e\ pc'}], as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{goto\ pc'}, as) & = ([\mathtt{goto\ pc'}], as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{vreturn}, \mathtt{e} :: as) & = ([\mathtt{return\ e}], as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{return}, as) & = ([\mathtt{return}], as)
\end{array}
$$

$$
\begin{array}{ll}
BC2BIR_i(\mathtt{pc}, \mathtt{add}, \mathtt{e}_1 :: \mathtt{e}_2 :: as) & = ([\mathtt{nop}], (\mathtt{e}_1 + \mathtt{e}_2) :: as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{div}, \mathtt{e}_1 :: \mathtt{e}_2 :: as) & = ([\mathtt{notzero}(\mathtt{e}_2)], (\mathtt{e}_1/\mathtt{e}_2) :: as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{getfield\ f}, \mathtt{e} :: as) & = ([\mathtt{notnull}(\mathtt{e})], \mathtt{e.f} :: as) \\
BC2BIR_i(\mathtt{pc}, \mathtt{store\ x}, \mathtt{e} :: as) & = \begin{cases} ([\mathtt{x := e}], as) & \text{if } \mathtt{x} \notin as^a \\ ([\mathtt{t}^{\mathtt{0}}_{\mathtt{pc}} := \mathtt{x}; \mathtt{x := e}], as[\mathtt{t}^{\mathtt{0}}_{\mathtt{pc}}/\mathtt{x}]) & \text{if } \mathtt{x} \in as^a \end{cases} \\
BC2BIR_i(\mathtt{pc}, \mathtt{new\ C}, as) & = ([\mathtt{mayinit(C)}], \mathit{ExprUninit}(\mathtt{C}, \mathtt{pc}) :: as)
\end{array}
$$

$$
\begin{array}{ll}
BC2BIR_i(\mathtt{pc}, \mathtt{putfield\ f}, & \mathtt{e'} :: \mathtt{e} :: as) = \\
 & ([\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} := \mathtt{e}_1; \ldots; \mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} := \mathtt{e}_n; \mathtt{e.f := e'}], as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/\mathtt{e}_i])^{ab} \\
BC2BIR_i(\mathtt{pc}, \mathtt{invokevirtual\ C.m}, & \mathtt{e}_1 :: \ldots :: \mathtt{e}_n :: \mathtt{e} :: as) = \\
 & \begin{cases} ([\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} := \mathtt{e}'_1; \ldots; \mathtt{t}^{\mathtt{m}}_{\mathtt{pc}} := \mathtt{e}'_m; \mathtt{t}^{\mathtt{0}}_{\mathtt{pc}} := \mathtt{e.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)], \mathtt{t}^{\mathtt{0}}_{\mathtt{pc}} :: as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}'_j]) & \text{if m returns a value}^{ac} \\ ([\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} := \mathtt{e}'_1; \ldots; \mathtt{t}^{\mathtt{m}}_{\mathtt{pc}} := \mathtt{e}'_m; \mathtt{e.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)], as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}'_j]) & \text{otherwise}^{a\ c} \end{cases} \\
BC2BIR_i(\mathtt{pc}, \mathtt{constructor\ C}, & \mathtt{e}_1 :: \ldots :: \mathtt{e}_n :: \mathtt{e} :: as) = \\
 & \begin{cases} ([\mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} := \mathtt{e}'_1; \ldots; \mathtt{t}^{\mathtt{m}}_{\mathtt{pc}} := \mathtt{e}'_m; \mathtt{t}^{\mathtt{0}}_{\mathtt{pc}} := \mathtt{new\ C}(\mathtt{e}_1, \ldots, \mathtt{e}_n)], as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}'_j][\mathtt{t}^{\mathtt{0}}_{\mathtt{pc}}/\mathtt{e}]) & \text{if e=}\mathit{ExprUninit}(\mathtt{C}, \mathtt{pc'})^c \\ ([\mathtt{notnull(e)}; \mathtt{t}^{\mathtt{1}}_{\mathtt{pc}} := \mathtt{e}'_1; \ldots; \mathtt{t}^{\mathtt{m}}_{\mathtt{pc}} := \mathtt{e}'_m; \mathtt{e.super}(\mathtt{C}, \mathtt{e}_1, \ldots, \mathtt{e}_n), as[\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}/\mathtt{e}'_j]]) & \text{otherwise}^c \end{cases}
\end{array}
$$

Figure 5.1: $BC2BIR_i$ – BC instruction transformation

[a] where $\mathtt{e} \neq \mathit{ExprUninit}(\mathtt{C}, \mathtt{pc'})$

[b] where $\mathtt{e}_i, \mathtt{i} = 1 \ldots n$ are all the elements of as such that $\mathtt{f} \in \mathtt{e}_i$

[c] where $\mathtt{e}'_j, \mathtt{j} = 1 \ldots m$ are all the elements of as that read a field

Transforming the bytecode `push c`, given an abstract stack as, consists in pushing the symbolic expression `c` on as and generating the BIR instruction `nop`. Cases of `pop`, `dup`, `add` and `load` are similar. Note that generated BIR instruction list is never empty: in order to simplify the lemmas and their proof, we generate a BIR `nop` instruction even in the `push c`, `pop`, `dup`, `add` and `iload`. This will make the step-matching easier in the simulation argument. All `nop` instructions could be removed in BIR program, without changing anything to its semantics. Transformations of return instructions are straightforward, as well as conditional and unconditional jumps.

For instruction `div`, generating `nop` is not sufficient for properly preserving execution errors: an assertion `notzero(e)` has to be generated. Figure 5.2 gives an example (source, BC, and BIR versions) of a simple function returning the third of its integer argument. Figure 5.2(c) gives, for each program point, the entry abstract used by the transformation. Let us illustrate with this example how this assertion is used to preserve the potential division by zero. At program point 3 in Figure 5.2(b), if the second top element of the (concrete) stack, i.e the value of `x2` is zero, executing the instruction `div` leads to the error state Ω_3^{DZ}. This execution error is matched in BIR at program point 3, because the generated assertion `notzero(x2)` will fail, leading to the BIR error state Ω_3^{DZ}. Otherwise, the assertion successes, and the execution goes on. Without this assertion, the potential division by zero would happen at program point 4 (when the expression is evaluated), leading to Ω_4^{DZ}. Even worst, it could never happen, in case the code were dead.

```
int f(int x1, int x2) {return (x1/x2) ;}
```
(a) source function

```
int f(x1,x2);
1: load x2;
2: load x1;
3: div;
4: vreturn;
```
(b) BC function

```
1: []
2: [x2]
3: [x1::x2]
4: [x1/x2]
```
(c) symbolic stack

```
int f(x1,x2);
1: nop;
2: nop;
3: notzero(x2);
4: vreturn x1/x2;
```
(d) BIR function

Figure 5.2: Example of bytecode transformation – error handling

The `getfield f` instruction reads the field `f` of the object pointed to by the reference on the top of the stack. The BIR assertion `notnull(e)` is generated. Here again, the goal of this assertion is to make sure that, if a null pointer is dereferenced by `getfield f`, the BIR program reaches the error state program point. Similar assertions are generated for constructor and method calls.

In the case of `store x`, there are two cases. The simpliest one is when the symbolic expression `x` is not used in the abstract stack *as*: the top expression of the stack is popped and the BIR instruction `x := e` is generated. Now, if `x` is used in *as*. Suppose we only generate `x := e` and pop `e` off the stack, the remainding expression `x` would not represent the old value of the variable `x` anymore, but the new one. Hence, before generating `x := e`, we have to store the old value of `x` in a fresh temporary variable $\mathtt{t}^{0}_{\mathtt{pc}}$. Then, every occurrence of the variable `x` in *as* has to be substituted for the new temporary $\mathtt{t}^{0}_{\mathtt{pc}}$. This substitution is written $as[\mathtt{t}^{0}_{\mathtt{pc}}/\mathtt{x}]$ and is defined in a standard way (inductively on the length of the abstract stack, and on the structure of the expression). Notice that for this instruction, we additionaly demand that the expression `e` is not *ExprUninit*(`C`, `pc'`): no valid BIR instruction could match this case, as constructors are fold. This hypothesis is made explicit: it is not part of the constraints checked by the BCV, as it fits the JVM specification. The same remark can be done about writing an object field (`putfield f`) and method calls.

The `putfield f` case is similar, except that there is more to save than the only `e.f`: because of aliasing, any expression $\mathtt{e}_\mathtt{i}$ could be evaluated to the same value (reference) than the variable `e`. Hence, when modifying `e.f`, $\mathtt{e}_\mathtt{i}$`.f` could be modified as well. We choose to store in fresh temporaries every element of *as* where the field `f` is used. Here, the substitutions by $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ is element-wise. More temporary variables than needed are introduced. This could be refined using an alias analysis. Note there is no conflict between substitutions as all $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are fresh.

There are two cases when transforming object method calls: either the method returns a value, or not (*Void*). This information is available in the method signature of the bytecode. In the former case, a fresh variable is introduced in the BIR in order to store the value of the result, and this variable is pushed onto the abstract stack as the result of the method call. The execution of the callee might modify the value of all heap cells. Hence, before executing the method call, every abstract stack element that accesses the heap must be stored in a fresh variable, in order to remember its value.

Let us describe now how *BC*2*BIR*$_i$ is dealing with object creation and initialization. We use the program in Figure 5.3 as a running example. The source program is given in Figure 5.3(a): it stores in the local variable `x` an object of class `A`, whose constructor is called with the field `f` of another object of class `A` as argument. When symbolically executing the bytecode `new C` at program point `pc`, *ExprUninit*(`C`, `pc`) is pushed on the abstract stack *as*. As already said, *ExprUninit*(`C`, `pc`) is the symbolic expression that corresponds to freshly allocated references. Recall we do not treat class initialization. If this were the case, in order to preserve the class initialization order, BIR should initialize the class `C` at this point as well. This is the reason for generating instruction `mayinit(C)`

at this point. In Figures 5.3(b), 5.3(c) and 5.3(c), it corresponds to program points 1 and 3 (for space reason, in Figure 5.3(c), we abbreviate *ExprUninit*(C, pc) by *EU*(C, pc)).

There are two ways for transforming the instruction `constructor C`. If this is the first constructor call on the reference (and in the abstract stack, on *ExprUninit*(C, j)), then the constructor is fold at this point: the BIR instruction $\mathtt{t}^{\emptyset}_{\mathtt{pc}} := \mathtt{new\ C}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ is generated. For instance, this is the case at points 6 and 8 in Figure 5.3(d). Note that *ExprUninit*(C, j) uniquely identify a given object, thanks to the alias information provided by *j*: (i) we assume the BC program to pass the BCV, no such expression could be in the stack after a backward branching, (ii) moreover, we do not allow to store uninitialized references in local variables (it does not belong to the BIR language of expressions). In BIR, every newly created (and initialized) object is stored in a local variable (here $\mathtt{t}^{\emptyset}_{\mathtt{pc}}$). In the resulting abstract stack, each occurrence of *ExprUninit*(C, j) is replaced with the variable $\mathtt{t}^{\emptyset}_{\mathtt{pc}}$. In Figure 5.3, at program point 6, the constructor arguments and *EU*(A, 1) are popped off the abstract stack, and the remainding *EU*(A, 1) (distinguished from *EU*(A, 3)) is replaced by $\mathtt{t}^{\emptyset}_{6}$.

The second case is when `constructor C` is called on an already created object, and is actually a call to a super constructor. Here, no new object should be created, and we have to decompile another way. The reference pointing to the current object can be the value any expression (including `this`). The generated BIR instruction is thus $\mathtt{e.super}(\mathtt{C}, \mathtt{e}_1, \ldots, \mathtt{e}_n)$, where C is a super class of the class of the current object, and whose name is available in the bytecode instruction.

```
void f() { A x = new A( (new A(1)).f) ; return ;}
```
(a) source function

```
void f();x;
1: new A;
2: dup;
3: new A;
4: dup;
5: push 1;
6: constructor A;
7: getfield f;
8: constructor A;
9: store x;
10: return;
```
(b) BC function

```
1: []
2: [EU(A,1)]
3: [EU(A,1)::EU(A,1)]
4: [EU(A,3)::EU(A,1)::EU(A,1)]
5: [EU(A,3)::EU(A,3)::EU(A,1)::EU(A,1)]
6: [1::EU(A,3)::EU(A,3)::EU(A,1)::EU(A,1)]
7: [t⁰₆::EU(A,1)::EU(A,1)]
8: [t⁰₆.f::EU(A,1)::EU(A,1)]
9: [t⁰₈]
10: []
```
(c) symbolic stack

```
void f();x;
1: mayinit(A)
2: nop;
3: mayinit(A);
4: nop;
5: nop;
6: t⁰₆ := new A(1);
7: notnull(t⁰₆);
8: t⁰₈ := new A(t⁰₆.f);
9: x:= t⁰₈;
10: return;
```
(d) BIR function

Figure 5.3: Example of bytecode transformation – Constructor folding

The basic instruction-wise transformation $BC2BIR_i$ is used in the algorithm to generate the BIR of a method. An entire BC program is obtained by translating each method of the class, and so for each class of the program. Figure 5.4 gives the algorithm transforming a whole method m of a given program P, where *length*(m) is the length of the code of *m* and $succ_{\mathtt{m}}(\mathtt{pc})$ is the set of all the successors of pc in the method *m*. We write *stackSize*(pc) for the (precomputed) size of the abstract stack at program point pc. We additionaly need to compute the set of branching points of the method:

$$jmpTgt^{\mathtt{P}}_{\mathtt{m}} = \{\ j \mid \exists \mathtt{pc}, instrAt_P(\mathtt{m}, \mathtt{pc}) = \mathtt{if\ j}\ \text{or}\ \mathtt{goto\ j}\ \}$$

All this information can be easily, statically computed and is thus supposed available at hand.

Along the algorithm, three arrays are computed. `IR[m]` contains the intermediate representation of the method m: for each pc, `IR[m,pc]` is the list of generated instructions. $\mathtt{AS}_{\mathtt{in}}[\mathtt{m}]$ is an array of entry symbolic stacks, required to compute IR and $\mathtt{AS}_{\mathtt{out}}[\mathtt{m}]$ contains the output abstract stack resulting from the instruction transformation.

Basically, transforming the whole code of a BC method consists in iterating the $BC2BIR_i$ function, passing on the abstract stack from one instruction to its successors. If the basic transformation fails, so

```
1  function BC2BIR(P,m) =
2    AS_in[m,1] := nil
3    for (pc = 1, pc ≤ length(m), pc++) do
4
5      // Compute the entry abstract stack
6      if (pc ∈ jmpTgt^P_m) then AS_in[m,pc] := newStackJmp(pc, stackSize(pc)) end
7      if (pc ∉ jmpTgt^P_m ∧ succ_m(pc) ∩ jmpTgt^P_m ≠ ∅) then
8        as_in := newStack(pc, stackSize(pc))
9      else as_in := AS_in[m,pc] end
10
11     // Decompile instruction
12     AS_out[m,pc], IR[m,pc] := BC2BIR_i(pc, instrAt_P(m,pc), as_in)
13     if ( succ_m(pc) ∩ jmpTgt^P_m ≠ ∅) then
14       if ( ∀b ∈ succ_m(pc) ∩ jmpTgt^P_m, b ∉ AS_in[m,pc] ) then
15         IR[m,pc] := paraAssignts(succ_m(pc), AS_out[m,pc]) ++ IR[m,pc]
16       else IR[m,pc] := Assignts(pc, AS_in[m,pc]) ++ paraAssignts(succ_m(pc), AS_out[m,pc]) ++ IR[m,pc]
17       end
18     end
19
20     // Pass around the output abstract stack
21     if (pc + 1 ∈ succ_m(pc) ∧ pc + 1 ∉ jmpTgt^P_m) then AS_in[m,pc+1] := AS_out[m,pc]  end
22 end
```

Figure 5.4: *BC2BIR* – BC method transformation

does the algorithm on the whole method. The algorithm consists in: (i) computing the entry abstract stack $\mathtt{as}_{\mathtt{in}}$ used by $BC2BIR_i$ (from Line 6 to 9) to transform the instruction, (ii) performing the BIR generation (from Line 12 to 16) and (iii) passing on the output abstract stack (Line 21)

Notice the transformation is performed on a single, linear pass on the bytecode. When the flow of control is linear (from pc to pc+1), the abstract stack resulting from $BC2BIR_i$ is transmitted as it is (Line 21). The case of control flow joins must be handled more carefully.

```
int f(int x) {return (x==0) ? 1 : -1;}
```

(a) source function

```
int f(x);
1: load x
2: if 5
3: push -1
4: goto 6

5: push 1
6: vreturn
```

(b) BC function

```
1: []
2: [x]
3: []
4: [-1]

5: []
6: [T^1_6]
```

(c) symbolic stack

```
int f(x);
1: nop;
2: if x == 0 goto 5;
3: nop;
4: T^1_6 := -1;
   goto 6;
5: T^1_6 := 1;
6: vreturn T^1_6;
```

(d) BIR function

Figure 5.5: Example of bytecode transformation – non-empty stack jumps

Thanks to the BCV hypothesis we make on the bytecode, we already know that at every branching point, the size of the stack is the same regardless of the predecessor point. Still, for this one-pass transformation to be correct, the content of the abstract stack must be uniquely determined at these points: stack elements are expressions used in the generated instructions and hence must be independant of the control flow path leading to these program points.

Let us illustrate this point with the example function of Figure 5.5. It returns 1 or -1 depending of whether the argument `x` is zero or not. Let us focus on program point 6, i.e a branching point : its predecessors are points 4 and 5, depending on the conditional at point 2. The abstract stack after having executed the instruction `goto 6` is `[-1]` (point 4), while it becomes `[1]` after program point 5. But, when transforming the `vreturn` at point 6, the abstract stack should be uniquely determined, since the control flow is unknown.

The idea here is to store, before reaching a branching point, every stack element in a temporary variable and then to use an abstract stack made of all of these variables at the branching point. A naming convention has to be decided so that (i) identifiers do not depend on the control flow path and (ii) each variable corresponds to exactly one stack element: we use the identifier $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ to store the i^{th} element of the stack when the jump target point is `pc`. Hence, for each $\mathtt{pc} \in \mathit{jmpTgt}^{\mathtt{P}}_{\mathtt{m}}$, the abstract stack used by $BC2BIR_i$ is (almost) entirely determined by `pc` and the size of the entry stack at this point. In Figures 5.5(c) and 5.5(d), at program points 4 and 5, we respectively store 1 and -1 in $\mathtt{T}^{1}_{6}$, a temporary variable that will be used as point 6 in the entry abstract stack of the transformation.

Thus, in the algorithm, when transforming a BC instruction that preceeds a branching point, the list or BIR instructions provided by $BC2BIR_i$ is no longer sufficient: we must prepend to it a list of assignments of each abstract stack element to $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ variables, and so for each potential target point `pc`. These assignments must happen before the instruction list given by $BC2BIR_i$, in case a jumping instruction were generated by $BC2BIR_i$ at this point (see e.g program point 4 in Figure 5.5(d)).

Suppose now the stack before the branching point `pc` contains an uninitialized reference, represented by $\mathit{ExprUninit}(\mathtt{C}, \mathtt{pcn})$. As this element is not a BIR expression, it cannot be replaced by any temporary variable – the assignment would not be a legal BIR instruction. Here, we need to assume the following structural constraint on the bytecode: before a branching point `pc`, if the stack contains any uninitialized reference at position i, then it is the case for every predecessor of `pc`. More formally, this hypothesis can be formulated as a constraint on the array $\mathtt{AS}_{\mathtt{out}}$:

$$\begin{array}{rll}
\forall \mathtt{pc}, \mathtt{pcn}, \mathtt{C}, \mathtt{i}. & (\exists \mathtt{pc}'. & \mathtt{pc} \in \mathit{succ}_{\mathtt{m}}(\mathtt{pc}') \\
& \quad \wedge & \mathtt{pc}' < \mathtt{pc} \\
& \quad \wedge & \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = \mathit{ExprUninit}(\mathtt{C}, \mathtt{pcn})) \\
\Rightarrow & (\forall \mathtt{pc}'. & \mathtt{pc} \in \mathit{succ}_{\mathtt{m}}(\mathtt{pc}') \\
& \quad \wedge & \mathtt{pc}' < \mathtt{pc} \\
& \quad \wedge & \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = \mathit{ExprUninit}(\mathtt{C}, \mathtt{pcn}))
\end{array}$$

Without this requirement, because constructors are fold in BIR the transformation would fail. We use the function *newStackJmp* (Line 6) defined as follows to compute the entry abstract stack at branching point `pc`, where n is the size of the abstract stack:

$$\mathit{newStackJmp}(\mathtt{pc}, n) \quad = \quad \mathtt{e}_1 :: \ldots :: \mathtt{e}_{\mathtt{n}}$$

$$\text{where } \forall i = 1 \ldots n, \mathtt{e}_{\mathtt{i}} = \begin{cases} \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} & \text{if } \exists \mathtt{pcn}, \mathtt{pc}', \mathtt{C}. \quad \mathtt{pc} \in \mathit{succ}_{\mathtt{m}}(\mathtt{pc}') \wedge \mathtt{pc}' < \mathtt{pc} \\ & \text{such that } \mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}']_{\mathtt{i}} = \mathit{ExprUninit}(\mathtt{C}, \mathtt{pcn}) \\ \mathtt{T}^{\mathtt{i}}_{\mathtt{pc}} & \text{otherwise} \end{cases}$$

Notice the use of this function is coherent with $\mathtt{AS}_{\mathtt{in}}[\mathtt{m}, 1]$: even if 1 is a branching point, the stack at the beginning of the method is empty.

Now, before reaching the branching point, we have to make sure all the $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ have been assigned. Given an abstract stack as and a set S of program points, $\mathit{paraAssignts}(\mathtt{S}, as)$ (Lines 15 and 16) returns the list of such assignments which are ensured to be mutually conflict-free – in case as contained some of the $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$, new variables would be used.

A last precaution has to be taken here. In case where some $\mathtt{T}^{\mathtt{i}}_{\mathtt{pc}}$ appeared in the entry stack used by the basic transformation, the value of these variables must be remembered: the semantics of the instruction list generated by $BC2BIR_i$ depends on them. This can only happens at non-branching points. In this case, the entry abstract stack is changed to $newStack(\mathtt{pc}, n)$ (Line 8), where n is the size of the stack at point pc. The corresponding assignments are generated by $Assignts(\mathtt{pc}, AS_{in}[m, pc])$ (Line 16).

$$newStack(\mathtt{pc}, n) \quad = \quad \mathtt{e}_1 :: \ldots :: \mathtt{e}_n$$

$$\text{where } \forall i = 1 \ldots n, \mathtt{e}_{\mathtt{i}} = \begin{cases} \tilde{\mathtt{T}}^{\mathtt{i}}_{\mathtt{pc}} & \text{if } \exists \mathtt{pc}', \mathtt{k}. \quad \mathtt{pc}' \in succ_{\mathtt{m}}(\mathtt{pc}) \\ & \wedge \quad \mathtt{T}^{\mathtt{k}}_{\mathtt{pc}'} \in \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}]_{\mathtt{i}} \\ \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}]_{\mathtt{i}} & \text{otherwise} \end{cases}$$

In the following section, we make further remarks on the algorithm and formalize the semantics preservation property of the above algorithm.

5.2 Semantics preservation

The BC2BIR algorithm we presented in the previous section is semantics-preserving. In this section, we formalize this notion of semantics preservation. The basic idea is that the BIR program $BC2BIR(P)$ simulates the initial BC program P and both have similar execution traces. The similarity between traces is defined using an equivalence relation over the two heaps. Although the two heaps are not equal, they keep related through a partial bijection[1]. This is due to the fact that the transformation does not preserves the order in which objects are allocated. Section 5.2.1 demonstrates this point using a simple example. In Section 5.2.2, we define the equivalence relations induced by this heap similarity. We need them in the propositions of Section 5.2.3 to express the execution trace preservation.

5.2.1 Object allocation orders

Starting from the same heap, execution of P and $P' = BC2BIR(P)$ will not preserve this equality. However, the two heaps keep isomorphic: there exists a partial bijection between them. Here, we illustrate this fact on the simple example given in Figure 5.6, where temporary variable identifiers have been simplified. The corresponding Java source is

```
A x = new A(new B(),3); return
```

The JVM specification regarding the operand stack when calling a constructor on a object is the following: the reference to the object has to be pushed on the stack before other arguments. This implies that the object must be created before other arguments are calculated. In the program of Figure 5.6(a), the object of class `A` is hence allocated before the object of class `B` is passed as argument to the former's constructor. In the BIR version of the program (Figure 5.6(b)), as constructors are fold, and because an object creation is not expression, the object of class `B` has to created (and initialized) before passing the temporary variable `t1` (that contains its reference) as an argument to the constructor of the object of class `A`.

When executing program in Figure 5.6, one object of class `A` is allocated at program point `1` through the reference r_1. At program point `3`, a second object is allocated in the heap, with the

[1]The rigorous definition of a bijective function demands that it is totally defined on its domain. The term "partial bijection" is however widely used and we consider it as equivalent to "partial injection".

```
void Test.mtest () {
  1 : new A
  2 : dup
  3 : new B
  4 : dup
  5 : constructor B
  6 : push 3
  7 : constructor A
  8 : store x
  9 : return
}
```

(a) BC program P

```
void Test.mtest () {
  1 : mayinit(A)
  2 : nop
  3 : mayinit(B)
  4 : nop
  5 : t1 := new B()
  6 : nop
  7 : t2 := new A(t1,3)
  8 : x := t2
  9 : return
}
```

(b) BIR program $P' = BC2BIR(P)$

Figure 5.6: Example program: BC and BIR versions

reference r_2. Whereas in P the `B` object is pointed to by r_2, it is pointed to by r_3 in P', and similarely for object of class `A` that is pointed to by r_4.

Heaps are not equal along the execution of the two programs: at program point `4` in P, the heap contains two objects while in P', the heap is still empty. However, after program points `5`, we know that each time the reference r_1 is used in P', it corresponds to the use of r_4 is P (both constructors have been called, so both references can be used freely). The same reasoning can be applied just after program points `7`: r_2 in P corresponds to r_3 in P'.

A bijection thus exists between references of programs P and P': heaps are equal modulo the allocation history. At each program point, objects being initialized (the constructor has just been called) are pointed to by references that are in the partial bijection. In our example program, the partial bijection is empty until program points `4`. At these points, the bijection is extendend with $\beta(r_1) = r_4$. β is then again extended at programs points `7` with $\beta(r_2) = r_3$.

The notion of a partial bijection relating two heaps has been earlier used in [BN05] and [BR05]. These works aim at ensuring the safety of information flows in Java and Java bytecode programs. Objects allocated in a high security context should not leak information. The bijection relates references to the so-called "high objects". Intuitively, the property that they want to ensure is: running a program starting from two isomorphic heaps leads to two indistinguishable executions.

5.2.2 Equivalence relations

As stated in the previous section, the object allocation order is not preserved by the algorithm, but there exists a partial bijection between the BC and BIR heaps that relates allocated objects as soon as their initialization is ongoing. In order to relate heaps, environments and execution traces, equivalence relations need to be defined. We need them in Section 5.2.3 to state and prove the semantics preservation. All of these are parametrised by the current partial bijection $\beta : \mathit{Reference} \hookrightarrow \mathit{Reference}$. First, an equivalence relation is defined over values:

Definition 4 (Value equivalence: $\overset{v}{\sim}_\beta$)**.**
The relation $\overset{v}{\sim}_\beta \subseteq \mathit{Value} \times \mathit{Value} \cup \mathit{Dummy}(\mathsf{C}, \mathsf{pc})$ is defined inductively by:

$$\frac{}{\mathit{Null} \overset{v}{\sim}_\beta \mathit{Null}} \quad \frac{}{\mathit{Void} \overset{v}{\sim}_\beta \mathit{Void}} \quad \frac{n \in \mathbb{Z}}{(\mathit{Num}\ n) \overset{v}{\sim}_\beta (\mathit{Num}\ n)} \quad \frac{\beta(r_1) = r_2}{(\mathit{Ref}\ r_1) \overset{v}{\sim}_\beta (\mathit{Ref}\ r_2)} \quad \frac{r_1 \notin \mathit{dom}(\beta)}{(\mathit{Ref}\ r_1) \overset{v}{\sim}_\beta \mathit{Dummy}(\mathsf{C}, \mathsf{pc})}$$

The interesting case is for references. References related by β are equivalent. Concerning objects on which no constructor has been called yet (i.e. a reference that is not in the domain of β), references pointing to them are equivalent to the special value *Dummy*.

We can now define the equivalence relation on heaps. First, only objects existing in the two heaps must be related by β. Secondly, the related objects are at least being initialized and must have the same initialization status, hence the same class. Finally, their fields must have equivalent values. More formally:

Definition 5 (Heap equivalence: $\overset{\text{H}}{\sim}_\beta$)**.**
Let h_1 and h_2 be two heaps. We have $h_1 \overset{\text{H}}{\sim}_\beta h_2$ if and only if:

- $dom(\beta) \subseteq dom(h_1)$ *and* $rng(\beta) \subseteq dom(h_2)$
- $\forall r \in dom(\beta), \mathsf{C} \in ClassName$ *such that* $h_1(r) = o_t$ *with* $t \in \{\mathsf{C}, \widetilde{\mathsf{C}}\}$*, we have*

$$(i)\ h_2(\beta(r)) = o'_t \qquad (ii)\ \forall f \in dom(o_t).\ o_t(f) \overset{\text{V}}{\sim}_\beta o'_t(f)$$

The proof of the semantics preservation will be done using a simulation diagram scheme. We thus have to relate environments: an BIR environment is equivalent to a BC environment if and only if both local variables (temporary variables are not taken into account) have equivalent values.

Definition 6 (Environment equivalence: $\overset{\text{E}}{\sim}_\beta$)**.**
Let $l_1 \in Env_{BC}$ and $l_2 \in Env_{BIR}$ be two environments. We have $l_1 \overset{\text{E}}{\sim}_\beta l_2$ if and only if: $dom(l_1) \subseteq dom(l_2)$ and $\forall x \in dom(l_1).\ l_1(x) \overset{\text{V}}{\sim}_\beta l_2(x)$

Finally, in order to relate execution traces, we need to define an equivalence relation over events. It is extended pointwise as is standard to event traces.

Definition 7 (Event equivalence: $\overset{!}{\sim}_\beta$)**.**
The equivalence relation over events $\overset{!}{\sim}_\beta$ is inductively defined:

$$\frac{}{\tau \overset{!}{\sim}_\beta \tau} \qquad \frac{}{mayinit(\mathsf{C}) \overset{!}{\sim}_\beta mayinit(\mathsf{C})} \qquad \frac{x \in var \quad v_1 \overset{\text{V}}{\sim}_\beta v_2}{[x \leftarrow v_1] \overset{!}{\sim}_\beta [x \leftarrow v_2]} \qquad \frac{\beta(r_1) = r_2 \quad v_1 \overset{\text{V}}{\sim}_\beta v_2}{[r_1.f \leftarrow v_1] \overset{!}{\sim}_\beta [r_2.f \leftarrow v_2]}$$

$$\frac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\text{V}}{\sim}_\beta v'_i}{[r_1 \leftarrow C.init(v_1, \ldots, v_n)] \overset{!}{\sim}_\beta [r_2 \leftarrow C.init(v'_1, \ldots, v'_n)]} \qquad \frac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\text{V}}{\sim}_\beta v'_i}{[r_1.C.m(v_1, \ldots, v_n)] \overset{!}{\sim}_\beta [r_2.C.m(v'_1, \ldots, v'_n)]}$$

5.2.3 Semantics preservation

Relating run-time and abstract stacks is the first step towards bridging the gap between the two representations of P. We thus define an equivalence relation over stacks. This relation holds between s and as if the ith element of as evaluates, given a BIR heap and environment, to a value that is equivalent w.r.t $\overset{\text{V}}{\sim}_\beta$ to the ith element of s, and so for each i:

Definition 8 (Stack equivalence: $\approx_{h,l,\beta}$)**.**
Let s be in Stack, as be in AbstrStack, h be in $Heap_{BIR}$ and l in Env_{BIR}. The stack equivalence is defined inductively as:

$$\frac{}{\varepsilon \approx_{h,l,\beta} \varepsilon} \qquad \frac{h, l \vDash e \Downarrow v' \quad v \overset{\text{V}}{\sim}_\beta v' \quad s \approx_{h,l,\beta} as}{v :: s \approx_{h,l,\beta} e :: as}$$

The proof correctness of the transformation is organized as follows. We show that one-step transitions are preserved by the basic transformation $BC2BIR_i$. These results will be used in the proof of one-step transitions by $BC2BIR$: the propositions hold, regarless of the potential assignments added in the algorithm. Finally, multi-step transitions will be shown to be preserved by $BC2BIR$ using a strong induction of the call-depth. Propositions are first stated in their normal and then in their error-execution variants. In the following, to lighten the notations, we write $\vec{\lambda}_{proj}$ for $\vec{\lambda}_{EventLocalStore \cup EventHeap \cup EventReturn}$, i.e. the projection of the trace $\vec{\lambda}$ to any category of events but *EventTempStore*.

Proposition 1 ($BC2BIR_i$ - zero call-depth one-step preservation - normal case)**.**
Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', m, pc', l', s'\rangle$. *Let* ht, lt, as, β *be such that:*

$$h \overset{\text{H}}{\sim}_\beta ht \qquad l \overset{\text{E}}{\sim}_\beta lt \qquad s \approx_{ht,lt,\beta} as \qquad BC2BIR_i(pc, instrAt_P(\mathtt{m}, \mathtt{pc}), as) = (\ell, as')$$

Then, there exist unique ht', lt' *and* $\vec{\lambda'}$ *such that* $\langle ht, m, (pc, \ell), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', m, (pc', instrsAt_P(\mathtt{m}, \mathtt{pc'})), lt'\rangle$ *with:*

$$h' \overset{\text{H}}{\sim}_\beta ht' \qquad l' \overset{\text{E}}{\sim}_\beta lt' \qquad \vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj} \qquad s' \approx_{ht',lt',\beta} as'$$

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', rv\rangle$, *then there exists* $\langle ht', rv'\rangle$ *such that:* $\langle ht, m, (pc, \ell), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', rv'\rangle$, *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_\beta ht'$ *and* $rv \overset{\text{V}}{\sim}_\beta rv'$.

Proof. We proceed by case analysis on the BC instruction at program point *pc*. Here, only interesting cases are detailed. Others are trivial or can be treated a similar way.

- **push** c We have $\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc + 1, l, (Num\ c) :: s\rangle$. Let as, ht, lt be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} as$. We have $BC2BIR_i(pc, \mathtt{push}\ c, as) = ([\mathtt{nop}], c :: as)$. Hence, $\langle ht, m, (pc, [\mathtt{nop}]), lt\rangle \xrightarrow{\tau} \langle ht, m, (pc + 1, instrsAt_P(\mathtt{m}, \mathtt{pc{+}1})), lt\rangle$. The heaps and environments are unchanged, both transitions are silent. Stacks stay trivially equivalent since $ht, lt \vDash c \Downarrow (Num\ c)$.

- **div** Here, because the execution does not reach the error state, only one case is possible : $n_2 \neq 0$, and $\langle h, m, pc, l, (Num\ n_1) :: (Num\ n_2) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc + 1, l, (Num\ n_1/n_2) :: s\rangle$. Let as, ht, lt be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $(Num\ n_1) :: (Num\ n_2) :: s \approx_{ht,lt,\beta} e_1 :: e_2 :: as$. We have $BC2BIR_i(pc, \mathtt{div}, e_1 :: e_2 :: as) = ([\mathtt{notzero(e_2)}], e_1/e_2 :: as)$. But $ht, lt \vDash e_2 \Downarrow (Num\ n_2')$ with $(Num\ n_2) \overset{\text{V}}{\sim}_\beta (Num\ n_2')$. Thus, $n_2' \neq 0$ and $\langle ht, m, (pc, [\mathtt{notzero(e_2)}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc + 1, instrsAt_P(\mathtt{m}, \mathtt{pc{+}1})), lt\rangle$. Heaps and environment are unchanged, and both transitions are silent. Finally, since $ht, lt \vDash e_1 \Downarrow (Num\ n_1')$ with $(Num\ n_1) \overset{\text{V}}{\sim}_\beta (Num\ n_1')$ and $ht, lt \vDash e_2 \Downarrow (Num\ n_2')$ with $(Num\ n_2) \overset{\text{V}}{\sim}_\beta (Num\ n_2')$, we have $ht, lt \vDash e_1/e_2 \Downarrow (Num\ n_1'/n_2')$ and $(Num\ n_1/n_2) \overset{\text{V}}{\sim}_\beta (Num\ n_1'/n_2')$.

- **load x** We have $\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc + 1, l, l(x) :: s\rangle$. Let as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} as$. We have $BC2BIR_i(pc, \mathtt{load}\ \mathtt{x}, as) = ([\mathtt{nop}], x :: as)$. Hence, $\langle ht, m, (pc, [\mathtt{nop}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc + 1, instrsAt_P(\mathtt{m}, \mathtt{pc{+}1})), lt\rangle$. Heaps and environments are unchanged and both transitions are silent. We now have to prove that stacks stay equivalent, i.e. that $l(x) :: s \approx_{ht,lt,\beta} x :: as$. We have $ht, lt \vDash x \Downarrow lt(x)$, $l \overset{\text{E}}{\sim}_\beta lt$ and $x \in var$. Hence, by the definition of $\overset{\text{E}}{\sim}_\beta$, we have $l(x) \overset{\text{V}}{\sim}_\beta lt(x)$.

- **`store x`** We have $\langle h, m, pc, l, v :: s\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, pc+1, l[x \mapsto v], s\rangle$. Let as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $v :: s \approx_{ht,lt,\beta} e :: as$. We distinguish two cases, whether x is already in as or not:
  - If $x \notin as$ then $BC2BIR_i(pc, \texttt{istore x}, e :: as) = ([\texttt{x := e}], as)$. But $v :: s \approx_{ht,lt,\beta} e :: as$ and $ht, lt \vDash e \Downarrow v'$ with $v \overset{V}{\sim}_\beta v'$. Hence $\langle ht, m, (pc, [\texttt{x := e}]), lt\rangle \xrightarrow{[x \leftarrow v']}_0 \langle ht, m, (pc+1, instrsAt_P(\texttt{m}, \texttt{pc+1})), lt[x \mapsto v']\rangle$. Now, heaps are not modified, and stay equivalent. Labels are equivalent: we have $x \in var$ because it is used in a bytecode instruction, and $v \overset{V}{\sim}_\beta v'$. Thus $[x \leftarrow v] \overset{!}{\sim}_\beta [x \leftarrow v']$. Environment stay equivalent: $l[x \mapsto v] \overset{E}{\sim}_\beta lt[x \mapsto v']$ since $l \overset{E}{\sim}_\beta lt$ by hypothesis and $v \overset{V}{\sim}_\beta v'$. We finally have to prove that $s \approx_{ht,lt',\beta} as$, where $lt' = lt[x \mapsto v']$. Stacks are the same height. Moreover, as $x \notin as$, for all abstract stack elements as_i, we have: $ht, lt' \vDash as_i \Downarrow v'_i$ and $ht, lt \vDash as_i \Downarrow v'_i$ with $v_i \overset{V}{\sim}_\beta v'_i$.
  - If $x \in as$ then $BC2BIR_i(pc, \texttt{istore x}, e :: as) = ([\texttt{t}^{\texttt{0}}_{\texttt{pc}} := \texttt{x}; \texttt{x := e}], as[\texttt{t}^{\texttt{0}}_{\texttt{pc}}/\texttt{x}])$. We hence have that $\langle ht, m, (pc, [\texttt{t}^1_{\texttt{pc}} := \texttt{x}; \texttt{x := e}]), lt\rangle \xrightarrow{[\texttt{t}^1_{\texttt{pc}} \leftarrow lt(x)]}_0 \langle ht, m, (pc, [\texttt{x := e}]), lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x)]\rangle$. $\texttt{t}^1_{\texttt{pc}}$ is fresh, so $\texttt{t}^1_{\texttt{pc}} \notin e$.

    Hence $ht, lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x)] \vDash e \Downarrow v'$ where v' is such that $ht, lt \vDash e \Downarrow v'$, and $v \overset{V}{\sim}_\beta v'$ by hypothesis. Thus, we have $\langle ht, m, (pc, [\texttt{t}^1_{\texttt{pc}} := \texttt{x}; \texttt{x := e}]), lt\rangle \xrightarrow{[\texttt{t}^1_{\texttt{pc}} \leftarrow lt(\texttt{x})].[\texttt{x} \leftarrow v']}_0 \langle ht, m, (pc+1, instrsAt_P(\texttt{m}, \texttt{pc+1})), lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x), \texttt{x} \mapsto v']\rangle$.

    Heaps are not modified. We have $[x \leftarrow v] \overset{E}{\sim}_\beta \left([\texttt{t}^1_{\texttt{pc}} \leftarrow lt(x)].[x \leftarrow v']\right)_{proj} = [x \leftarrow v']$ because only $\texttt{t}^1_{\texttt{pc}} \in tvar$ and $v \overset{V}{\sim}_\beta v'$. Environments stay equivalent because $\texttt{t}^1_{\texttt{pc}} \in tvar$ and $x \in var$ is assigned the value v' with $v \overset{V}{\sim}_\beta v'$.

    We now have to show that $s \approx_{ht,lt',\beta} as[\texttt{t}^1_{\texttt{pc}}/x]$, where $lt' = lt[\texttt{t}^1_{\texttt{pc}} \mapsto lt(x), x \mapsto v']$. But for all elements $as[\texttt{t}^1_{\texttt{pc}}/x]_i$ of the abstract stack, we have: $ht, lt' \vDash as[\texttt{t}^1_{\texttt{pc}}/x]_i \Downarrow v_i$ where v_i is such that $ht, lt \vDash as_i \Downarrow v_i$ because $lt'(\texttt{t}^1_{\texttt{pc}}) = lt(x)$ and $\texttt{t}^1_{\texttt{pc}}$ is fresh, so $\texttt{t}^1_{\texttt{pc}} \notin as_i$.

- **`if pc'`** According to the top element of the stack, there are two cases. We only treat the case of a jump, the other one is similar. We have $\langle h, m, pc, l, (Num\ 0) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle$. Let as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $(Num\ 0) :: s \approx_{ht,lt,\beta} e :: as$. We have $BC2BIR_i(pc, \texttt{if pc'}, e :: as) = ([\texttt{if } e\ pc'], as)$. But stacks are equivalent by hypothesis, thus e evaluates to zero and $\langle ht, m, (pc, [\texttt{if } e\ pc']), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc', instrsAt_P(\texttt{m}, \texttt{pc'})), lt\rangle$ and labels are equivalent. Heaps and environments are unchanged. Stacks stay trivially equivalent.

- **`new C`** We have $\langle h, m, pc, l, s\rangle \xrightarrow{mayinit(\texttt{C})}_0 \langle h', m, pc+1, l, (Ref\ r) :: s\rangle$, with $(Ref\ r)$ freshly allocated and $h' = h[r \mapsto Blank(\widetilde{\texttt{C}}_{\texttt{pc}})]$. Let as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} as$. We have that $BC2BIR_i(pc, \texttt{new } C, as) = ([\texttt{mayinit(C)}], ExprUninit(\texttt{C}, \texttt{pc}) :: as)$. Hence $\langle ht, m, (pc, [\texttt{mayinit(C)}]), lt\rangle \xrightarrow{[mayinit(\texttt{C})]}_0 \langle ht, m, (pc+1, instrsAt_P(\texttt{m}, \texttt{pc+1})), lt\rangle$. Labels are equal and environments are not modified. The reference $(Ref\ r)$ is pointing to an uninitialized object in h', so β is not extended, and heaps keep related. Finally, we have $(Ref\ r) :: s \approx_{ht,lt,\beta} ExprUninit(\texttt{C}, \texttt{pc}) :: as$ because $ht, lt \vDash ExprUninit(\texttt{C}, \texttt{pc}) \Downarrow Dummy(\texttt{C}, \texttt{pc})$ and $(Ref\ r) \overset{V}{\sim}_\beta Dummy(\texttt{C}, \texttt{pc})$ because $r \notin dom(\beta)$.

- **`getfield f`** The execution does not reach the error state. Hence, we have $\langle h, m, pc, l, (Ref\ r) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, h(r)(f) :: s\rangle$, with $h(r) = o_{\texttt{C}}$. Let e, as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and

$(Ref\ r) :: s \approx_{ht,lt,\beta} e :: as$. We have $BC2BIR_i(pc, \texttt{getfield f}, e :: as) = ([\texttt{notnull(e)}], e.f :: as)$. By hypothesis on the stacks, we have that $ht, lt \vDash e \Downarrow (Ref\ r')$. Hence, `e` does not evaluates to *Null* and $\langle ht, m, (pc, [\texttt{notnull(e)}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc+1, instrsAt_P(\mathtt{m}, \mathtt{pc{+}1})), lt\rangle$. Heaps and environments are not modified, labels are equivalent. We now have to show that stacks keep related. By hypothesis, we have $\beta(r) = r'$ since the object pointed to by r is initialized. Besides, $ht, lt \vDash e.f \Downarrow ht(r')(f)$ since $ht, lt \vDash e \Downarrow (Ref\ r')$ and $ht(r')(f) = ht(\beta(r))(f)$. We know that $h \overset{\mathrm{H}}{\sim}_\beta ht$ by hypothesis, hence $h(r)(f) \overset{\mathrm{V}}{\sim}_\beta ht(\beta(r'))(f)$. Stacks are hence equivalent.

- **`putfield f`** We have $\langle h, m, pc, l, v :: (Ref\ r) :: s\rangle \xrightarrow{\tau.[r.f \leftarrow v]}_0 \langle h[r(f) \mapsto v], m, pc+1, l, s\rangle$ (the field of the object pointed to by r is modified), with $h(r) = o_\mathtt{C}$. Let e, e', as, ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $v :: (Ref\ r) :: s \approx_{ht,lt,\beta} e' :: e :: as$. There are two cases:

  - If `f` is not in any expression of the abstract stack, we have $BC2BIR_i(pc, \texttt{putfield f}, e' :: e :: as) = ([\texttt{notnull(e)}; \texttt{e.f := e'}], as)$. But $v :: (Ref\ r) :: s \approx_{ht,lt,\beta} \texttt{e'} :: \texttt{e} :: as$. We get that $v \overset{\mathrm{V}}{\sim}_\beta v'$ where $ht, lt \vDash \texttt{e'} \Downarrow v'$ and that there exists r' such that $ht, lt \vDash \texttt{e} \Downarrow (Ref\ r')$ with $(Ref\ r) \overset{\mathrm{V}}{\sim}_\beta (Ref\ r')$, and r' points in ht to an initialized object, since the BC field assignment is permitted.

    We hence have $\langle ht, m, (pc, [\texttt{notnull(e)}; \texttt{e.f := e'}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc, [\texttt{e.f := e'}]), lt\rangle \xrightarrow{[r'.f \leftarrow v']}_0 \langle ht[r'(f) \mapsto v'], m, (pc+1, instrsAt_{pc+1}(), ,)lt\rangle$. Environments are unchanged and stay related. We have to show that $h' = h[r(f) \mapsto v] \overset{\mathrm{H}}{\sim}_\beta ht' = ht[r'(f) \mapsto v']$. We have $(Ref\ r) \overset{\mathrm{V}}{\sim}_\beta (Ref\ r')$, hence $\beta(r) = r'$. Besides, $v \overset{\mathrm{V}}{\sim}_\beta v'$ with $ht, lt \vDash e' \Downarrow v'$. Fields of the two objects pointed to by r and r' have hence equivalent values w.r.t β. Finally, we have $\tau.[r.f \leftarrow v] \overset{!}{\sim}_\beta \tau.[r'.f \leftarrow v']$ since $v \overset{\mathrm{V}}{\sim}_\beta v'$ and $(Ref\ r) \overset{\mathrm{V}}{\sim}_\beta (Ref\ r')$.
  - If $\texttt{f} \in as$, we have $BC2BIR_i(pc, \texttt{putfield f}, e' :: e :: as) = ([\texttt{notnull(e)}; \mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := as_i; e.f := e'], as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i])$. But $v :: (Ref\ r) :: s \approx_{ht,lt,\beta} e' :: e :: as$ hence, as in the previous case: $v \overset{\mathrm{V}}{\sim}_\beta v'$ where v' is such that $ht, lt \vDash e' \Downarrow v'$ and there exists r' such that $ht, lt \vDash e \Downarrow (Ref\ r')$ with $(Ref\ r) \overset{\mathrm{V}}{\sim}_\beta (Ref\ r')$.

    Suppose now that n elements of as are expressions using the field `f`. For all $i \in [1; n]$, let v_i be such that $ht, lt \vDash as_i \Downarrow v_i$. Thus, $\langle ht, m, (pc, [\texttt{notnull(e)}; \mathtt{t}^{\mathtt{i}}_{\mathtt{pc}} := as_i; e.f := e']), lt\rangle \xrightarrow{\tau.[\mathtt{t}^1_{\mathtt{pc}} \leftarrow v_1] \ldots [\mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \leftarrow v_n]}_0 \langle ht, m, (pc, [e.f := e']), lt[\mathtt{t}^1_{\mathtt{pc}} \mapsto v_1, \ldots \mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \mapsto v_n]\rangle$.
    All $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are fresh, they hence do not appear in e or e'. Let $lt' = lt[\mathtt{t}^1_{\mathtt{pc}} \mapsto v_1, \ldots \mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \mapsto v_n]$. Thus $ht, lt' \vDash e' \Downarrow v'$ with $ht, lt \vDash e' \Downarrow v'$ and $ht, lt' \vDash e \Downarrow (Ref\ r')$. Thus, $\langle ht, m, (pc, [e.f := e']), lt'\rangle \xrightarrow{[r'.f \leftarrow v']}_0 \langle ht[r'(f) \mapsto v'], m, (pc+1, instrsAt_P(\mathtt{m}, \mathtt{pc{+}1})), lt'\rangle$.
    Events are equivalent: $\tau.[r.f \leftarrow v] \overset{!}{\sim}_\beta (\tau.[\mathtt{t}^1_{\mathtt{pc}} \mapsto v_1] \ldots [\mathtt{t}^{\mathtt{n}}_{\mathtt{pc}} \leftarrow v_n].[r'.f \leftarrow v'])_{proj}$ because all $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are in $tvar$, $\beta(r) = r'$ and $v \overset{\mathrm{V}}{\sim}_\beta v'$. Environments stay equivalent: $l \overset{\mathrm{E}}{\sim}_\beta lt'$ because all $\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}$ are fresh. Besides, since $\beta(r) = r'$ and $v \overset{\mathrm{V}}{\sim}_\beta v'$, we have $h[r(f) \mapsto v] \overset{\mathrm{H}}{\sim}_\beta ht[r'(f) \mapsto v']$. Finally, we have that $s \approx_{ht',lt',\beta} as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i]$, where $ht' = ht[r'(f) \mapsto v']$, by the definition of $as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/as_i]$.

□

Proposition 2 ($BC2BIR_i$ - zero call-depth one-step preservation - error case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \Omega^k_{pc'}$. *Let* ht, lt, as, β *be such that:*

$$h \overset{\mathrm{H}}{\sim}_\beta ht \qquad l \overset{\mathrm{E}}{\sim}_\beta lt \qquad s \approx_{ht,lt,\beta} as \qquad BC2BIR_i(pc, instrAt_P(\mathtt{m}, \mathtt{pc}), as) = (\ell, as')$$

Then, there exist unique $\vec{\lambda'}$ *such that* $\langle ht, m, (pc, \ell), lt\rangle \xrightarrow{\vec{\lambda'}}_0 \Omega^k_{pc'}$ *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$

Proof. Here again, we proceed by case analysis on the instruction at program point pc.

- **div** Here, only one case is possible: $\langle h, m, pc, l, (Num\ n_1) :: (Num\ 0) :: s\rangle \xrightarrow{\tau}_0 \Omega^{DZ}_{pc}$. Let as, ht, lt be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $(Num\ n_1) :: (Num\ 0) :: s \approx_{ht,lt,\beta} e_1 :: e_2 :: as$. We have $BC2BIR_i(pc, \texttt{div}, e_1 :: e_2 :: as) = ([\texttt{notzero(e}_2\texttt{)}], e_1/e_2 :: as)$. But $ht, lt \vDash e_2 \Downarrow (Num\ n'_2)$ with $(Num\ 0) \overset{V}{\sim}_\beta (Num\ n'_2)$. Thus, $n'_2 = 0$ and $\langle ht, m, (pc, [\texttt{notzero(e}_2\texttt{)}]), lt\rangle \xrightarrow{\tau}_0 \Omega^{DZ}_{pc}$ and both transitions are silent.

- **getfield** The execution reaches the error state. Hence, we have $\langle h, m, pc, l, Null :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$. Let e, as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $Null :: s \approx_{ht,lt,\beta} e :: as$. We have $BC2BIR_i(pc, \texttt{getfield f}, e :: as) = ([notnull(e)], e.f :: as)$. By hypothesis on the stacks, we have that $ht, lt \vDash e \Downarrow Null$ and $\langle ht, m, (pc, [notnull(e)]), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$.

- **putfield f** We have $\langle h, m, pc, l, v :: Null :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$. Let e, e', as, ht, lt, β be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $v :: Null :: s \approx_{ht,lt,\beta} e' :: e :: as$. We have $BC2BIR_i(pc, \texttt{putfield f}, e' :: e :: as) = ([\texttt{notnull(e)}; \texttt{t}^{\texttt{i}}_{\texttt{pc}} := as_i; e.f := e'], as[\texttt{t}^{\texttt{i}}_{\texttt{pc}}/as_i])$. But $v :: Null :: s \approx_{ht,lt,\beta} e' :: e :: as$, hence $ht, lt \vDash e \Downarrow Null$. Thus, $\langle ht, m, (pc, [\texttt{notnull(e)}; \texttt{t}^{\texttt{i}}_{\texttt{pc}} := as_i; e.f := e']), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$.

- **invokevirtual** We only treat here the case where the method returns *Void*. We have $\langle h, m, pc, l, v_1 :: \ldots :: v_n :: Null :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$. Let $e_i, e, as, ht, lt, \beta$ be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $v_1 :: \ldots :: v_n :: Null :: s \approx_{ht,lt,\beta} e_1 :: \ldots :: e_n :: e :: as$. We have $BC2BIR_i(pc, \texttt{invokevirtualC.m}', e_1 :: \ldots :: e_n :: e :: as) = ([\texttt{notnull(e)}; \texttt{t}^{\texttt{1}}_{\texttt{pc}} := \texttt{e}'_1; \ldots; \texttt{t}^{\texttt{m}}_{\texttt{pc}} := \texttt{e}'_{\texttt{m}}; \texttt{e.m(e}_1, \ldots, \texttt{e}_\texttt{n}\texttt{)}], as[\texttt{t}^{\texttt{j}}_{\texttt{pc}}/\texttt{e}'_{\texttt{j}}])$. $ht, lt \vDash e \Downarrow Null$. Thus, $\langle ht, m, (pc, [\texttt{notnull(e)}; \texttt{t}^{\texttt{j}}_{\texttt{pc}} := e'_j; \texttt{e.m(e}_1, \ldots, \texttt{e}_\texttt{n}\texttt{)}]), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$.

- **constructor C** We have $\langle h, m, pc, l, v_1 :: \ldots :: v_n :: Null :: s\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$. Let $e_i, e, as, ht, lt, \beta$ be such that $h \overset{H}{\sim}_\beta ht$, $l \overset{E}{\sim}_\beta lt$ and $v_1 :: \ldots :: v_n :: Null :: s \approx_{ht,lt,\beta} e_1 :: \ldots :: e_n :: e :: as$. By the hypothesis on the stacks, we know that $\texttt{e} \neq ExprUninit(\texttt{C}, \texttt{pc'})$ since e should evaluate to *Null*. Then, $BC2BIR_i(pc, \texttt{constructorC}, e_1 :: \ldots :: e_n :: e :: as) = ([\texttt{notnull(e)}; \texttt{t}^{\texttt{1}}_{\texttt{pc}} := \texttt{e}'_1; \ldots; \texttt{t}^{\texttt{m}}_{\texttt{pc}} := \texttt{e}'_{\texttt{m}}; \texttt{e.super(e}_1, \ldots, \texttt{e}_\texttt{n}\texttt{)}], as[\texttt{t}^{\texttt{j}}_{\texttt{pc}}/\texttt{e}'_{\texttt{j}}])$. But $ht, lt \vDash e \Downarrow Null$. Thus, $\langle ht, m, (pc, [\texttt{notnull(e)}; \texttt{t}^{\texttt{j}}_{\texttt{pc}} := e'_j; \texttt{e.super(e}_1, \ldots, \texttt{e}_\texttt{n}\texttt{)}]), lt\rangle \xrightarrow{\tau}_0 \Omega^{NP}_{pc}$.

□

With the two above propositions, we can now show that the algorithm given in Figure 5.4 preserves zero-call depth one-step transitions.

Proposition 3 (*BC2BIR* - zero call-depth one-step preservation - normal case)**.**
Suppose we have $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', m, pc', l', s'\rangle$ *and* ht, lt, β *are s.t.*

$$h \overset{H}{\sim}_\beta ht \qquad l \overset{E}{\sim}_\beta lt \qquad s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$$

Then there exist unique $ht', lt', \vec{\lambda'}$ *such that* $\langle ht, m, (pc, \texttt{IR}[\texttt{m}, \texttt{pc}]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', m, (pc', \texttt{IR}[\texttt{m}, \texttt{pc'}]), lt'\rangle$ *with:*

$$h' \overset{H}{\sim}_\beta ht' \qquad l' \overset{E}{\sim}_\beta lt' \qquad \vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj} \qquad s' \approx_{ht',lt',\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc'}]$$

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', rv\rangle$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}]$ *then there exists a unique* $\langle ht', rv'\rangle$ *such that:* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', rv'\rangle$*, with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$*,* $h' \overset{\mathrm{H}}{\sim}_\beta ht'$ *and* $rv \overset{\mathrm{V}}{\sim}_\beta rv'$*.*

Proof. Suppose $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_0 \langle h', m, pc', l', s'\rangle$ and ht, lt, β are such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $s \approx_{ht,lt,\beta} \mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}]$. What differs from basic $BC2BIR_i$ transformation is that (i) the entry abstract stack is not always transmitted as it is from one instruction to its successors and (ii) additional assignments might be preprended to the BIR instruction basically generated.

The proof is hence organized as follows. We first have to show that s and $\mathtt{as_{in}}$ (the actual abstract stack used in the basic transformation) keep related in the possibly modified environment. This intermediate result makes us able to use Proposition 1. Finally, we must ensure that the transmitted abstract stack is equivalent to s' with respect to the new BIR heap and environment obtained by executing the basic BIR instructions.

First, we show that s and $\mathtt{as_{in}}$ keep equivalent with regards to the potentially modified environment. There are two cases whether pc is a branching point or not.

- If $\mathtt{pc} \in jmpTgt^{\mathrm{P}}_{\mathtt{m}}$, then $\mathtt{as_{in}}$ is set to $\mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}]$. Now, assignments potentially generated by $Assignts(\mathtt{pc}, \mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}])$ and $paraAssignts(succ_{\mathtt{m}}(\mathtt{pc}), \mathtt{AS_{out}}[\mathtt{m},\mathtt{pc}])$ have to be taken into account. We show they do not alterare the stack equivalence between s and $\mathtt{as_{in}}$. There are two cases according to whether successors of pc are branching points or not.
  - If none of them is a branching point, then no additional assignment is generated. Hence, the local environment *lt* is not modified and stacks keep related.
  - Suppose now some successors of pc are branching points (denoted by pcb). First, because pc is a branching point, the entry stack $\mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}]$ has been normalized.
    - if $\mathtt{pc} \neq \mathtt{pcb}$, the condition Line **??** is satisfied: none of the assigned $\mathtt{T^k_{pcb}}$ can be used in the elements of $\mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}] = \mathtt{as_{in}}$. Hence, assignments do not modify the stack equivalence.
    - if $\mathtt{pc} = \mathtt{pcb}$, then the condition is not meet. In this case, the instruction at point pc is `goto pc`. Then assignments are $\tilde{\mathtt{T}}^{\mathtt{j}}_{\mathtt{pc}} := \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}; \ldots; \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}} := \mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}$. If pc is not its only predecessor, $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}$ are already defined in the environment, and assignments do not modify their value. Now, if pc is its only predecessor, then $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}}$ are not yet defined in the environment: the semantics of the program is stuck. However, the only case where this instruction is reacheable is when it is the first instruction of the method, and in this case, the stack is empty, hence no assignments are preprended to the BIR. Hence, the stack equivalence still holds.
- If $\mathtt{pc} \notin jmpTgt^{\mathrm{P}}_{\mathtt{m}}$, we distinguish two cases:
  - If $succ_{\mathtt{m}}(\mathtt{pc}) \cap jmpTgt^{\mathrm{P}}_{\mathtt{m}} \neq \emptyset$, then $\mathtt{as_{in}} = newStack(\mathtt{pc}, stackSize(\mathtt{pc}))$. The stack equivalence has to be checked in the environment modified by $Assignts(\mathtt{pc}, \mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}])$ and $paraAssignts(succ_{\mathtt{m}}(\mathtt{pc}), \mathtt{AS_{out}}[\mathtt{m},\mathtt{pc}])$.

    First, none of the assigned $\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}}$ are used in $\mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}]$: they are put onto the abstract stack only at point pc and the stack is normalised with a different naming convention on backward branches. Hence, stacks keep related with regards to the environment $lt[\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}} \mapsto v_k]$, where v_k is the value of the k^{th} element of $\mathtt{AS_{in}}[\mathtt{m},\mathtt{pc}]$.

Now, assignments generated by $paraAssignts(succ_{\mathtt{m}}(\mathtt{pc}), \mathtt{AS}_{\mathtt{out}}[m, pc])$ modify $lt[\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}} \mapsto v_k]$ but without changing the stack equivalence: all assigned $\mathtt{T}^{\mathtt{j}}_{\mathtt{pcb}}$ (where $\mathtt{pcb} \in succ_{\mathtt{m}}(\mathtt{pc})$ is a branching point) have different identifiers from the $\tilde{\mathtt{T}}^{\mathtt{k}}_{\mathtt{pc}}$.

– Otherwise, $\mathtt{as}_{\mathtt{in}}$ is set to $\mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}]$. But no assignment is preprend to the BIR. Hence, the environment is not modified and stacks are equivalent.

Thus $s \approx_{ht,\widetilde{lt},\beta} \mathtt{as}_{\mathtt{in}}$, where $\widetilde{lt}$ is equal to lt that has been potentially modified by assignments preprended to the BIR. In addition, the heap ht is not modified. The hypotheses of Proposition 1 are thus satisfied, and we obtain that:

$$\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda_1}}{\Rightarrow}_0 \langle ht, m, (pc, \mathtt{instrs}), \widetilde{lt}\rangle \overset{\vec{\lambda_2}}{\Rightarrow}_0 \langle ht', m, (pc', IR[m, pc']), lt'\rangle$$

where the intermediate state $\langle ht, m, (pc, \mathtt{instrs}), \widetilde{lt}\rangle$ is obtained by executing potential additional assignments. By Proposition 1, we have that resulting heaps and environments are equivalent w.r.t. β, and $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda_2}_{proj}$. Furthermore, $\vec{\lambda_1}$ is only made of temporary variable assignment events, hence $\vec{\lambda_1}_{proj}$ is empty, and $\vec{\lambda} \overset{!}{\sim}_\beta (\vec{\lambda_1}.\vec{\lambda_2})_{proj}$.

We conclude the proof by showing the transmitted abstract stack is equivalent to s' with regards to ht', lt' and β. Here again, there are two cases:

- If $\mathtt{pc}'$ is not a branching point, then the transmitted abstract stack is $\mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}]$, resulting from the basic transformation $BC2BIR_i$. The stack equivalence is here simply given by Proposition 1.
- If $\mathtt{pc}' \in jmpTgt^{\mathtt{P}}_{\mathtt{m}}$, the transmitted abstract stack is $newStackJmp(\mathtt{pc}', stackSize(\mathtt{pc}'))$. All of the $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}'}$ have been assigned, but we must show that they have not been modified since then by the BIR instructions generated by $BC2BIR_i$. An environment can be modified by BIR instructions that are either obtained by transforming an `store x` instruction, or instructions that could modify the value of $\mathtt{AS}_{\mathtt{out}}[\mathtt{m}, \mathtt{pc}]$ elements (see Figure 5.1 for variable or field assignment). In the first case, the variable is used at BC level and is hence different from all $\mathtt{T}^{\mathtt{j}}_{\mathtt{pc}'}$. In the second case, temporary variables are $\mathtt{t}^{\mathtt{k}}_{\mathtt{pc}}$ and have also different identifiers.

Thus, we have $s' \approx_{ht',lt',\beta} \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}']$. □

Proposition 4 (*BC2BIR* - zero call-depth one-step preservation - error case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\rightarrow}_0 \Omega^k_{pc'}$ *and* ht, lt, β *are s.t.* $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}]$.

Then there exists a unique $\vec{\lambda'}$ *such that* $\langle ht, m, (pc, \mathtt{IR}[\mathtt{m}, \mathtt{pc}]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \Omega^k_{pc'}$ *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$

Proof. Similar to Proposition 3, but using Proposition 2. □

Proposition 5 (*BC2BIR* - zero call-depth preservation - normal case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_0 \langle h', m, pc', l', s'\rangle$ *and* ht, lt, β *are such that* $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}]$.

Then there exist unique $ht', lt', \vec{\lambda'}$ *such that* $\langle ht, m, (pc, \mathtt{IR}[\mathtt{m}, \mathtt{pc}]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', m, (pc', \mathtt{IR}[\mathtt{m}, \mathtt{pc}']), lt'\rangle$ *with* $h' \overset{\mathrm{H}}{\sim}_\beta ht'$, $l' \overset{\mathrm{E}}{\sim}_\beta lt'$, $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$ *and* $s' \approx_{ht',lt',\beta} \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}']$.

Under the same hypotheses, if $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_0 \langle h', rv\rangle$ *and* $s \approx_{ht,lt,\beta} \mathtt{AS}_{\mathtt{in}}[\mathtt{m}, \mathtt{pc}]$ *then there exists a unique* $\langle ht', rv'\rangle$ *such that:* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \langle ht', rv'\rangle$, *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$, $h' \overset{\mathrm{H}}{\sim}_\beta ht'$ *and* $rv \overset{\mathrm{V}}{\sim}_\beta rv'$.

Proof. Similar to Proposition 3, using an induction on the number of steps of the BC computation. □

Proposition 6 (*BC2BIR* - zero call-depth preservation - error case)**.**

Suppose we have $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_0 \Omega^k_{pc'}$ *and* ht, lt, β *are s.t.* $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ *and* $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist unique $\vec{\lambda'}$ *such that* $\langle ht, m, (pc, \texttt{IR}[\texttt{m}, \texttt{pc}]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_0 \Omega^k_{pc'}$ *with* $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

Proof. As the error state is reached after a given number of normal execution steps, we use here Propositions 5 and 2. □

We now have to state propositions similar to Propositions 5 and 6, dealing with an arbitrary call-depth.

Proposition 7 (*BC2BIR* - multi-step preservation - normal case)**.**

Let $n \in \mathbb{N}$. *Suppose that* $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h', m, pc', l', s'\rangle$ *and* ht, lt, β *are such that* $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist unique $ht', lt', \vec{\lambda'}$ *and a unique* β' *extending* β *such that* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht', m, (pc', IR[m, pc']), lt'\rangle$ *with* $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_{\beta'} ht'$, $l' \overset{\text{E}}{\sim}_{\beta'} lt'$ *and* $s' \approx_{ht',lt',\beta'} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}']$.

Proposition 8 (*BC2BIR* - multi-step preservation - error case)**.**

Let $n \in \mathbb{N}$. *Suppose that* $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \Omega^k_{pc'}$ *and* ht, lt, β *are such that* $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist a unique $\vec{\lambda'}$ *and a unique* β' *extending* β *such that* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \Omega^k_{pc'}$ *with* $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$

The proof of Propositions 7 and 8 will be done by strong induction on the call-depth. For the sake of clarity, let $\mathcal{P}(n, m)$ and $\mathcal{P}_\Omega(n, m)$ denote respectively Propositions 7 and 8, where n is the call depth and m denotes the method that is being executed. Here, Propositions 5 and 6 are respectively the base cases $\mathcal{P}(0, m)$ and $\mathcal{P}_\Omega(0, m)$. Concerning induction cases, we use an induction on the number of steps of the BC computation. The base cases are shown using Proposition 9 and 10.

Proposition 9 (*BC2BIR* - one-step preservation - normal case)**.**

Let $n \in \mathbb{N}$. *Suppose that* $\mathcal{P}(k, m)$ *for all* m *and* $k < n$. *Suppose that* $\langle h, m, pc, l, s\rangle \overset{\vec{\lambda}}{\rightarrow}_n \langle h', m, pc', l', s'\rangle$. *Let* ht, lt, β *be such that* $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Then there exist unique ht', lt' *and a unique* β' *extending* β *such that* $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht', m, (pc', IR[m, pc]), lt'\rangle$ *with* $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$, $h' \overset{\text{H}}{\sim}_{\beta'} ht'$, $l' \overset{\text{E}}{\sim}_{\beta'} lt'$ *and* $s' \approx_{ht',lt',\beta'} \texttt{AS}_{\texttt{in}}[\texttt{m}, \texttt{pc}]$.

Proof. Here we use the same proof structure than for Proposition 3. Arguments are the same for showing that the stack equivalence between s and $\texttt{as}_{\texttt{in}}$ is preserved by the potential additional assignments. Hence, we have $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda_1}}{\Rightarrow}_0 \langle ht, m, (pc, instrs), \widetilde{lt}\rangle$ with $\widetilde{lt}$ is the new environment, $l \overset{\text{E}}{\sim}_\beta \widetilde{lt}$ and $s \approx_{ht,\widetilde{lt},\beta} \texttt{as}_{\texttt{in}}$. The instruction list `instrs` is obtained by the basic transformation $BC2BIR_i$. We now have to match the BC execution step. We proceed by case analysis on the BC instruction $instrAt_P(\texttt{m}, \texttt{pc})$.

- **`constructor C`** Here, $s = V :: (\mathit{Ref}\ r) :: s'$. By case analysis on the semantic rule that is used, we have four cases. In the first possible rule, we have $h(r) = o_t$ with $t = \widetilde{\mathtt{C}_\mathtt{j}}$ where $C \neq \mathit{Object}$. Let h_1 be the heap such that $h_1 = h[r \mapsto \mathit{upInit}(o, \widetilde{\mathtt{C}})]$. We have that

$$\langle h_1, C.\mathit{init}, 1, [\mathtt{this} \mapsto (\mathit{Ref}\ r), \mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_\mathtt{n} \mapsto v_n], \varepsilon\rangle \overset{\vec{\lambda_2}}{\Rightarrow}_{n-1} \langle h_2', \mathit{Void}\rangle$$

  Only one form of abstract stack is compatible with the transformation to succeed (see Figure 5.1): only one symbolic expression can be evaluated to a value equivalent to $(\mathit{Ref}\ r)$ (β is not defined on r).

  Whenever the abstract stack is not simply transmitted to `pc` from its direct predecessor, then the abstract stack might be $\mathit{newStackJmp}(\mathtt{pc}, \mathit{stackSize}(\mathtt{pc}))$ or $\mathit{newStack}(\mathtt{pc}, \mathit{stackSize}(\mathtt{pc}))$. But both functions preserve the $\mathit{ExprUninit}(\mathtt{C}, \mathtt{j})$. Moreover, by hypothesis on the bytecode, we know that all $\mathtt{AS}_\mathtt{out}[\mathtt{m}, \mathtt{pc}']$ contain the same $\mathit{ExprUninit}(\mathtt{C}, \mathtt{j})$ at the same places, where `pc'` is a predecessor. Thus, $\mathtt{as}_\mathtt{in}$ is of the form: $e_1 :: \ldots :: e_n :: \mathit{ExprUninit}(\mathtt{C}, \mathtt{j}) :: as$ and $v_1 :: \ldots :: v_n :: (\mathit{Ref}\ r) :: s \approx_{ht,\widetilde{lt},\beta} \mathtt{as}_\mathtt{in}$.

  We have $\mathit{BC2BIR}_i(pc, \mathtt{constructor}\ \mathtt{C}, e_1 :: \ldots :: e_n :: \mathit{ExprUninit}(\mathtt{C}, \mathtt{j}) :: as) = ([\mathtt{t}^1_\mathtt{pc} := e_1'; \ldots; \mathtt{t}^\mathtt{m}_\mathtt{pc} := e_m'; \mathtt{t}^\mathtt{0}_\mathtt{pc} := \mathit{new}\ C(e_1, \ldots, e_n)], as[\mathtt{t}^\mathtt{i}_\mathtt{pc}/\ \mathtt{e}_\mathtt{i}][\mathtt{t}^\mathtt{0}_\mathtt{pc}/\mathit{ExprUninit}(\mathtt{C}, \mathtt{j})])$

  Let us follow the semantics of BIR. Let $(ht_1, r') = \mathit{newObject}(C, ht)$ and $ht_1' = ht_1[r' \mapsto \mathit{upInit}(o, \widetilde{\mathtt{C}})$. We hence have $ht_1'(r') = \mathit{Blank}(\widetilde{\mathtt{C}})$. By hypothesis, stacks are equivalent. Thus, for all i, $ht_1', \widetilde{lt} \vDash e_i \Downarrow v_i'$ and $v_i \overset{\mathrm{v}}{\sim}_\beta v_i'$.

  We extend β to β' to take r' into account: $\beta'(r) = r'$, and we have that $h_1 \overset{\mathrm{H}}{\sim}_{\beta'} ht_1'$: objects pointed to by r and r' have the same initialization status, their class is equal, and each of their field has default values (we have $h_1(r) = \mathit{Blank}(\widetilde{\mathtt{C}})$, and before the call to the constructor, nothing can be done on this object). Hence, both constructors are called on equivalent initial configurations. We can now apply $\mathcal{P}(n-1, C.\mathit{init})$. Hence, we get that there exists β'' extension of β' relating the two resulting heaps and constructor execution traces.

  Now, from m point of view, the traces are equivalent: $r \leftarrow C.\mathit{init}(v_1, \ldots, v_n) \overset{!}{\sim}_{\beta''} r' \leftarrow C.\mathit{init}(v_1', \ldots, v_n')$ and the remainder of both traces are equivalent by $\mathcal{P}(n-1, C.\mathit{init})$. The heaps have been shown to be equivalent w.r.t β'', and the environments keep related (only fresh temporary variables have been introduced). The stacks keep equivalent, since `t` is fresh, and is now evaluated to r', which is an equivalent value to r w.r.t β''.

  For other rules, the proof is similar (for the last one, references are already related through the bijection, which does not need to be extended).

- **`invokevirtual m`** We proceed similarely. The current objects are already initialized, hence the bijection is not extended. We can distinguish between void and value-returning methods: this information is available in the bytecode program, and we use the equivalence over the last label of method execution trace given by $\mathcal{P}(n-1, mc)$ to deduce that the BIR method has the same signature.

Similarely to Proposition 3, we conclude by showing the transmitted abstract stack $\mathtt{AS}_\mathtt{in}[\mathtt{m}, \mathtt{pc}']$ is equivalent to s'. Here again, arguments are the same. □

Proof of Proposition 7. We proceed by strong induction on n.

- As already said, $\mathcal{P}(0, m)$ is exactly Proposition 5.

- Now suppose that $\forall m$ and $k < n, \mathcal{P}(k, m)$. We show $\mathcal{P}(n, m)$ by induction on np, the number of steps of the BC computation.
  - If $np = 1$, then we use Proposition 9.
  - Suppose now that $np > 1$ and $\mathcal{P}(n, m)$ holds for all np'-step computations with $np' < np$. Thus, by the definition of call-depth indices, we have that

$$\langle h, m, pc, l, s\rangle \underset{(1)}{\overset{\vec{\lambda_1}}{\Rightarrow_{n_1}}} \langle h_1, m, pc_1, l_1, s_1\rangle \underset{(2)}{\overset{\vec{\lambda_2}}{\longrightarrow_{n_2}}} \langle h_2, m, pc_2, l_2, s_2\rangle$$

    with $n_1 + n_2 = n$. Step (1) of the computation is made of $np - 1$ steps, and step (2) is a one-step transition. In both cases, we can use the induction hypothesis (respectively on $np - 1$ and 1) to get the result.

□

Proposition 10 (*BC2BIR* - one-step preservation - error case)**.**

Let $n \in \mathbb{N}$. Suppose that $\mathcal{P}_\Omega(k, m)$ for all m and $k < n$. Suppose that $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_n \Omega^k_{pc'}$. Let ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$, $s \approx_{ht,lt,\beta} \mathtt{AS_{in}[m, pc]}$.

Then there exist a unique β' extending β such that $\langle ht, m, (pc, IR[m, pc]), lt\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \Omega^k_{pc'}$ with $\vec{\lambda} \overset{!}{\sim}_{\beta'} \vec{\lambda'}_{proj}$.

Proof. Here again the structure of the proof follows the one of Proposition 3.

Suppose that $\langle h, m, pc, l, s\rangle \xrightarrow{\vec{\lambda}}_n \Omega^k_{pc'}$. By similar arguments than before, we get that $s \approx_{ht,\widetilde{lt},\beta} \mathtt{as_{in}}$.

To match the BC computation step, we now proceed by case analysis on the BC instruction at program point `pc`.

- **`constructor C`** Here, four error computation steps are possible, according to the initialization status of the object pointed to by the reference on which the constructor is called.

  In the first two cases, the initialization tag of the object is $\widetilde{\mathtt{C_j}}$. A similar reasoning than in the proof of Proposition 9 can be done to deduce the form of the abstract stack $\mathtt{as_{in}}$ and to state that initial configurations on which the constructor execution starts are equivalent (the bijection is extended to β' to take into account the newly allocated object in the BIR heap).

  Now, the execution of the BC constructor fails in the state $\Omega^k_{pc'}$. We use here proposition $\mathcal{P}_\Omega(n-1, \mathtt{C.init})$ to obtain that the BIR constructor execution fails too in $\Omega^k_{pc'}$, and their traces are equivalent with regards to β'.

  Traces equivalence holds also from the point of view of the method `m` (the projection of traces preserves their equivalence). Finally, error states are equal.

  In both other cases, the reference pointing to the object on which the constructor is called is already tagged as being initialized. Here, the bijection does not need to be extended. Second, the τ event at the head of the BC trace is matched by the normal execution of the assertion generated: the initialization of the object is ongoing, thus the reference pointing to it is in the domain of β, and its equivalent value is also a non-null reference. The rest of the proof is similar.

- **`invokevirtual`** Here, the reference pointing to the object on which the method is called is initialized. Hence the bijection does not need to be extended. Arguments are similar to the previous case.

□

Proof of Proposition 8. We proceed by strong induction on n.

- As already said, $\mathcal{P}_\Omega(0, m)$ is exactly Proposition 6.
- Now suppose that $\forall m$ and $k < n, \mathcal{P}_\Omega(k, m)$. We show $\mathcal{P}_\Omega(n, m)$ by case analysis on np, the number of steps of the BC computation.
  - If $np = 1$, then we use Proposition 10.
  - Suppose now that $np > 1$. Thus, by the definition of call-depth indices, we have that

$$\langle h, m, pc, l, s\rangle \underset{(1)}{\overset{\vec{\lambda_1}}{\Rightarrow}_{n_1}} \langle h_1, m, pc_1, l_1, s_1\rangle \underset{(2)}{\overset{\vec{\lambda_2}}{\longrightarrow}_{n_2}} \langle h_2, m, pc_2, l_2, s_2\rangle$$

    with $n_1 + n_2 = n$. For the step (1) of this computation, we apply Proposition 7. Then, we are back in the base case for step (2), and we use Proposition 10.

□

From this proposition, we can derive the final theorem of semantics preservation. For the sake of simplicity, we only consider the main method of the program. We denote by $BC2BIR(P)$ the mapping of $BC2BIR(P, m)$ on all the methods m of P.

Theorem 1 (Semantics preservation - normal case)**.**
Let P be a BC program and P′ be its BIR version P′ = BC2BIR(P). Let h_0 denote the empty heap and l_0 an arbitrary environment. Let $\langle h, \texttt{main}, pc, l, s\rangle$ be a reachable state of P, i.e.

$$\langle h_0, \texttt{main}, 1, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h, \texttt{main}, pc, l, s\rangle$$

Then, there exist a heap ht, an environment lt and partial bijection β such that

$$\langle h_0, \texttt{main}, (1, instrsAt_{P'}(\texttt{main}, 1)), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht, \texttt{main}, (pc, instrsAt_{P'}(\texttt{main}, \texttt{pc})), lt\rangle$$

with $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

Under the same hypotheses, if $\langle h_0, \texttt{main}, 1, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle h, rv\rangle$ Then, there exist a heap ht and partial bijection β such that $\langle h_0, \texttt{main}, (1, instrsAt_{P'}(\texttt{main}, 1)), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \langle ht, rv'\rangle$ with $h \overset{\text{H}}{\sim}_\beta ht$, $rv \overset{\text{V}}{\sim}_\beta rv'$ and $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

A similar theorem is obtained about executions leading to an error state.

Theorem 2 (Semantics preservation - normal case)**.**
Let P be a BC program and P′ be its BIR version P′ = BC2BIR(P). Let h_0 denote the empty heap and l_0 an arbitrary environment. If $\langle h_0, \texttt{main}, 1, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \Omega^k_{pc'}$, then there exist a partial bijection β such that $\langle h_0, \texttt{main}, (1, instrsAt_{P'}(\texttt{main}, 1)), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n \Omega^k_{pc'}$ with $\vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$.

Proof. We use Proposition 7. Starting states fullfill the hypotheses since the heaps are empty, the environments are equal and the translation algorithm begins on an empty abstract stack. □

We presented the translation algorithm BC2BIR and proved it preserves the semantics of the initial BC program: the BIR program $BC2BIR(P)$ simulates the program P, and their execution trace are equivalent. This result gives us some guarantees about the IR of a given program, and brings the hope that static analyses results obtained about the BIR program could be shifted back to the initial BC program. In the next section we make a first step towards this "safeness preservation", through two simple examples.

5.3 Application example

In this section, we aim at demonstrating how the result of a static analysis on a BIR program can be translated back to the initial BC program. We illustrate this on three examples of safety property.

Null-pointer error safety In [HJP08], Hubert *et al* propose a null-pointer analysis on a subset of Java source and show it correct. Their analysis is based on abstract interpretation and infers, for each field of each class of the program, whether the field is definitely non-null or possibly null after object initialization. Adapting their definition for BC we obtain the following safety property:

Definition 9 (BC Null-Pointer error safety)**.** *A BC program is said to be null pointer error safe if, for all pc', $\langle h_0, \texttt{main}, 1, l_0, \varepsilon\rangle \vec{\lambda}\Rightarrow_n s$ implies $s \neq \Omega^{NP}_{pc'}$ where h_0 is the empty heap and l_0 is the empty environment (the main method is assumed to have no parameters).*

Hubert later proposed a Bytecode version for the analysis in [Hub08]. It uses expression reconstruction to improve the accuracy of the analysis. Additionnaly, as it deals with object initialization, this analysis definitely needs to reconstruct the link between freshly allocated reference in the heap and the call of the constructor on it. The BIR language provides this information, and this would have eased the analysis. The safety property shifted to BIR is defined as follows:

Definition 10 (BIR Null-Pointer error safety)**.** *A BIR program P is said to be null pointer error safe if, for all pc', $\langle h_0, \texttt{main}, (1, instrsAt_P(\texttt{main}, 1)), l_0\rangle \vec{\lambda}\Rightarrow_n s$ implies $s \neq \Omega^{NP}_{pc'}$ where h_0 is the empty heap and l_0 is the empty environment.*

Suppose now given a correct BIR analysis, with regards to the Definition 10. As a direct consequence of Theorem 1, we can show that if a program is deemed safe by the BIR analysis, then the initial BC program is also null-pointer error safe.

Bounded field value Suppose we want to ensure that in a given program, the integer field `f` of each object of class `C` always has a value within the interval $[0, 10]$. Interval analyses are the typical solution for this problem: they determine at each program point an interval in which variables and object fields take their values. Then, the analysis would check that at every interest program points, i.e. `f` fields assignments, the value given to the field is in an appropriate interval. In Section **??**, we saw that the precision of such an analysis is increased with the help of symbolic expressions. Hence, it would be easier to perform this analysis on the BIR version of the program. Here we prove that if the program $BC2BIR(P)$ is shown to satisfy this safety property (by mean of a correct static analysis), then the initial BC program P is also safe. The proof is performed at a rather intuitive level of details. Further formalization on this is ongoing work.

The problem can be formulated this way. Let $\mathcal{L}$ be one of our two languages BC or BIR. A program p is safe if and only if $[\![p]\!]_{\mathcal{L}} \subseteq Safe_{\mathcal{L}}$, where $[\![p]\!]_{\mathcal{L}}$ is the set of all reachable states of p (as defined in Theorem 1) and $Safe_{\mathcal{L}}$ is the set of all states in $State_{\mathcal{L}}$ that are safe:

Definition 11 (Safe states)**.**
Let p be a $\mathcal{L}$ program. A state $s \in State_{\mathcal{L}}$ is in $Safe_{\mathcal{L}}$ if:

- $\langle h_0, main_p, 1, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n s$, *where h_0 is empty, and l_0 is an arbitrary environment,*
- $\vec{\lambda} = \lambda_1\lambda_2\ldots\lambda_n$ *and* $\forall i, Safe_E(\lambda_i)$

where $Safe_E(\lambda) \Leftrightarrow \lambda \notin \{r.\mathtt{f} \leftarrow (Num\ n) \mid r \in Reference, n \in [-\infty; -1] \cup [11; +\infty]\}$

Note that safe states could here include error states Ω^k_{pc}, as the safety property only deals with field assignments. A static analysis can be seen as a function $prog_{\mathcal{L}} \rightarrow \{fail, ok\}$, that takes a program written in $\mathcal{L}$ as argument, and returns $fail$ or ok depending on whether an invalid value can be assigned to an `f` field. A given analysis *Analyse* is said to be correct if the following holds:

$$\forall p \in prog_{\mathcal{L}}, Analyse(p) = ok \Rightarrow [\![p]\!]_{\mathcal{L}} \subseteq Safe_{\mathcal{L}}$$

Now, suppose we are given a correct analysis on BIR, $Analyse_{BIR}$. Let P be a BC program and $P' = BC2BIR(P)$ its BIR version. Suppose that $Analyse_{BIR}(P') = ok$, and hence that the BIR program is safe. We show that P is also safe. Let s be a reachable state of P, i.e. $\langle h_0, main_p, 1, l_0, \varepsilon\rangle \overset{\vec{\lambda}}{\Rightarrow}_n s$ with h_0 the empty heap, l_0 an arbitrary environment, and $\vec{\lambda} = \lambda_1\lambda_2\ldots\lambda_n$. Applying Theorem **??**, we get that there exist a state s', partial bijection β and a trace $\vec{\lambda'}$ such that

$$\langle h_0, main_{P'}, (1,0), l_0\rangle \overset{\vec{\lambda'}}{\Rightarrow}_n s' \text{ with } \vec{\lambda} \overset{!}{\sim}_\beta \vec{\lambda'}_{proj}$$

But P' is safe, hence for all $i \in [1; n]$, we have that $Safe_E(\lambda'_i)$ where $\vec{\lambda'}_{proj} = \lambda'_1\ldots\lambda'_n$. Take j and n such that $\lambda_j = r_j.\mathtt{f} \leftarrow (Num\ n_j)$. By the definition of $\overset{!}{\sim}_\beta$, we have that $\lambda'_j = r'_j.\mathtt{f} \leftarrow (Num\ n_j)$. We know that $Safe_E(\lambda'_j)$, thus $n \in [0; 10]$ and $Safe_E(\lambda_j)$. Hence, P is safe.

File access safety Our claim is that the correctness propositions of the last section suits well safety properties expressed as FSM languages. To illustrate this, we take the example of checking whether a given program correctly accesses a given file: every writing to it is performed after having it opened, and before closing it. Here again, an analysis on the BIR version of the program appears to be more easy to performed, thanks to the method call folding – the content of the stack does not need to be analysed anymore.

Let $\mathcal{A} = (\Sigma, Q, \delta, \mathcal{I}, \mathcal{F})$ be a FSM, as is standardly defined. Transitions in δ relates states of Q and are labelled with elements of the alphabet Σ. Initial and final states are respectively in subsets $\mathcal{I}$ and $\mathcal{F}$ of Q. The safety property we are interested in can be expressed as a language $\mathcal{L}(\mathcal{A})$ of the FSM $\mathcal{A}$ given in Figure 5.7, where the entry word is extracted from the execution trace of the program. More formally,

Definition 12 (File access safety)**.**
During the execution of the BIR program P, the object file pointed to in the heap by the reference (Ref r) is accessed safely if, whenever $\langle h_0, \mathtt{main}, (1, instrsAt_P(\mathtt{main}, 1)), l_0\rangle \overset{\vec{\lambda}}{\Rightarrow}_n \langle ht, Void\rangle$ *then* $ToSigma(\vec{\lambda}_{r\ file}) \in \mathcal{L}(\mathcal{A})$

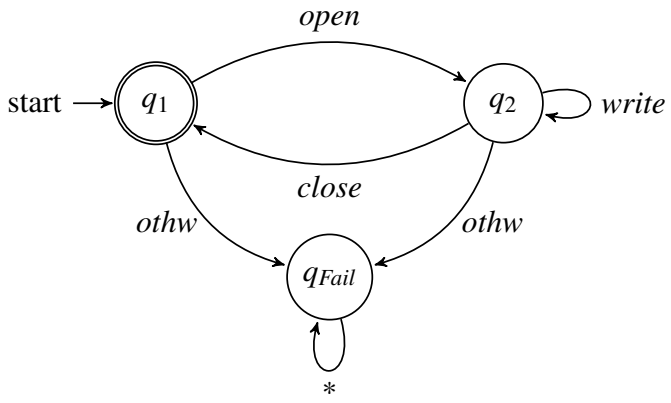


Figure 5.7: The FSM $\mathcal{A}$ accepting only safe file accesses

where the projection $\vec{\lambda}_{r\ file}$ is

$$\vec{\lambda} \text{ restricted to events in } \{r.File.open(), r.File.write(v_1, \ldots, v_n), r.File.close()\}$$

and the function *ToSigma* is defined as:

$$ToSigma(\lambda_1.\vec{\lambda}) = \begin{cases} open.ToSigma(\vec{\lambda}) & \text{if } \lambda_1 = [r.File.open()] \\ write.ToSigma(\vec{\lambda}) & \text{if } \lambda_1 = [r.File.write(v_1, \ldots, v_n)] \\ close.ToSigma(\vec{\lambda}) & \text{if } \lambda_1 = [r.File.close()] \end{cases}$$

It follows from Theorem 1 that if a BIR program execution trace $\vec{\lambda}$ is safe, then the initial BC program executes producing a trace $\vec{\lambda'}$ and that $ToSigma(\vec{\lambda'}_{\beta^{-1}(r)\ file}) \in \mathcal{L}(\mathcal{A})$ (events of trace $\vec{\lambda'}$ have been filtered to file accesses to the object pointed to by the corresponding reference in the BC heap).

In this section, we demonstrate on an example how the semantics preservation property of BC2BIR algorithm could help shifting the results of a given analysis on a BIR program back to the initial BC program. Our claim is that a similar reasoning could be applied to many other analyses. We also believe that there exist analyses for which nothing can be said about the initial BC program, given its result on the BIR version of the program. A direct example would be an analysis that deals with the allocation history: the bijection β is never made explicit, we only ensure its existence. Investigating this intuition, and further formalizing this "safety shifting" is left as future work.

So far, we formalized both source and target languages of the transformation. In this chapter, we formalized the algorithm *BC2BIR*. This algorithm, more precisely some variants of it, already exist in the literature (see e.g [CFM+97],[XX99], or [WCN05]). Our contribution here is the proof of its correctness: the semantics of the initial BC program is preserved by the transformation. The proof argument is based on a commutative diagram (also known as the simulation argument), a rather classical technique to prove the correctness of program transformations. The notion of simulation is defined relatively to a partial bijection that relates both heaps throughout the execution of P and $BC2BIR(P)$. BC is a subset of the Java bytecode language. It is realistic in the sense that it includes object oriented features and method calls, while it is simple enough to carry the proof with pen and paper. Scaling the source language up to real-world bytecode requires other tools and techniques: mechanized proofs. This is subject of the next chapter.

Chapter 6

Conclusions

As noticed by Logozzo and Fähndrich [LF08], static analysis of bytecode programs is made specially difficult because of their intensive use of the operand stack. This paper provides a semantically sound, provably correct specification of a transformation of byte code into an intermediate representations (IR) of bytecode that i) removes the use of the operand stack and rebuilds tree expressions, ii) makes more explicit the throwing of exception and takes care of preserving their order, iii) rebuild the initialisation chain of an object with a dedicated instruction `x := new C(arg1,arg2,...)`.

Further extensions may be twofold. First we would like to extend this work into a multi-threading context. This is a challenging task because symbolic expressions may be invalidated by concurrent accesses. The second extension concerns mechanization of the development. We believe the current transformation would be a valuable layer on top of the formal JVM semantics Bicolano that have been developed during the European MOBIUS project. We would like to use the Coq extraction mechanism to extract certified and efficient Caml code for the algorithm from a Coq formalisation of the algorithm.

Bibliography

[AAG+07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *Proc. of ESOP'07*, volume 4421, pages 157–172. Springer-Verlag, 2007.

[BCF+99] M G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño dynamic optimizing compiler for java. In *Proc. of JAVA '99*, pages 129–141. ACM, 1999.

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3):273–291, 2006.

[BN05] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[BR05] G. Barthe and T. Rezk. Non-interference for a jvm-like language. In *Proc. of TLDI '05*, pages 103–112, New York, NY, USA, 2005. ACM.

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[FM99] Stephen N. Freund and John C. Mitchell. The type system for object initializatiion in the jave bytecode language. *ACM Trans. Program. Lang. Syst.*, 21(6):1196–1250, 1999.

[FM03] S. N. Freund and J. C. Mitchell. A type system for the java bytecode language and verifier. *J. Autom. Reason.*, 30(3-4):271–321, 2003.

[GHM00] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proc. of SAS'00*, pages 199–219. Springer-Verlag, 2000.

[HJP08] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of FMOODS 2008*, volume 5051 of *LNCS*, pages 132–149. Springer Berlin, June 2008.

[Hub08] Laurent Hubert. A Non-Null annotation inferencer for Java bytecode. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'08)*. ACM, November 2008.

[LF08] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Proc. of CC 2008*, pages 197–212. Springer LNCS 4959, 2008.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. of CASCON '99*. IBM Press, 1999.

[WCN05] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of BYTECODE 2005, Electronic Notes in Computer Science*, 2005.

[Wha99] J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[XX99] H. Xi and S. Xia. Towards array bound check elimination in java tm virtual machine language. In *Proc. of CASCON '99*, page 14. IBM Press, 1999.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399

Certification using the Mobius Base Logic

Lennart Beringer[1], Martin Hofmann[1], and Mariela Pavlova[2]

[1] Institut für Informatik, Universität München
Oettingenstrasse 67, 80538 München, Germany
[2] Trusted Labs, Sophia-Antipolis, France
{beringer|mhofmann}@tcs.ifi.lmu.de,Mariela.Pavlova@trusted-labs.fr

Abstract. This paper describes a core component of Mobius' Trusted Code Base, the Mobius base logic. This program logic facilitates the transmission of certificates that are generated using logic- and type-based techniques and is formally justified w.r.t. the Bicolano operational model of the JVM. The paper motivates major design decisions, presents core proof rules, describes an extension for verifying intensional code properties, and considers applications concerning security policies for resource consumption and resource access.

1 Introduction: Role of the logic in Mobius

The goal of the Mobius project consists of the development of proof-carrying code (PCC) technology for the certification of resource-related and information-security-related program properties [16]. According to the PCC paradigm, code consumers are invited to specify conditions ("policies") which they require transmitted code to satisfy before they are willing to execute such code. Providers of programs then complement their code with formal evidence demonstrating that the program adheres to such policies. Finally, the recipient validates that the obtained evidence ("certificate") indeed applies to the transmitted program and is appropriate for the policy in question before executing the code.

One of the cornerstones of a PCC architecture is the trusted computing base (TCB), i.e. the collection of notions and tools in whose correctness the recipient implicitly trusts. Typically, the TCB consists of a formal model of program execution, plus parsing and transformation programs that translate policies and certificates into statements over these program executions. The Mobius architecture applies a variant of the *foundational* PCC approach [2] where large extents of the TCB are represented in a theorem prover, for the following reasons.

- Formalising a (e.g. operational) semantics of transmitted programs in a theorem prover provides a precise definition of the model of program execution, making explicit the underlying assumptions regarding arithmetic and logic
- The meaning of policies may be made precise by giving formal interpretations in terms of the operational model
- Theorem provers offer various means to define formal notions of certificates, ranging from proof scripts formulated in the user interface language (including tactics) of the theorem prover to terms in the prover's internal representation language for proofs (e.g. lambda-terms).

In particular, the third item allows one to employ a variety of certificate notions in a uniform framework, and to explore their suitability for different certificate generation techniques or families of policies. In contrast to earlier PCC systems which targeted mostly type- and memory-safety [27, 2], policies and specifications in Mobius are more expressive, ranging from (upper) bounds on resource consumption, via access regulations for external resources and security specifications limiting the flow of information to lightweight functional specifications [16]. Thus, the Mobius TCB is required to support program analysis frameworks such as type systems and abstract interpretation, but also logical reasoning techniques.

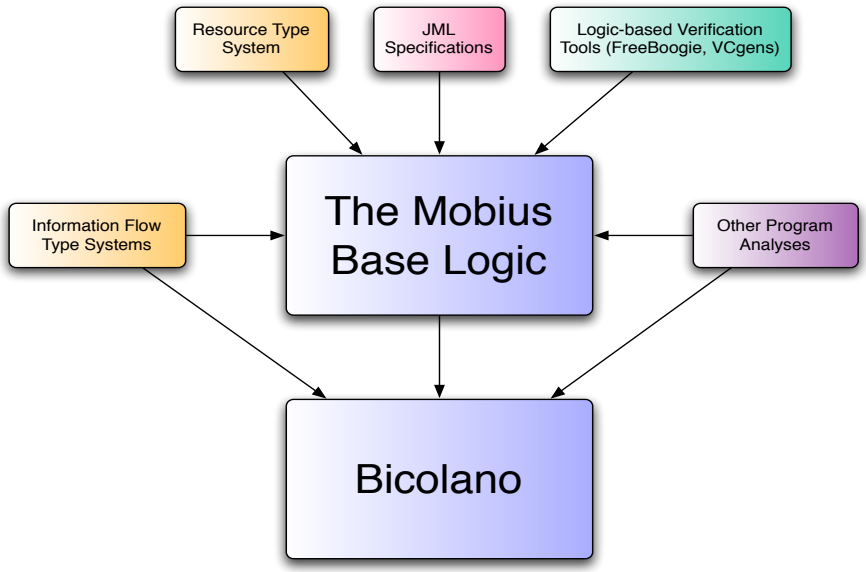


Fig. 1. Core components of the MOBIUS TCB

Figure 1 depicts the components of the Mobius TCB and their relations. The base of the TCB is formed by a formalised operational model of the Java Virtual Machine, Bicolano [30], which will be briefly described in the next section. Its purpose is to define the meaning of JVML programs unambiguously and to serve as the foundation on which the PCC framework is built. In order to abstract from inessential details, a program logic is defined on top of Bicolano. This provides support for commonly used verification patterns such as the verification of loops. Motivated by verification idioms used in higher-level formalisms such as type systems, the JML specification language, and verification condition generators, the logic complements partial-correctness style specifications by two further assertion forms: *local annotations* are attached to individual program points and

are guaranteed to hold whenever the annotated program point is visited during a program execution. *Strong invariants* assert that a particular property will continue to hold for all future states during the execution of a method, including states inside inner method invocations. The precise interpretation of these assertion forms, and a selection of proof rules will be described in Section 3.

We also present an extension of the program logic that supports reasoning about the effects of computations. The extended logic arises uniformly from a corresponding generic extension of the operational semantics. Using different instantiations of this framework one may obtain domain-specific logics for reasoning about access to external resources, trace properties, or the consumption of resources. Polices for such domains are difficult if not impossible to express purely in terms of relations between initial and final states. The extension is horizontal in the sense of Czarnik and Schubert [20] as it is conservative over the non-extended ("base") architecture.

The glue between the components is provided by the theorem prover Coq, i.e. many of the soundness proofs have been formalised. The encoding of the program logics follow the approach advocated by Kleymann and Nipkow [25, 29] by employing a shallow embedding of formulae. Assertions may thus be arbitrary Coq-definable predicates over states. Although the logic admits the encoding of a variety of program analyses and specification constructs, it should be noted that the architecture does not mandate that all analyses be justified with respect to this logic. Indeed, some type systems for information flow, for example, are most naturally expressed directly in terms of the operational semantics, as already the definition of information flow security is a statement over two program executions. In neither case do we need to construct proofs for concrete programs by hand which would be a daunting task in all but the simplest examples. Such proofs are always obtained from a successful run of a type system or program analysis by an automatic translation into the Mobius infrastructure. Examples of this method are given in Sections 4 and 5.2.

Outline We give a high-level summary of the operational model Bicolano [30], restricted to a subset of instructions relevant for the present paper, in Section 2. In Section 3 we present the program logic. Section 4 contains an example of a type-based verification and shows how a bytecode-level type system guaranteeing a constant upper bound on the number of heap allocations may be encoded in the logic. The extended program logic is outlined in Section 5, together with an application concerning a type system for numeric correspondence assertions [34]. We first discuss some related work.

1.1 Related work

The basic design decisions for the base logic were presented in [8], and the reader is referred to loc.cit. for a more in-depth motivation of the chosen format of assertions and rules. In that paper, we also presented a type-system for constant heap space consumption for a functional intermediate language, such that typing

derivations could be translated into program logic derivations over an appropriately restricted judgement form. In contrast, the type system given in the present paper works directly on bytecode and hence eliminates the language translation from the formalised TCB.

The first proposal for a program logic for bytecode we are aware of is the one by Quigley [31]. In order to justify a rule for while loops, Quigley introduces various auxiliary notions for relating initial states to intermediate states of an execution sequence, and for relating states that behave similarly but apply to different classes. Bannwart and Müller [4] present a logic where assertions apply at intermediate states and are interpreted as preconditions of the assertions decorating the successor instructions. However, the occurrence of these local specifications in positive and negative positions in this interpretation precludes the possibility of introducing a rule of consequence. Indeed, our proposed rule format arose originally from an attempt to extend Bannwart and Müller's logic with a rule of consequence and machinery for allowing assertions to mention initial states. Strong invariants were introduced by the Key project [6] for reasoning about transactional safety of Java Card applications using dynamic logics [7].

Regarding formal encodings of type systems into program logics, Hähnle et al. [23], and Beringer and Hofmann [9] consider the task of representing information flow type systems in program logics, while the MRG project focused on a formalising a complex type system for input-dependent heap space usage [10].

Certified abstract interpretation [11] complements the type-based certificate generation route considered in the present paper. Similar to the relationship between Necula-Lee-style PCC [27] and foundational PCC by Appel et al. [2], certified abstract interpretation may be seen as a foundational counterpart to Albert et al.'s Abstraction-carrying code [1]. Bypassing the program logic, the approach chosen in [11] justifies the program analysis directly with respect to the operational semantics. A generic framework for certifying program analyses based on abstract interpretation is presented by Chang et al. [14]. The possibility to view abstract interpretation frameworks as inference engines for invariants and other assertions in program logics in general was already advocated in one of the classic papers by Cousot & Cousot in [18].

Nipkow et al.'s VeryPCC project [33] explores an alternative foundational approach by formally proving the soundness of verification condition generators. In particular, [32] presents generic soundness and completeness proofs for VCGens, together with an instantiation of the framework to a safety policy preventing arithmetic overflows. Generic PCC architectures have recently been developed by Necula et al. [15] and the FLINT group [22].

2 Bicolano

Syntax and States We consider an arbitrary but fixed bytecode program P that assigns to each method identifier M a method implementation mapping instruction labels l to instructions. We use the notation $M(l)$ to denote the instruction at program point l in M, and $init_M$, $suc_M(l)$, and par_M to denote the initial

label of M, the successor label of l in M, and the list of formal parameters of M, respectively. While the Bicolano formalisation supports the full sequential fragment of the JVML, this paper treats the simplified language given by the *basic* instructions

$$basic(M,l) \equiv M(l) \in \left\{ \begin{array}{l} \mathsf{load}\ x, \mathsf{store}\ x, \mathsf{dup}, \mathsf{pop}, \mathsf{push}\ z, \\ \mathsf{unop}\ \mathsf{u}, \mathsf{binop}\ o, \mathsf{new}\ c, \mathsf{athrow}, \\ \mathsf{getfield}\ c\ f, \mathsf{putfield}\ c\ f, \mathsf{getstatic}\ c\ f, \mathsf{putstatic}\ c\ f \end{array} \right\}$$

and additionally conditional and unconditional jumps $\mathsf{ifz}\ l$ and $\mathsf{goto}\ l$, static and virtual method invocations $\mathsf{invokestatic}\ M$ and $\mathsf{invokevirtual}\ M$, and $\mathsf{vreturn}$.

Values and states The domain $\mathcal{V}$ of values is ranged over by $v, w, \ldots$ and comprises constants (integers z and Null), and addresses $a, \ldots \in \mathcal{A}$. States are built from operand stacks, stores, and heaps

$$O \in \mathcal{O} = \mathcal{V}\ list \qquad S \in \mathcal{S} = \mathcal{X} \rightharpoonup_{fin} \mathcal{V} \qquad h \in \mathcal{H} = \mathcal{A} \rightharpoonup_{fin} \mathcal{C} \times (\mathcal{F} \rightharpoonup_{fin} \mathcal{V})$$

where $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ are the domains of variables, class names, and field names, respectively. In addition to local states comprising operand stacks, stores, and heaps,

$$s, r \in \Sigma = \mathcal{O} \times \mathcal{S} \times \mathcal{H},$$

we consider initial states Σ_0 and terminal states $\mathcal{T}$

$$s_0 \in \Sigma_0 = \mathcal{S} \times \mathcal{H} \qquad t \in \mathcal{T} ::= \mathit{NormState}(h, v) + \mathit{ExcnState}(h, a)$$

These capture states which occur at the beginning and the end of a frame's execution. Terminal states t are tagged according to whether the return value represents a pointer to an unhandled exception object (constructor $\mathit{ExcnState}(.,.)$) or an ordinary return value (constructor $\mathit{NormState}(.,.)$). For $s_0 = (S, h)$ we write $\mathit{state}(s_0) = ([\,], S, h)$ for the local state that extends s_0 with an empty operand stack. For $\mathit{par}_M = [x_1, \ldots, x_n]$ and $O = [v_1, \ldots, v_n]$ we write $\mathit{par}_M \mapsto O$ for $[x_i \mapsto v_i]_{i=1,\ldots,n}$. We write $\mathit{heap}(s)$ to access the heap component of a state s, and similarly for initial and terminal states. Finally, $\mathit{lv}(.)$ denotes the local variable component of a state and $\mathit{getClass}(h, a)$ extracts the dynamic class of the object at location a in heap h.

Operational judgements Bicolano defines a variety of small-step and big-step judgements, with compatibility proofs where appropriate. For the purpose of the present paper, the following simplified setup suffices[3] (cf. Figure 2):

Non-exceptional steps The judgement $\vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r$ describes the (non-exceptional) execution of a single instruction, where l' is the label of the next instruction (given by $\mathit{suc}_M(l)$ or jump targets). The rules are largely standard, so we only give a rule for the invocation of static methods, InvsNorm.

[3] The formalisation separates the small-step judgements for method invocations from the execution of basic instructions and jumps, and then defines a single recursive judgement combining the two. See [30] for the formal details.

Exceptional steps The judgement $\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a$ describes exceptional small steps where the execution of the instruction at program point M, l in state s results in the creation of a fresh exception object, located at address a in the heap h. In the case of method invocations, a single exceptional step is also observed by the callee if the invoked method raised an exception that could not be locally handled (cf. rule InvsExcn).

Small step judgements Non-exceptional and handled exceptional small steps are combined to the small step judgement $\vdash_M l, s \Rightarrow l', r$ using the two rules NormStep and ExcnStep. The reflexive transitive closure of this relation is denoted by $\vdash_M l, s \Rightarrow^* l', r$

Big-step judgements The judgement form $\vdash_M l, s \Downarrow t$ captures the execution of method M from the instruction at label l onwards, until the end of the method. This relation is defined by the three rules Comp, Vret and Uncaught.

Deep step judgements The judgement $\vdash_M l, s \Uparrow r$ is defined similarly to the big-step judgement, by the rules D-Refl, D-Trans D-Invs, and D-Uncaught. This judgement associates states across invocation boundaries, i.e. r may occur in a subframe of the method M. This is achieved by rule D-Invs which associates a call state of a (static) method with states reachable from the initial state of the callee. A similar rule for virtual methods is omitted from this presentation.

Small and big-step judgements are mutually recursive due to the occurrence of a big-step judgement in hypotheses of the rules for method invocations on the one hand and rule Comp on the other.

3 Base logic

This section outlines the non-resource-extended program logic.

3.1 Phrase-oriented assertions and judgements

The structure of assertions and judgements of the logic are governed by the requirement to enable the interpretation of type systems as well as the representation of core idioms of JML. High-level type systems typically associate types (in contexts) to program phrases. Compiling a well-formed program phrase into bytecode yields a code segment that is the postfix of a JVM method, i.e. all program points without control flow successors contain return instructions. Consequently, judgements in the logic associate assertions to a program label which represents the execution of the current method invocation from the current point (i.e. a state applicable at the program point) onwards. In case of method termination, a partial-correctness assertion (post-condition) applies that relates this current state to the return state. As the guarantee given by type soundness results often extends to infinite computations (e.g. type safety, i.e. absence of type errors), judgements furthermore include assertions that apply to non-terminating

$$\textsc{InvsNorm}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Downarrow NormState(k, v)}{\vdash_M l, (O@O', S, h) \Rightarrow_{\mathsf{norm}} suc_M(l), (v :: O', S, k)}$$

$$\textsc{InvsExcn}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Downarrow ExcnState(k, a)}{\vdash_M l, (O@O', S, h) \Rightarrow_{\mathsf{excn}} k, a}$$

$$\textsc{NormStep}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r}{\vdash_M l, s \Rightarrow l', r} \qquad \textsc{ExcnStep}\ \frac{\vdash_M l, (O, S, h) \Rightarrow_{\mathsf{excn}} k, a \quad getClass(k, a) = e \quad Handler(M, l, e) = l'}{\vdash_M l, (O, S, h) \Rightarrow l', ([a], S, k)}$$

$$\textsc{Comp}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Downarrow t}{\vdash_M l, s \Downarrow t} \qquad \textsc{Vret}\ \frac{M(l) = \mathsf{vreturn}}{\vdash_M l, (v :: O, S, h) \Downarrow NormState(h, v)}$$

$$\textsc{Uncaught}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \quad getClass(h, a) = e \quad Handler(M, l, e) = \emptyset}{\vdash_M l, s \Downarrow ExcnState(h, a)}$$

$$\textsc{D-Refl}\ \frac{}{\vdash_M l, s \Uparrow s} \qquad \textsc{D-Trans}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Uparrow s''}{\vdash_M l, s \Uparrow s''}$$

$$\textsc{D-Invs}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Uparrow s}{\vdash_M l, (O@O', S, h) \Uparrow s}$$

$$\textsc{D-Uncaught}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \quad getClass(h, a) = e \quad Handler(M, l, e) = \emptyset}{\vdash_M l, s \Uparrow ([a], \emptyset, h)}$$

Fig. 2. Bicolano: selected judgements and operational rules

computations. These *strong invariants* relate the state valid at the subject label to each future state in the current method invocation. This interpretation includes states in subframes, i.e. in method invocations that are triggered in the phrase represented by the subject label.

Infinite computations are also covered by the interpretation of local annotations in JML, i.e. assertions occurring at arbitrary program points which are to be satisfied whenever the program point is visited. The logic distinguishes these explicitly given annotation from strong invariants as the former ones are not necessarily present at all program points. A further specification idiom of JML that has a direct impact on the form of assertions is `\old` which refers to the initial state of a method invocation and may appear in post-conditions, local annotations, and strong invariants.

Formulae that are shared between postconditions, local annotations, and strong invariant, and additionally only concern the relationship between the subject state and the initial state of the method may be captured in pre-conditions.

Thus, the judgement of the logic are of the form $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ where M, l denotes a program point (composed of a method identifier and an instruc-

tion label), and the assertions forms are as follows, where $\mathcal{B}$ denotes the set of booleans.

Assertions $A \in \mathit{Assn} = \Sigma_0 \times \Sigma \to \mathcal{B}$ occur as preconditions A and local annotations Q, and relate the current state to the initial state of the current frame.

Postconditions $B \in \mathit{Post} = \Sigma_0 \times \Sigma \times \mathcal{T} \to \mathcal{B}$ relate the current state to the initial and final state of a (terminating) execution of the current frame.

Invariants $I \in \mathit{Inv} = \Sigma_0 \times \Sigma \times \Sigma \to \mathcal{B}$ relate the initial state of the current method, the current state, and any future state of the current frame or a subframe of it.

The component $\mathcal{G}$ of a judgement represents a proof context and is represented as an association of specification triples $(A, B, I) \in \mathit{Assn} \times \mathit{Post} \times \mathit{Inv}$ to program points.

The behaviour of methods is described using three assertion forms.

Method preconditions $R \in \mathit{MethPre} = \Sigma_0 \to \mathcal{B}$ are interpreted hypothetically, i.e. their satisfaction implies that of the method postconditions and invariants but is not directly enforced to hold at all invocation points.

Method postconditions $T \in \mathit{MethSpec} = \Sigma_0 \times \mathcal{T} \to \mathcal{B}$ constrain the behaviour of terminating method executions and thus relate only initial and final states.

Method invariants $\Phi \in \mathit{MethInv} = \Sigma_0 \times \Sigma \to \mathcal{B}$ constrain the behaviour of terminating and non-terminating method executions by relating the initial state of a method frame to any state that occurs during its execution.

A program specification is given by a method specification table $\mathcal{M}$ that associates to each method a method specification $\mathcal{S} = (R, T, \Phi)$, a proof context $\mathcal{G}$, and a table $\mathcal{Q}$ of local annotations $Q \in \mathit{Assn}$. From now on, let $\mathcal{M}$ denote some arbitrary but fixed specification table satisfying $\mathit{dom}\ \mathcal{M} = \mathit{dom}\ P$.

3.2 Assertion transformers

In order to notationally simplify the presentation of the proof rules, we define operators that relate assertions occurring in judgements of adjacent instructions. The following operators apply to the non-exceptional single-step execution of basic instructions.

$$\begin{aligned}
\mathsf{Pre}(M, l, l', A)(s_0, r) &= \exists\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \wedge A(s_0, s) \\
\mathsf{Post}(M, l, l', B)(s_0, r, t) &= \forall\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \to B(s_0, s, t) \\
\mathsf{Inv}(M, l, l', I)(s_0, r, t) &= \forall\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \to I(s_0, s, t)
\end{aligned}$$

These operators resemble WP-operators, but are separately defined for pre-conditions, post-conditions, and invariants.

Exceptional behaviour of basic instructions is captured by the operators

$$\begin{aligned}
\mathsf{Pre}^{\mathsf{excn}}(M,l,e,A)(s_0,r) &= \exists\, s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \wedge \mathit{getClass}(h,a) = e \wedge \\
&\qquad r = ([a], \mathit{lv}(s), h) \wedge A(s_0,s) \\
\mathsf{Post}^{\mathsf{excn}}(M,l,e,B)(s_0,r,t) &= \forall\, s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow \mathit{getClass}(h,a) = e \rightarrow \\
&\qquad r = ([a], \mathit{lv}(s), h) \rightarrow B(s_0,s,t) \\
\mathsf{Inv}^{\mathsf{excn}}(M,l,e,I)(s_0,r,t) &= \forall\, s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow \mathit{getClass}(h,a) = e\ \rightarrow \\
&\qquad r = ([a], \mathit{lv}(s), h) \rightarrow I(s_0,s,t)
\end{aligned}$$

In the case of method invocations, we replace the reference to the operational judgement by a reference to the method specifications, and include the construction and destruction of a frame. For example, the operators for non-exceptional execution of static methods are

$$\begin{aligned}
&\mathsf{Pre}_{\mathsf{sinv}}(R,T,A,[x_1,\ldots,x_n])(s_0,s) = \\
&\quad \exists\, O\ S\ h\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n, h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n, h), (k,v))) \wedge \\
&\qquad\qquad s = (v :: O, S, k) \wedge A(s_0, ([v_1,\ldots,v_n]@O, S, h)) \\
&\mathsf{Post}_{\mathsf{sinv}}(R,T,B,[x_1,\ldots,x_n])(s_0,r,t) = \\
&\quad \forall\, O\ S\ k\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n, h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n, h), (k,v)))\ \rightarrow \\
&\qquad\qquad r = (v :: O, S, k) \rightarrow B(s_0, ([v_1,\ldots,v_n]@O, S, h), t) \\
&\mathsf{Inv}_{\mathsf{sinv}}(R,T,I,[x_1,\ldots,x_n])(s_0,s,r) = \\
&\quad \forall\, O\ S\ k\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n, h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n, h), (k,v)))\ \rightarrow \\
&\qquad\qquad s = (v :: O, S, k) \rightarrow I(s_0, ([v_1,\ldots,v_n]@O, S, h), r)
\end{aligned}$$

The exceptional operators for static methods cover exceptions that are raised during the execution of the invoked method but not handled locally. Due to space limitations we omit the operators for exceptional (null-pointer exceptions w.r.t. the invoking object) and non-exceptional behaviour of virtual methods.

3.3 Selected proof rules

An addition to influencing the types of assertions, type systems also motivate the use of a certain form of judgements and proof rules. Indeed, one of the advantages of type systems is their compositionality i.e. the fact that statements regarding a program phrase are composed from the statements referring to the constituent phrases, as in the following typical proof rule for a language of expressions

$$\frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 + e_2 : \mathbf{int}}.$$

Transferring this scheme to bytecode leads to a rule format where hypothetical judgements refer to the control flow successors of the phrase in the judgement's conclusion. In addition to supporting syntax-directed reasoning, this orientation renders the explicit construction of a control flow graph unnecessary, as no control flow predecessor information is required to perform a proof.

Figure 3 presents selected proof rules. These are motivated as follows.

$$\textsc{Instr}\ \frac{\begin{array}{c} basic(M,l) \qquad SC_1 \qquad SC_2 \qquad l'' = suc_M(l) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l'',A)\}\, M,l'\, \{\mathsf{Post}(M,l,l'',B)\}\, (\mathsf{Inv}(M,l,l'',I)) \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \\ \mathcal{G} \vdash \{\mathsf{Pre}^{\mathsf{excn}}(M,l,e,A)\}\, M,l'\, \{\mathsf{Post}^{\mathsf{excn}}(M,l,e,B)\}\, (\mathsf{Inv}^{\mathsf{excn}}(M,l,e,I)) \\ \forall\, s_0\ s\ h\ a.\ (\forall\, e.\ getClass(h,a) = e\ \rightarrow Handler(M,l,e) = \emptyset)\ \rightarrow \\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow A(s_0,s) \rightarrow B(s_0,s,(h,a)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{Goto}\ \frac{\begin{array}{c} M(l) = \mathsf{Goto}\ l' \qquad SC_1 \qquad SC_2 \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l',A)\}\, M,l'\, \{\mathsf{Post}(M,l,l',B)\}\, (\mathsf{Inv}(M,l,l',I)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{If0}\ \frac{\begin{array}{c} M(l) = \mathsf{ifz}\ l' \qquad SC_1 \qquad SC_2 \qquad l'' = suc_M(l) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l',A)\}\, M,l'\, \{\mathsf{Post}(M,l,l',B)\}\, (\mathsf{Inv}(M,l,l',I)) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l'',A)\}\, M,suc_M(l)\, \{\mathsf{Post}(M,l,l'',B)\}\, (\mathsf{Inv}(M,l,l'',I)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{InvS}\ \frac{\begin{array}{c} M(l) = \mathsf{invokestatic}\ M' \quad \mathcal{M}(M') = (R,T,\Phi) \quad SC_1 \quad SC_2 \\ \forall\, s_0\ O\ S\ h\ O'\ r\ v_i.\ (R(par_{M'} \mapsto O,h) \rightarrow \Phi((par_{M'} \mapsto O,h),r))\ \rightarrow \\ A(s_0,(O@O',S,h)) \rightarrow I(s_0,(O@O',S,h),r) \\ A_1 = \mathsf{Pre}_{\mathsf{sinv}}(R,T,A,par_{M'}) \qquad B_1 = \mathsf{Post}_{\mathsf{sinv}}(R,T,B,par_{M'}) \\ \mathcal{G} \vdash \{A_1\}\, M,suc_M(l)\, \{B_1\}\, (\mathsf{Inv}_{\mathsf{sinv}}(R,T,I,par_{M'})) \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \\ \mathcal{G} \vdash \{\mathsf{Pre}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,A,e,par_{M'})\}\, M,l'\, \{\mathsf{Post}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,B,e,par_{M'})\} \\ \qquad (\mathsf{Inv}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,I,e,par_{M'})) \\ \forall\, s_0\ O\ S\ h\ O'\ k\ a.\ (R(par_{M'} \mapsto O,h) \rightarrow \Phi((par_{M'} \mapsto O,h),(k,a)))\ \rightarrow \\ (\forall\, e.\ getClass(k,a) = e\ \rightarrow Handler(M,l,e) = \emptyset)\ \rightarrow \\ A(s_0,(O@O',S,h)) \rightarrow B(s_0,(O@O',S,h),(k,a)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{Ret}\ \frac{\begin{array}{c} M(l) = \mathsf{vreturn} \qquad SC_1 \qquad SC_2 \\ \forall\, s_0\ v\ O\ S\ h.\ A(s_0,(v :: O,S,h)) \rightarrow B(s_0,(v :: O,S,h),(h,v)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{Conseq}\ \frac{\begin{array}{ll} \mathcal{G} \vdash \{A'\}\, \ell\, \{B'\}\, (I') & \forall\, s_0\ s.\ A(s_0,s) \rightarrow A'(s_0,s) \\ \forall\, s_0\ s\ t.\ B'(s_0,s,t) \rightarrow B(s_0,s,t) & \forall\, s_0\ s\ r.\ I'(s_0,s,r) \rightarrow I(s_0,s,r) \end{array}}{\mathcal{G} \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

$$\textsc{Ax}\ \frac{\begin{array}{c} \mathcal{G}(\ell) = (A,B,I) \qquad \forall\, s_0\ s.\ A(s_0,s) \rightarrow I(s_0,s,s) \\ \forall\, Q.\ \mathcal{Q}(\ell) = Q \rightarrow (\forall\, s_0\, s.\ A(s_0,s) \rightarrow Q(s_0,s)) \end{array}}{\mathcal{G} \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

Fig. 3. Program logic: selected syntax-directed rules

Rule INSTR describes the behaviour of basic instructions. The hypothetical judgement for the successor instruction involves assertions that are related to the assertions in the conclusion by the transformers for normal termination. A further hypothesis captures exceptions that are handled locally, i.e. those exceptions e to which the exception handler of the current method associates a handling instruction (predicate $\mathit{Handler}(M, l, e) = l'$). Exceptions that are not handled locally result in abrupt termination of the method. Consequently, these exceptions are modelled in a side condition that involves the method postcondition rather than a further judgemental hypothesis.

Finally, the side conditions SC_1 and SC_2 ensure that the invariant I and the local annotation Q (if existing) are satisfied in any state reaching label l.

$$SC_1 = \forall\, s_0\ s.\ A(s_0, s) \to I(s_0, s, s)$$
$$SC_2 = \forall\, Q.\ \mathcal{Q}(M, l) = Q \to (\forall\, s_0\, s.\ A(s_0, s) \to Q(s_0, s))$$

In particular, SC_2 requires us to prove any annotation that is associated with label l. Satisfaction of I in later states, and satisfaction of local annotations Q' of later program points are guaranteed by the judgement for $\mathit{suc}_M(l)$.

The rules for conditional and unconditional jumps include a hypotheses for the control flow successors, and the same side conditions for local annotations and invariants as rule Instr. No further hypotheses or side conditions regarding exceptional behaviour are required as these instructions do not raise exceptions. These rules also account for the verification of loops which on the level of bytecode are rendered as jumps. Loop invariants can be inserted as postconditions B at their program point. Rule Ax allows one to use such invariants whereas according to Definition 1 they must be established once in order for a verification to be valid.

In rule InvS, the invariant of the callee, namely Φ (more precisely: the satisfaction of Φ whenever the initial state of the callee satisfies the precondition R), and the local precondition A may be exploited to establish the invariant I. This ensures that I will be satisfied by all states that arise during the execution of M', as these states will always conform to Φ. The callee's post-condition T is used to construct the assertions that occur in the judgement for the successor instruction l'. Both conditions reflect the transfer of the method arguments and return values between the caller and the callee. This protocol is repeated in the hypothesis and the side condition for the exceptional cases which otherwise follow the pattern mentioned in the description of the rule Instr.

A similar rule for virtual methods is omitted. The rule for method returns, Ret, ties the precondition A to the post-condition B w.r.t. the terminal state that is constructed using the topmost value of the operand stack.

Finally, the *logical rules* Conseq and Ax arise from the standard rules by adding suitable side conditions for strong invariants and local assertions.

3.4 Behavioural subtyping and verified programs

We say that method specification (R, T, Φ) *implies* (R', T', Φ') if

– for all s_0 and t, $R(s_0) \to T(s_0, t)$ implies $R'(s_0) \to T'(s_0, t)$, and
– for all s_0 and s, $R(s_0) \to \Phi(s_0, s)$ implies $R'(s_0) \to \Phi'(s_0, s)$

Furthermore, we say that $\mathcal{M}$ satisfies *behavioural subtyping* for P if whenever P contains an instruction invokevirtual M' with $\mathcal{M}(M') = (\mathcal{S}', \mathcal{G}', \mathcal{Q}')$, and M overrides M', then there are $\mathcal{S}$, $\mathcal{G}$ and $\mathcal{Q}$ with $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$ such that $\mathcal{S}$ implies $\mathcal{S}'$. Finally, we call a derivation $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ *progressive* if it contains at least one application of a non-logical rule.

Definition 1. *P is* verified *with respect to $\mathcal{M}$, notation $\mathcal{M} \vdash P$, if*

– *$\mathcal{M}$ satisfies behavioural subtyping for P, and*
– *for all M, $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$, and $\mathcal{S} = (R, T, \Phi)$*
  - *a progressive derivation $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ exists for any l, A, B, and I with $\mathcal{G}(M, l) = (A, B, I)$, and*
  - *a progressive derivation $\mathcal{G} \vdash \{A\}\, M, \mathit{init}_M\, \{B\}\, (I)$ exists for*

$$A(s_0, s) \equiv s = \mathit{state}(s_0) \wedge R(s_0)$$
$$B(s_0, s, t) \equiv s = \mathit{state}(s_0) \to T(s_0, t)$$
$$I(s_0, s, r) \equiv s = \mathit{state}(s_0) \to \Phi(s_0, r).$$

As the reader may have noticed, behavioural subtyping only affects method specifications but not the proof contexts $\mathcal{G}$ or annotation tables $\mathcal{Q}$. Technically, the reason for this is that no constraints on these components are required in order to prove the logic sound. Pragmatically, we argue that proof contexts and local annotations tables of overriding methods indeed should not be related to contexts and annotation tables of their overridden counterparts, as both kinds of tables expose the internal structure of method implementations. In particular, entries in proof contexts and annotation tables are formulated w.r.t. specific program points, which would be difficult to interprete outside the method boundary or indeed across different (overriding) implementations of a method.

The distinction between progressive and non-progressive derivations prevents attempts to justify a proof context or method specification table simply by applying the axiom rule to all entries. In program logics for high-level languages, the corresponding effect is silently achieved by the unfolding of the method body in the rule for method invocations [29]. As our judgemental form does not permit such an unfolding, the auxiliary notion of progressive derivations is introduced. In our formalisation, the separation between progressive and other derivations is achieved by the introduction of a second judgement form, as described in [8].

3.5 Interpretation and soundness

Definition 2. *The triple $(\mathcal{Q}, B, I)$ is* valid at (M, l) for (s_0, s) *if*

– *for all r, if $\vdash_M l, s \Downarrow t$ then $B(s_0, s, t)$*
– *for all l' and r, if $\vdash_M l, s \Rightarrow^* l', r$ and $\mathcal{Q}(l') = Q$, then $Q(s_0, r)$, and*
– *for all r, if $\vdash_M l, s \Uparrow r$ then $I(s_0, s, r)$.*

Note that the second clause applies to annotations Q associated with arbitrary labels l' in method M that will be visited during the execution of M from (l, s) onwards. Although these annotations are interpreted without recourse to the state s, the proof of $Q(s_0, r)$ may exploit the precondition $A(s_0, s)$.

The soundness result is then as follows.

Theorem 1. *For $\mathcal{M} \vdash P$ let $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$, $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ be a progressive derivation, and $A(s_0, s)$. Then $(\mathcal{Q}, B, I)$ is valid at (M, l) for (s_0, s).*

In particular, this theorem implies that for $\mathcal{M} \vdash P$ all method specifications in $\mathcal{M}$ are honoured by their method implementations. The proof of this result may be performed in two ways. Following the approach of Kleymann and Nipkow [25, 29, 3], one would first prove that the derivability of a judgement entails its validity, under the hypothesis that contextual judgements have already been validated. For this task, the standard technique involves the introduction of relativised notions of validity that restrict the interpretation of judgements to operational judgements of bounded height. Then, the hypothesis on contextual judgements is eliminated using structural properties of the relativised validity. An alternative to this approach has been developed by Benjamin Gregoire in the course of the formalisation of the present logic. It consists of (i) defining a family of syntax-directed judgements (one judgement form for each instruction form, inlining the rule of consequence), (ii) proving that property $\mathcal{M} \vdash P$ implies that the last step in a derivation of $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ can be replaced by an application of the syntax-directed judgement corresponding to the instruction at M, l (in particular, an application of the axiom rule is replaced by the derivation for the corresponding code blocks from $\mathcal{G}$), and (iii) proving the main claim of Theorem 1 by treating the three parts of Definition 2 separately, each one by induction over the respective operational judgement.

4 Type-based verification

In this section we present a type system that ensures a constant bound on the heap consumption of bytecode programs. The type system is formally justified by a soundness proof with respect to the MOBIUS base logic, and may serve as the target formalism for type-transforming compilers.

The requirement imposed on programs is similar to that of the analysis presented by Cachera et al. in [13] in that recursive program structures are denied the facility to allocate memory. However, our analysis is presented as a type system while the analysis presented in [13] is phrased as an abstract interpretation. In addition, Cachera et al.'s approach involves the formalisation of the calculation of the program representation (control flow graph) and of the inference algorithm (fixed point iteration) in the theorem prover. In contrast, our presentation separates the algorithmic issues (type inference and checking) from semantic issues (the property expressed or guaranteed) as is typical for a type-based formulation. Depending on the verification infrastructure available at the code consumer side, the PCC certificate may either consist of (a digest of) the

typing derivation or an expansion of the interpretation of the typing judgements into the MOBIUS logic. The latter approach was employed in our earlier work [10] and consists of understanding typing judgements as derived proof rules in the program logic and using syntax-directed proof tactics to apply the rules in an automatic fashion. In contrast to [10], however, the interpretation given in the present section extends to non-terminating computations, albeit for a far simpler type system.

The present section extends the work presented in [8] as the type system is now phrased for bytecode rather than an intermediate functional language and includes the treatment of exceptions and virtual methods.

Bytecode-level type system The type system consists of judgements of the form $\vdash_{\Sigma,\Lambda} \ell : n$, expressing that the segment of bytecode whose initial instruction is located at ℓ is guaranteed not to allocate more than n memory cells. Here, ℓ denotes a program point M, l while signatures Σ and Λ assign types (natural numbers n) to identifiers of methods and bytecode instructions (in particular, when those are part of a loop), respectively.

$$\text{C-New}\ \frac{n \geq 1 \qquad M(l) = \mathsf{New}\ C \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n-1}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-Instr}\ \frac{\begin{array}{c} n \geq 1 \qquad basic(M,l) \qquad \neg M(l) = \mathsf{New}\ C \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \vdash_{\Sigma,\Lambda} M, l' : n-1 \end{array}}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-If}\ \frac{n \geq 0 \qquad M(l) = \mathsf{ifz}\ l' \qquad \vdash_{\Sigma,\Lambda} M, l' : n \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-Invoke}\ \frac{\begin{array}{c} M(l) \in \{\mathsf{invokestatic}\ M', \mathsf{invokevirtual}\ M'\} \qquad \Sigma(M') = k \\ n \geq 1 \qquad k \geq 0 \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \vdash_{\Sigma,\Lambda} M, l' : n-1 \end{array}}{\vdash_{\Sigma,\Lambda} M, l : n+k}$$

$$\text{C-Ret}\ \frac{M(l) = \mathsf{vreturn}}{\vdash_{\Sigma,\Lambda} M, l : 0} \qquad \text{C-Sub}\ \frac{\vdash_{\Sigma,\Lambda} \ell : n \qquad n \leq k}{\vdash_{\Sigma,\Lambda} \ell : k} \qquad \text{C-Assum}\ \frac{\Lambda(\ell) = n}{\vdash_{\Sigma,\Lambda} \ell : n}$$

Fig. 4. Type system for constant heap space

The rules are presented in Figure 4. The first rule, C-New, asserts that the memory consumption of a code fragment whose first instruction is $\mathsf{new}\ C$ is the increment of the remaining code. Rule C-Instr applies to all basic instructions (in the case of $\mathsf{goto}\ l'$ we take $suc_M(l)$ to be l'), except for $\mathsf{new}\ C$ – the predicate $basic(m, l)$ is defined as in Section 3.3. The memory effect of these instructions is zero, as is the case for return instructions, conditionals, and (static) method

invocations in the case of normal termination. For exceptional termination, the allocation of a fresh exception object is accounted for by decrementing the type for the code continuation by one unit. The rule C-Assum allows for using the annotation attached to the instruction if it matches the type of the instruction.

A typing derivation $\vdash_{\Sigma,\Lambda} \ell : k$ is called *progressive* if it does not solely contain applications of rules C-Sub and C-Assum. Furthermore, we call P *well-typed* for Σ, notation $\vdash_\Sigma P$, if for all M and n with $\Sigma(M) = n$ there is a local specification table Λ such that a progressive derivation $\vdash_{\Sigma,\Lambda} M, \mathit{init}_M : n$ exists, and for all ℓ with $\Lambda(\ell) = k$ we have a progressive derivation $\vdash_{\Sigma,\Lambda} \ell : k$.

Type checking and inference The tasks of checking and automatically finding (inference) of typing derivations are not our main concern here. Nevertheless, we discuss briefly how this can be achieved.

For this simple type system checking a given typing derivation amounts to verifying the inequations that arise as side conditions. Furthermore, given Σ, Λ a corresponding typing derivation can be reconstructed by applying the typing rules in a syntax-directed fashion. In order to construct Σ, Λ as well (type inference) one writes down a "skeleton derivation" with indeterminates instead of actual numeric values and then solves the arising system of linear inequalities. Alternatively, one can proceed by counting allocation statements along paths and loops in the control-flow graph.

Our main interest here is, however, the use of existing type derivations however obtained in order to mechanically construct proofs in the program logic. This will be described now.

Interpretation of the type system The interpretation for the above type system is now obtained by defining for each number n a triple $[\![n]\!] = (A, B, I)$ consisting of a precondition A, a postcondition B, and an invariant I, as follows.

$$[\![n]\!] \equiv \begin{pmatrix} \lambda\,(s_0, s).\ \mathit{True}, \\ \lambda\,(s_0, s, t).\ |\mathit{heap}(t)| \leq |\mathit{heap}(s)| + n, \\ \lambda\,(s_0, s, r).\ |\mathit{heap}(r)| \leq |\mathit{heap}(s)| + n \end{pmatrix}$$

Here, $|h|$ denotes the size of heap h and $\mathit{heap}(s)$ extracts the heap component of a state. We specialise the main judgement form of the bytecode logic to

$$\mathcal{G} \vdash \ell\ \{n\} \equiv \mathit{let}\ (A, B, I) = [\![n]\!]\ \mathit{in}\ \mathcal{G} \vdash \{A\}\,\ell\,\{B\}\,(I).$$

By the soundness of the MOBIUS logic, the derivability of a judgement $\mathcal{G} \vdash \ell\ \{n\}$ guarantees that executing the code located at ℓ will not allocate more that n items, in terminating (postcondition B) and non-terminating (invariant I) cases, provided that $\mathcal{M} \vdash P$ holds. For $(A, B, I) = [\![n]\!]$ we also define the method specification

$$\mathit{Spec}\ n \equiv (\lambda\ s_0.\ \mathit{True}, \lambda\,(s_0, t).\ B(s_0, \mathit{state}(s_0), t), \lambda\,(s_0, s).\ I(s_0, \mathit{state}(s_0), s)),$$

and for a given Λ we define $\mathcal{G}_\Lambda$ pointwise by $\mathcal{G}_\Lambda(\ell) = [\![\Lambda(\ell)]\!]$.

Finally, we say that $\mathcal{M}$ satisfies Σ, notation $\mathcal{M} \models \Sigma$, if for all methods M, $\mathcal{M}(M) = (\mathit{Spec}\ n, \mathcal{G}_\Lambda, \emptyset)$ holds precisely if $\Sigma(M) = n$, where Λ is the context associated with M in $\vdash_\Sigma P$. Thus, method specification table $\mathcal{M}$ contains for each method the precondition, postcondition and invariant from Σ, the (complete) context determined from Λ, and the empty local annotation table $\mathcal{Q}$.

We can now prove the soundness of the typing rules with respect to this interpretation. By induction on the typing rules, we first show that the interpretation of a typing judgement is derivable in the logic.

Proposition 1. *For $\mathcal{M} \models \Sigma$ let M be provided in $\mathcal{M}$ with some annotation table Λ such that $\vdash_{\Sigma,\Lambda} M, l : n$ is progressive. Then $\mathcal{G}_\Lambda \vdash M, l\ \{n\}$.*

From this, one may obtain the following, showing that well-typed programs satisfy the verified-program property:

Theorem 2. *Let $\mathcal{M} \models \Sigma$ and $\vdash_\Sigma P$, and let $\mathcal{M}$ satisfy behavioural subtyping for P. Then $\mathcal{M} \vdash P$.*

Discussion In order to improve the precision of the analysis, a possibility is to combine the type system with a null-pointer analysis. For this, we would specialise the proof rules for instructions which might throw a null-pointer exception. At program points for which the analysis guarantees absence of such exceptions, we may then use a specialised typing rule. For example, a suitable rule for the field access operation is the following.

$$\text{C-GETFLD1}\ \frac{\mathit{getField}(m,l) \quad \mathit{refNotNull}(m,l) \quad \vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n}{\vdash_{\Sigma,\Lambda} m, l : n}$$

Program points for which the analysis is unable to discharge the side condition $\mathit{refNotNull}(m,l)$ would be dealt with using the standard rule. Similarly, instructions that are guaranteed not to throw runtime exceptions (like load x, store x, dup) may be typed using the optimised rule

$$\text{C-NORTE}\ \frac{\vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n \quad \mathit{noExceptionInstr}(m,l)}{\vdash_{\Sigma,\Lambda} m, l : n}$$

We expect that justifying these specialised rules using the program logic would not pose major problems, while the formal integration with other program analyses (such as the null-pointer analysis) is a topic for future research.

5 Resource-extended program logic

In this section we give a brief overview of an extension of the MOBIUS base logic as described in Section 3 for dealing with resources in a generic way. The extension addresses the following shortcoming of the basic logic:

Resource consumption Specific resources that we would like to reason about include instruction counters, heap allocation, and frame stack height. A well-known technique for modelling these resources is *code instrumentation*, i.e. the introduction of (real or ghost) variables and instructions manipulating these. However, code instrumentation appears inappropriate for a PCC environment, as it does not provide an end-to-end guarantee that can be understood without reference to the program at hand. In particular, the overall satisfaction of a resource property using code instrumentation requires an analysis of the annotated program, i.e. a proof that the instrumentation variables are introduced and manipulated correctly. Furthermore, the interaction between additional variables of different domains, and between auxiliary variables and proper program variables is difficult to reason about.

Execution traces Here, the goal is to reason about properties concerning a full terminating or non-terminating execution of a program, for example by imposing that an execution satisfies a formula expressed in temporal logics or a policy given in terms of a security automaton. Such specifications may concern the entire execution history, i.e. be defined over a sequence of (intermediate) Bicolano states, and are thus not expressible in the MOBIUS base logic.

Ghost variables are heavily used in JML, both for resource-accounting purposes as well as functional specifications, but are not directly expressible in the base logic.

In this section we extend the base logic by a generic resource-accounting mechanism that may be instantiated to the above tasks. In addition to the work reported here, we have also performed an analysis of the usage made of ghost variables in JML, and have developed interpretations of ghost variables in native and resource-extended program logics [24]. In particular, loc.cit. contains a formalised proof demonstrating how resource counting using ghost variables in native logics may be effectively eliminated, by translating each proof derivation into a derivation in the resource-extended logic.

5.1 Semantic modelling of generic resources

In order to avoid the pitfalls of code instrumentation discussed above, a semantic modelling of resource consumption was chosen. The logic is defined over an extended operational semantics, the judgements of which are formulated over the same components as the standard Bicolano operational semantics, plus a further resource-accounting component [20]. The additional component is of the a priori unspecified type ACT, and occurs as a further component in initial, final, and intermediate states. In addition, we introduce transfer functions that update the content of this component according to the other state components, including the program counter. The operational semantics of the extended framework is then obtained by embedding each non-extended judgement form in a judgement form over extended states and invoking the appropriate transfer functions on the resource component. While these definitions of the operational semantics

are carried out once and for all, the implementation of the transfer functions themselves is programmable. Thus, realisations of the framework for particular resources may be obtained by instantiating the ACT to some specific type and implementing the transfer functions as appropriate. The program logic remains conceptually untouched, i.e. it is structurally defined as the logic from Section 3, but the definitions of assertion transformers and rules, and the soundness proof, are adapted to extended states and modified operational judgements.

In comparison to admitting the definition of ad-hoc extensions to the program logic, we argue that the chosen approach is better suited to the PCC applications, as the consumer has a single point of reference where to specify his policy, namely the implementation of the transfer functions.

5.2 Application: block-booking

As an application of the resource-extended program logic, we consider a scenario where an application repeatedly sends some data across a network provided that each such operation is sanctioned by an interaction with the user. In order to avoid authorisation requests for individual send operations, a high-level language might contain a primitive **auth**(n) that asks the user to authorise n messages in one interaction. A reasonable resource policy for the code consumer then is to require that no send operation be carried out without authorisation, and that at each point of the execution, the acquired authorisations suffice for servicing the remaining **send** operations. (For simplicity, we assume that refusal by the user to sanction an authorisation request simply blocks or leads to immediate non-termination without any observable effect.)

We note that as in the case of the logic loop constructs from the high-level language are mapped to conditional and unconditional jumps that must be typed using the corresponding rules.

We now outline a bytecode-level type and effect system for this task, for a sublanguage of scalar (integer) values and unary static methods. Effects τ are rely-guarantee pairs (m, n) of natural numbers: a code fragment with this effect satisfies the above policy whenever executed in a state with at least m unused authorisations, with at least n unused authorisations being left over upon termination. The number of authorisations that are additionally acquired, and possibly used, during the execution are unconstrained. Types $C, D, \ldots$ are sets of integers constraining the values stored in variables or operand stack positions. Judgements take the form $\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} \ell : C, \tau$, with the following components:

- the abstract store Δ maps local variables to types
- the abstract operand stack η is represented as a list of types
- Ξ is an equivalence relation relation ranging over identifiers ρ from $dom\ \Delta \cup dom\ \eta$ where $dom\ \eta$ is taken to be the set $\{0, \ldots, |\eta| - 1\}$. The role of Ξ is to capture equalities between values on the operand stack and the store.
- instruction labels $\ell = (M, l)$ indicate the current program point, as before
- the type C describes the return type
- the effect τ captures the pre-post-behaviour of the subject phrase with respect to authorisation and send events

- the proof context Λ associates sets of tuples $(\Delta, \eta, \Xi, C, \tau)$ to labels l (implicitly understood with respect to method M).
- the method signature table Σ maps method names to type signatures of the form $\forall i{\in}I.\ C_i \xrightarrow{(m_i,n_i)} D_i$. Limiting our attention to static methods with a single parameter, such a poly-variant signature indicates that for each i in some (unspecified) index set I, the method is of type $C_i \xrightarrow{(m_i,n_i)} D_i$, i.e. takes arguments satisfying constraint C_i to return values satisfying D_i with (latent) effect (m_i, n_i).

In addition to ignoring virtual methods (and consequently avoiding the need for a condition enforcing behavioural subtyping of method specifications), we also ignore exceptions. Finally, while our example program contains simple objects we do not give proof rules for object construction or field access. We argue that this impoverished fragment of the JVML suffices for demonstrating the concept of certificate generation for effects, and leave an extension to larger language fragments as future work.

For an arbitrary relation R, we let $Eq(R)$ denote its reflexive, transitive and symmetric closure. We also define the operations $\Xi - \rho$, $\Xi + \rho$ and $\Xi[\rho := \rho']$ on equivalence relation Ξ and identifiers ρ and ρ', as follows.

$$\begin{aligned} \Xi - \rho &\equiv \Xi \setminus \{(\rho_1, \rho_2) \mid \rho = \rho_1 \vee \rho = \rho_2\} \\ \Xi + \rho &\equiv \Xi \cup \{(\rho, \rho)\} \\ \Xi[\rho := \rho'] &\equiv Eq((\Xi - \rho) \cup \{(\rho, \rho')\}) \end{aligned}$$

The interpretation of position ρ in a pair (O, S) is given by $[\![x]\!]_{(O,S)} = S(x)$ and $[\![n]\!]_{(O,S)} = O(n)$. The interpretation of a triple Δ, η, Ξ in a pair (O, S) is given by the formula

$$[\![\Delta, \eta, \Xi]\!]_{(O,S)} = \begin{cases} dom\ \Delta \subseteq dom\ S \wedge |\eta| = |O| \wedge \\ \forall x \in dom\ \Delta.\ S(x) \in \Delta(x) \wedge \\ \forall i < |\eta|.\ O(i) \in \eta(i) \wedge \\ \forall (\rho, \rho') \in \Xi.\ [\![\rho]\!]_{(O,S)} = [\![\rho']\!]_{(O,S)} \end{cases}$$

With the help of these operations, the type system is now defined by the rules given in Figure 5. Due to the formulation at the bytecode level, the authorisation primitive does not have a parameter but obtains its argument from the operand stack.

The rule for conditionals, E-If, exploits the outcome of the branch condition by updating the types of all variables associated with the top operand stack position in Ξ. This limited form of copy propagation will be made use of in the verification of an example program below.

In the rule of consequence, E-Sub, subtyping on types is denoted by $C <: D$ and given by subset inclusion, and is extended to abstract stores (notation $\Delta <: \Delta'$) and abstract operand stacks (notation $\eta <: \eta'$) in a pointwise fashion. Sub-effecting is given by the reflexive closure of the rule

$$\frac{k \geq m + d \quad l \leq n + d}{(m, n) <: (k, l)}.$$

$$\text{E-Send}\ \frac{M(l) = \mathbf{send} \qquad \Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m-1, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Auth}\ \frac{M(l) = \mathbf{auth} \qquad \forall i \in C.\ i \geq k \qquad \Delta, \eta, \Xi - |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m+k, n)}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Goto}\ \frac{M(l) = \mathsf{goto}\ \ l' \qquad \Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l' : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-If}\ \frac{\begin{array}{c} M(l) = \mathsf{ifz}\ \ l' \qquad \Xi' = \Xi - |\eta| \\ \Delta_1 = \Delta[x \mapsto \Delta(x) \cap (\mathbf{Z} \setminus \{0\})]_{(|\eta|, x) \in \Xi} \\ \eta_1 = \eta[i \mapsto \eta(i) \cap (\mathbf{Z} \setminus \{0\})]_{(|\eta|, i) \in \Xi\ \wedge\ 0 \leq i < |\eta|} \\ \Delta_2 = \Delta[x \mapsto \Delta(x) \cap \{0\}]_{(|\eta|, x) \in \Xi} \\ \eta_2 = \eta[i \mapsto \eta(i) \cap \{0\}]_{(|\eta|, i) \in \Xi\ \wedge\ 0 \leq i < |\eta|} \\ \Delta_1, \eta_1, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : (m, n) \qquad \Delta_2, \eta_2, \Xi' \vdash_{\Sigma,\Lambda} M, l' : D, (m, n) \end{array}}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Store}\ \frac{M(l) = \mathsf{store}\ \ x \qquad \Xi' = (\Xi[x := |\eta|]) - |\eta| \qquad \Delta[x \mapsto C], \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Load}\ \frac{M(l) = \mathsf{load}\ \ x \qquad \Xi' = \Xi[|\eta| := x] \qquad \Delta, \Delta(x) :: \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Push}\ \frac{M(l) = \mathsf{push}\ \ c \quad \Delta, \{c\} :: \eta, \Xi + |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Binop}\ \frac{M(l) = \mathsf{binop}\ \oplus \qquad C = \{z | z = x\ \oplus\ y, x \in C_1, y \in C_2\} \qquad \Delta, C :: \eta, ((\Xi - |\eta|) - (|\eta| + 1)) + |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, C_1 :: C_2 :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-InvS}\ \frac{\begin{array}{c} M(l) = \mathsf{invokestatic}\ M' \qquad \Sigma(M') = \forall i \in I.\ C_i \xrightarrow{\tau_i} D_i \qquad k \in I \\ \Xi' = (\Xi - |\eta|) + |\eta| \qquad \Delta, D_k :: \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (n_k, n) \end{array}}{\Delta, C_k :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m_k, n)}$$

$$\text{E-Vret}\ \frac{M(l) = \mathsf{vreturn}}{\Delta, D, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (0, 0)} \qquad \text{E-Ax}\ \frac{(\Delta, \eta, \Xi, D, \tau) \in \Lambda(l)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, \tau}$$

$$\text{E-Sub}\ \frac{\begin{array}{c} \Delta', \eta', \Xi' \vdash_{\Sigma,\Lambda} \ell : C, \tau' \\ \Delta <: \Delta' \qquad \eta <: \eta' \\ C <: D \quad \tau' <: \tau \quad \Xi' \subseteq \Xi \end{array}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} \ell : D, \tau} \qquad \text{E-Univ}\ \frac{\forall\ O\ S.\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} = \mathit{False}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

Fig. 5. Type and effect system for block-booking

The final rule, E-Univ, allows us to associate an arbitrary effect and result type to a code segment under the condition that the constraints Δ, η, Ξ on the initial state are unsatisfiable. The main use of this rule is in cases where branch conditions render one branch dead code.

In order to prove the soundness of the type system in the extended program logic, we instantiate the parameter ACT to the type of finite words over the set $\{\mathbf{send}\} \cup \{\mathbf{auth}(z) \mid z \geq 0\}$ and implement the transfer functions such that each execution of the primitives **send** and **auth** results in appending the appropriate action to the trace - in case of authorisation events, the number z is obtained by inspecting the topmost value of the operand stack.

We interpret a judgement $\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)$ as the logic statement

$$[\![\Lambda]\!]_M \vdash \{\lambda\, s_0.\ \mathit{True}\}\, M, l\, \{[\![(\Delta, \eta, \Xi, m, n, D)]\!]\}\, ([\![(\Delta, \eta, \Xi, m)]\!]),$$

with the following components. The postcondition $[\![(\Delta, \eta, \Xi, m, n, D)]\!]$ is

$$\lambda\, (s_0, (O, S, h, X), (h, v, Y)).\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} \rightarrow \\ (\exists Z.\ v \in D \wedge Y = XZ \wedge |Z|_{\mathsf{auth}} + m \geq |Z|_{\mathsf{send}} + n).$$

For any terminating execution starting in an initial store and operand stack conforming to the abstractions Δ and η, and respecting the equivalence relation Ξ, this property guarantees that the return value satisfies D. Furthermore, the sub-traces for authorisation and send events (obtained by projecting from the trace Z of all events encountered during the execution of the phrase) satisfy the inequality interpreting the effect.

A similar explanation holds for the definition of the invariant $[\![(\Delta, \eta, \Xi, m)]\!]$,

$$\lambda\, (s_0, (O, S, h, X), (O', S', h', X')).\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} \rightarrow \\ (\exists Z.\ X' = XZ \wedge |Z|_{\mathsf{auth}} + m \geq |Z|_{\mathsf{send}}).$$

The local proof context $[\![\Lambda]\!]_M$ is given by

$$[(M, l) \mapsto (\mathit{True}, [\![(\Delta, \eta, \Xi, m, n, D)]\!], [\![(\Delta, \eta, \Xi, m)]\!])]_{\Lambda(l)=(\Delta,\eta,\Xi,D,(m,n))},$$

i.e. by translating the entries of Λ pointwise. Finally, each specification entry $\Sigma(M) = \forall i \in I.\ C_i \xrightarrow{(m_i,n_i)} D_i$ results in an entry $\mathcal{M}(M) = (R, T, \Phi)$ in the bytecode logic specification table, where

$$\begin{aligned} R(s_0) &= \mathit{True} \\ T((S, h, X), (h, v, Y)) &= \forall i \in I.\, S(\mathrm{arg}) \in C_i \rightarrow \\ &\qquad (\exists\ Z.\ v \in D_i \wedge Y = XZ \wedge \\ &\qquad\quad |Z|_{\mathsf{auth}} + m_i \geq |Z|_{\mathsf{send}} + n_i) \\ \Phi((S, h, X), (O, S', h', X')) &= \forall i \in I.\, S(\mathrm{arg}) \in C_i \rightarrow \\ &\qquad (\exists\ Z.\ X' = XZ \wedge |Z|_{\mathsf{auth}} + m_i \geq |Z|_{\mathsf{send}}) \end{aligned}$$

where arg is the formal parameter. Based on this interpretation, certificate generation may now be obtained by deriving the typing rules from the program logic

and introducing appropriate notions of progressive derivations and well-typed programs (in the absence of virtual methods: without a behavioural subtyping condition), in a similar way as in Section 4. The formalisation of this is left as future research.

5.3 Example

We assume two builtin integer-valued functions `size_string` yielding the number of SMS messages required to send a given string, and `size_book` which gives the size of an address book. Figure 6 presents Java-style pseudocode for sending a given string to all addresses of a given address book after requiring the necessary permissions. The program first computes the total number of SMS messages

```
public interface Parameters {
  int p=...; //some constant >= 0
}
class BlockBooking {
  static void send () {...};
  static void auth (int p) {...};
  void block_send(Java.lang.String s, addrbook b) {
       int n = size_string(s);
       int m = size_book(b);
       int nb_sms = n * m;
       int j = 0;
       int sent = 0;
       while (nb_sms - sent > 0) {
         if j > 0 {
           //current authorisations suffice
           send();
           sent = sent + 1;
           j = j - 1
         } else {
           //acquire p new authorisations
           auth (Parameters.p);
           j = Parameters.p;
         }
       }
       return 0;
  }
}
```

Fig. 6. Program for sending a message using authorisation chunks of size p

and then sends the messages where authorisations are acquired in blocks of size

p, for arbitrary fixed $p \geq 0$. The primitives for sending and authorising messages are modelled as additional (static) methods.

Figure 7 shows the bytecode for method `block_send`, which comprises six basic blocks. In order to verify that this method does not send more messages

```
0   aload_1 //variable s
1   invokestatic sizestring
4   istore_3 //variable n
5   aload_2 // variable b
6   invokestatic sizebook
9   istore 4 //variable m
11  iload_3
12  iload 4
14  imul
15  istore 5 //variable nbms
17  iconst_0
18  istore 6 //variable j
20  iconst_0
21  istore 7 //variable sent

23  iload 5
25  iload 7
27  isub
28  ifle 64

31  iload 6
33  ifle 54

36  invokestatic send
39  iload 7
41  iconst_1
42  iadd
43  istore 7
45  iload 6
47  iconst_1
48  isub
49  istore 6
51  goto 23

54  iconst_3 // parameter p
55  invokestatic auth
58  iconst_3
59  istore 6
61  goto 23

64  iconst_0
65  ireturn
```

Fig. 7. Bytecode for method `BlockBooking.block_send`.

than authorised, we derive the typing

$$[s \mapsto C, b \mapsto D], [\,], \emptyset \vdash_{\Sigma,\Lambda} \texttt{block\_send}, 0 : \{0\}, (0,0)$$

where C and D are arbitrary and

$$\begin{aligned}
\Sigma &\equiv [\texttt{sizestring} \mapsto \{(C,0,0,\mathbf{Z})\}, \texttt{sizebook} \mapsto \{(D,0,0,\mathbf{Z})\}] \\
\Lambda &\equiv [23 \mapsto \{spec_d \mid 0 \leq d\}] \\
spec_d &\equiv (\Delta_d, [\,], \Xi_d, \{0\}, (d,0)) \\
\Delta_d &\equiv [n \mapsto \mathbf{Z}, m \mapsto \mathbf{Z}, nbsms \mapsto \mathbf{Z}, j \mapsto \{d\}, sent \mapsto \mathbf{Z}^{\geq 0}] \\
\Xi_d &\equiv \{(n,n),(m,m),(nbsms,nbsms),(j,j),(sent,sent)\}.
\end{aligned}$$

The proof context Λ contains a single entry, namely a polyvariant loop invariant for instruction 23. The invariant contains one entry for each $0 \leq d$, where the index specifies precisely the content of variable j and links this value to the pre-effect. The equivalence relation relevant at this program point contains

merely the reflexive entries for all (integer) variables. The verification of the above judgement applies the rules syntax-directedly for instructions $0, \ldots, 21$, and then applies the axiom rule for label 23, guarded by an application of rule E-Sub.

The overall verification complements the verification of the above judgement with a justification of the context Λ, by providing a progressive derivation for the loop invariant. Again, this verification proceeds syntax-directedly through the loop, terminating in (subtyping-protected) applications of the rule E-Ax. At the point where method `send` is invoked (instruction label 36) a case-split is performed on the condition $d = 0$. If this condition holds, a vacuous statement is obtained as the invocation occurs in the branch $j > 0$, and our invariant ensures that j contains the value d. The vacuity is detected as the entry for j in Δ is $\emptyset$ at that point: the load instruction at label 36 inserts $(0, j)$ into Ξ, hence the type associated with j in the fall-through-hypothesis of the branch at label 33 (in particular: at label 36) is $\{d\} \cap (\mathbf{Z} \setminus \{0\}) = \emptyset$ where the term $\{d\}$ was propagated unmodified to instruction 36 from instruction 23. Consequently, the case $d = 0$ may be immediately discharged by an invocation of rule E-Univ. The case $d > 0$ admits the application of the proof rule E-Send, and the remainder of the branch is again proven in a syntax-directed fashion.

Type checking and inference Again, we briefly discuss these issues for this system. The type system is generic in that types may be arbitrary sets of integers. In order to support effective typechecking and inference one must of course restrict these sets themselves and also the sets of types that arise in annotations and method specifications. A popular and for our intended application sufficient way consists of restricting types to convex polyhedra specified by a system of linear inequalities and to confine sets of types to those arising by intersecting a fixed convex polyhedron with a hyperplane specified by one or more additional parameters. Notice that the types in our running example are all of this form.

When we make this restriction (formally by applying the subtyping rule immediately after each rule to bring the types back into the polyhedral format) then type checking amounts to checking inclusion of convex polyhedra which can be efficiently performed by linear programming. Furthermore, Farkas' Lemma also furnishes short, efficiently computable, and efficiently checkable certificates [21, 28]. Indeed, since any convex polyhedron is the intersection of hyperplanes, deciding containment of convex polyhedra reduces to deciding whether a convex polyhedron $H = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$ is contained in a hyperplane of the form $P = \{\boldsymbol{x} \mid \boldsymbol{c}^T\boldsymbol{x} \leq d\}$. This, however, is the case iff $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in H\} \leq d$; a linear programming problem. Now, the latter inequality can be certified by providing a vector $\boldsymbol{r} \geq 0$ (componentwise) such that $\boldsymbol{r}^T A = \boldsymbol{c}^T$ and $\boldsymbol{r}^T\boldsymbol{b} \leq d$. For then, whenever $\boldsymbol{x} \in H$, i.e., $A\boldsymbol{x} \leq \boldsymbol{b}$ then $\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{r}^T A\boldsymbol{x} \leq \boldsymbol{r}^T\boldsymbol{b} \leq d$. Farkas' lemmas asserts that such a vector $\boldsymbol{r}$ exists whenever $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in H\} \leq d$. Given its existence we can efficiently compute it by minimising $\boldsymbol{y}^T\boldsymbol{b}$ subject to $\boldsymbol{y}^T A = \boldsymbol{c}^T$ and $\boldsymbol{y} \geq 0$.

Regarding automatic type inference as opposed to type checking one has to find unknown convex polyhedra specified by fixpoint equations. Besson et al. [12]

report that this can be done by iteration using widening heuristics from [19]. The range and efficiency remains, however, unexplored in loc. cit. In our particular application we expect constraints to be sufficiently simple so that these heuristics or those proposed in [26] will be successful. Inference of the equivalence relations Ξ can be achieved by employing standard copy-propagation techniques known from compiler constructions.

6 Discussion

We have described the use of the Mobius base logic as a unified backend for both program analyses and type systems. The Mobius base logic has been formally proved sound with respect to the Bicolano formalisation of the JVM. Compared to direct soundness proofs of type systems and analyses with respect to Bicolano the use of the Mobius base logic as an intermediary offers two distinctive advantages. First, the soundness proof of the Mobius base logic already does much of the work that is common to soundness proofs, in particular inducting on steps in the operational semantics and stack height. The Mobius logic is more transparent and allows for proof by invariant and recursion. Secondly, the standardised format of assertions in the Mobius base logic makes it easier to compare results of different type systems and analyses and also to assess whether the asserted property coincides with the intuitively desired property.

The resource extension to both Bicolano and the Mobius base logic allows for direct specification and certification of resource-related intensional properties without having to go through indirect observations such as values of ordinary program variables that are externally known to reflect some resource behaviour. This is particularly important in the PCC scenario where providers and users of specifications and certificates do not coincide and might have different objectives.

Similarly, the strong invariants enhance the expressive power of the Mobius base logic compared to standard Hoare logics in that resource behaviour of nonterminating programs is appropriately accounted for. In this way, the usual strong guarantees of type systems and program analyses may be adequately reflected in the logic.

We have demonstrated this use of the Mobius base logic on one of the Mobius case studies: a block-booking scheme whose deployment could avoid the inflation of permission requests that lead to social vulnerabilities.

Acknowledgements This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein. We are grateful to all members of the MOBIUS Working Group on work package 3, in particular Benjamin Gregoire, David Pichardie, Aleksy Schubert and Randy Pollack, for the numerous discussions on program logics, JML, and types, and on formalising these in theorem provers. The constructive feedback from the reviewers helped us to improve content and presentation of the paper.

References

1. E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *Logic for Programming Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 380–397. Springer-Verlag, 2005.
2. A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.
3. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Theorem Proving in Higher-Order Logic*, volume 3223 of *Lecture Notes in Computer Science*, pages 34–49, Berlin, Sept. 2004. Springer-Verlag.
4. F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.
5. G. Barthe and C. Fournet, editors. *Trustworthy Global Computing, Third Symposium (TGC'07), Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
7. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, Apr. 2003.
8. L. Beringer and M. Hofmann. A bytecode logic for JML and types. In *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science 4279, pages 389–405. Springer-Verlag, 2006.
9. L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Workshop*. IEEE Press, 2007.
10. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452, pages 347–362. Springer-Verlag, 2005.
11. F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 2006.
12. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Inria Research Report 6333, 2007.
13. D. Cachera, T. P. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer-Verlag, 2005.
14. B. Chang, A. Chlipala, and G. Necula. A framework for certified program analysis and its applications to mobile-code safety. In E. Emerson and K.S.Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI'06), Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, 2006.
15. B. Chang, A. Chlipala, G. Necula, and R. Schneck. The open verifier framework for foundational verifiers. In J. Morrisett and M. Fähndrich, editors, *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 1–12. ACM Press, 2005.

16. MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements. Available online from http://mobius.inria.fr, 2006.
17. MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.
18. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, Aug. 1977.
19. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.
20. P. Czarnik and A. Schubert. Extending operational semantics of the java bytecode. In Barthe and Fournet [5], pages 57–72.
21. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
22. X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 67–78, New York, NY, USA, January 2007. ACM Press.
23. R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In U. Montanari, D. Sannella, and R. Bruni, editors, *Trustworthy Global Computing, Second Symposium (TGC'06), Revised Selected Papers*, volume 4661 of *Lecture Notes in Computer Science*, pages 116–131. Springer-Verlag, 2007.
24. M. Hofmann and M. Pavlova. Elimination of ghost variables in program logics. In Barthe and Fournet [5], pages 1–20.
25. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.
26. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. ACM POPL 2004*, pages 330–341, 2004.
27. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
28. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC Computer Science Laboratory, June 1981.
29. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2002.
30. D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inia.fr/bicolano. Summary appears in [17], 2006.
31. C. L. Quigley. A Programming Logic for Java Bytecode Programs. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, (TPHOLs'03), Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.
32. M. Wildmoser. *Verified Proof Carrying Code*. PhD thesis, Institut für Informatik, Technische Universität München, 2005.
33. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, Aug. 2004.
34. T. Y. Woo and S. S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.

Efficient Type-Checking for Amortised Heap-Space Analysis

Martin Hofmann and Dulma Rodriguez

Department of Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
{martin.hofmann|dulma.rodriguez}@ifi.lmu.de

Abstract. The prediction of resource consumption in programs has gained interest in the last years. It is important for a number of areas, notably embedded systems and safety critical systems. Different approaches to achieve bounded resource consumption have been analysed. One of them, based on an amortised complexity analysis, has been studied by Hofmann and Jost in 2006 for a Java-like language.
In this paper we present an extension of this type system consisting of more general subtyping and sharing relations that allows us to type more examples. Moreover we describe efficient automated type-checking for a finite, annotated version of the system. We prove soundness and completeness of the type checking algorithm and show its efficiency.
Keywords: Type systems, Resource analysis, Semantics, OOP

1 Introduction

The prediction of resource consumption in programs has gained interest in the last years. It is important for a number of areas, in particular embedded systems and mobile computing. A variety of approaches to resource analysis have been proposed based in particular on recurrence solving [AAG+07,Gro01], abstract interpretation [GL98,NCQR05], sized types [HP99], and amortised analysis [HJ03,HJ06,Cam08].

The amortised approach which the present paper belongs to is particularly useful in situations where heap-allocated data structures must be costed whose size is proportional to parts of the input. Typical examples are various sorting algorithms where trees, lists, or heaps appear as intermediate data structures. In such cases amortised analysis can infer very good bounds based on intuitive programmer annotations in the form of types and the solution of linear inequations.

In [HJ06] amortised analysis has been applied to a Java-like class-based object-oriented language without garbage collection, but with explicit deallocation similar to C's `free()`. The evaluation of such programs is carried out by maintaining a set of free memory units called freelist. When an object is created, a number of heap units required to store it is taken from the freelist if it contains enough units, otherwise the program execution is aborted. Finally, each deallocated heap unit is returned to the freelist.

The goal of the analysis is to predict a bound on the initial size that the freelist must have so that a given program may be executed without causing unsuccessful abortion due to insufficient memory. This has been achieved by combining amortized analysis [Tar85,Oka98] with type-based techniques in order to define potentials.

Essentially each object is ascribed an abstracted portion of the freelist, referred to as *potential*, which is just a number, denoting the size of freelist portion associated with the object. Any object creation must be paid for from the potential in scope. The initial potential thus represents an upper bound on the total heap consumption.

While type inference and automated type checking have already been developed for a functional language within the EmBounded Project ([HDF$^+$05], [HBH$^+$07]), most of the properties of the type system for the Java-like language (called Resource Aware JAva – RAJA) are still unknown.

This paper provides algorithmic typing rules for that system. We prove soundness and completeness of algorithmic typing with respect to the declarative typing from [HJ06]. This allows for automatic type checking under relatively mild annotations. In particular, we automatically construct types arising from sharing and conditionals which had to be provided manually beforehand. This enables a realistic implementation of the system which we also provide.

The notion of subtyping we use is slightly more flexible than the one from [HJ06] and thus allows more examples to be typed. Semantic soundness of the improved system is a direct extension of the soundness proof in [HJ06] and can be found in the following manuscript: [HJR].

Contents. Section 2 describes briefly the system RAJA and motivates it with some examples. In Section 3 we define the type-checking algorithm and show its soundness and completeness w.r.t. the declarative system. We then argue that typechecking can be performed efficiently, i.e. in small-degree polynomial time. Finally, in Sections 4 and 5, we discuss future and related work.

2 FJEU and RAJA

Our formal model of Java, FJEU, is an extension of Featherweight Java (FJ) [IPW99] with attribute update, conditional and explicit deallocation. It is thus similar to Classic Java [FKF98]. An FJEU program $\mathscr{C}$ is a partial finite map from class names to class definitions, which we also refer to as *class table*. Each class table $\mathscr{C}$ implies a subtyping relation $<:$ among the class names in the standard way by inheritance. The syntax of FJEU is given in Fig. 1. The let-normal form of terms was merely chosen to eliminate boring redundancies from our proofs. In our implementation we transform nested expressions into let-normal form and infer a type for the let expressions by a simple preprocessing.

We will use a couple of shorthand notations: We write $\mathsf{S}(C)$ to denote the *super-class* D of a class C, provided that C has a super-class. We write $\mathsf{A}(C)$ to denote the ordered set of attributes of C, including inherited ones, i.e. $\mathsf{A}(C) := \{a_1, \ldots, a_k\} \dot\cup \mathsf{A}(D)$. We write $C.\boldsymbol{a_i}$ to denote the class type of each attribute a_i of class C. Similarly we write $\mathsf{M}(C)$ to denote the set of all defined method names

$$
\begin{array}{rll}
c ::= & \texttt{class}\ C\ [\texttt{extends}\ D]\ \{A_1;\ldots;A_k; M_1\cdots M_j\} & \\
A ::= & C\ a & \\
M ::= & C_0\ m(C_1\ x_1,\ldots,C_j\ x_j)\{\texttt{return}\ e;\} & \\
e ::= & x & \text{(Variable)}\\
 & \mid \mathsf{null} & \text{(Constant)}\\
 & \mid \texttt{new}\ C & \text{(Construction)}\\
 & \mid \texttt{free}(x) & \text{(Destruction)}\\
 & \mid (C)x & \text{(Cast)}\\
 & \mid x.a_i & \text{(Access)}\\
 & \mid x.a_i \texttt{<-} x & \text{(Update)}\\
 & \mid x.m(x_1,\ldots,x_j) & \text{(Invocation)}\\
 & \mid \texttt{if}\ x\ \texttt{instanceof}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \text{(Conditional)}\\
 & \mid \mathsf{let}\ C\,x\ \texttt{=}\ e_1\ \mathsf{in}\ e_2 & \text{(Let)}
\end{array}
$$

Fig. 1. The syntax of FJEU

of C, including inherited ones. For a method m of class C we write $\mathsf{M}_{\mathsf{body}}(C,m)$ to denote the term that comprises the *method body* of method m and $C.m$ to denote the *method type* of m in class C. We base our statical resource analysis on the standard operational semantics that can be found in [HJR].

Example 1 (Copy of singly-linked lists). Suppose we have defined a class of singly-linked lists in an object-oriented style which harnesses dynamic dispatch to obtain the functionality of pattern-matching. Most programmers would use this style only for more complex tree-like data structures relying on "null" to model the empty list. We use it here in order to have a simple enough running example.

```
class List { List copy(){return null;} }
class Nil  extends List { List copy() { return this; }}
class Cons extends List { int elem; List next;
     List copy(){ let List res = new Cons in
           let List res1 = res.elem <- this.elem in
           let List res2 = res1.next <- this.next.copy() in return res2;}}
```

2.1 The system RAJA

Definition 1. *A* RAJA program *is an annotation of an FJEU class table $\mathscr{C}$ in the form of a sextuple $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \diamondsuit(\cdot), \mathsf{A}^{\text{get}}(\cdot\,,\cdot)\,, \mathsf{A}^{\text{set}}(\cdot\,,\cdot)\,, \mathsf{M}(\cdot\,,\cdot))$ specified as follows:*

$\mathscr{V}$ is a possibly infinite set of views. *A* RAJA class *or* refined type *consists of a class C and a view r and is written C^r. We use the letters r, s, p, q to denote views. The meaning of views is given by the maps:*

1. *$\diamondsuit(\cdot)$ assigns to each RAJA class C^r a number $\diamondsuit(C^r) \in \mathbb{D}$, where $\mathbb{D} = \mathbb{Q}^+ \cup \infty$.*
2. *$\mathsf{A}^{\text{get}}(\cdot\,,\cdot)$ and $\mathsf{A}^{\text{set}}(\cdot\,,\cdot)$ assign to each RAJA class C^r and attribute $a \in \mathsf{A}(C)$ two views $q = \mathsf{A}^{\text{get}}(C^r, a)$ and $s = \mathsf{A}^{\text{set}}(C^r, a)$.*
3. *$\mathsf{M}(\cdot\,,\cdot)$ assigns to each RAJA class C^r and method $m \in \mathsf{M}(C)$ having method type $E_1,\ldots,E_j \to E_0$ a j-ary polymorphic RAJA method type $\mathsf{M}(C^r, m)$. A j-*ary polymorphic RAJA method type *is a (possibly empty or infinite) set of j-ary monomorphic RAJA method types. A j*-ary monomorphic RAJA method

type *consists of* $j+1$ *views and two numbers* $p, q \in \mathbb{D}$, *written* $r_1, \ldots, r_j \xrightarrow{p/q} r_0$. *We sometimes write* $E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ *to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.*

We introduce views and RAJA classes because we want to be able to assign objects of the same class different potentials.The number $\Diamond(\cdot)$ will be used to define the potential of a heap configuration under a given static RAJA typing. The exact definition is omitted here for lack of space and can be found in [HJR]. Essentially, the potential of a program state is the sum of the annotations of all its objects determined by their RAJA-type. Each access path (alias) to an object makes a separate contribution to that sum. In reasonable typings of circular data structures one arranges that all but finitely many paths make a nonzero contribution.

If $D = C.a$ is the FJEU type of attribute a in C then the RAJA class $D^{\mathsf{A}^{\mathsf{get}}(C^r.a)}$ will be the type used when reading a, whereas the (intendedly stronger) type $D^{\mathsf{A}^{\mathsf{set}}(C^r.a)}$ must be used when updating a. The stronger typing is needed since an update will possibly affect several aliases.

If a method m has a RAJA method type $E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ then it may be called with arguments $v_1 : E_1^{r_1}, \ldots, v_j : E_j^{r_j}$, whose associated potential will be consumed, as well as an additional potential of p. Upon successful completion the return value will be of type $E_0^{r_0}$ hence carry an according potential. In addition to this a potential of another q units will be returned.

Example 2 (RAJA annotation of copy of singly-linked lists).

We aim at analysing the heap-space requirements of the program of Example 1. It is clear that the memory consumption of a call `l.copy()` will equal the length of the list `l`. To calculate this formally we use a view rich which assigns to `List` itself the potential 0, to `Nil` the potential 0 and to `Cons` the potential 1. Another view is needed to describe the result of `copy()` for otherwise we could repeatedly copy lists without paying for it. Thus, we introduce another view poor that assigns potential 0 to all classes. In the following we show the RAJA annotation of Example 1 in the syntax of our implementation.

```
class List { rich, poor : pot = 0;
      rich : List<poor>,0 copy(0) { return null; }
}
class Nil extends List { rich, poor : pot = 0;
      rich : List<poor>,0 copy(0) { return this; }
}
class Cons extends List { rich : pot = 1; poor : pot = 0;
      rich : List<rich,rich> next;
      poor : List<poor,poor> next;
      rich, poor: int elem;

      rich : List<poor>,0 copy(0) { let List res = new Cons in
        let List res1 = res.elem <- this.elem in
        let List res2 = res1.next <- this.next.copy() in return res2; }
}
```

The RAJA type of the method copy states that it is only defined in $\mathsf{List}^{\mathsf{rich}}$, $\mathsf{Nil}^{\mathsf{rich}}$ and $\mathsf{Cons}^{\mathsf{rich}}$, but not in, e.g., $\mathsf{List}^{\mathsf{poor}}$. It will consume the potential of this and no additional potential. Upon successful completion the return value will be of type $\mathsf{List}^{\mathsf{poor}}$ hence carry potential 0. In addition to this the method will return no more potential. Thus, the typing amounts to saying that the memory consumption of every call to copy is bounded by the potential of this, that in case of $\mathsf{Cons}^{\mathsf{rich}}$ is equal 1 and in case of $\mathsf{Nil}^{\mathsf{rich}}$ is equal 0. If a list of length n is to be copied, the method will be called $n+1$ times, and the potential consumed will be bounded by n. More examples can be found in the RAJA web page [raj].

RAJA Subtyping Relation RAJA subtyping is an extension of FJEU subtyping ($<:$), which is based on inheritance. We provide here a new definition of subtyping w.r.t. [HJ06]. There, a subtyping relation $r \sqsubseteq s$ on views was defined, based on all classes of the class table. Then, subtyping of RAJA classes $C^r <: D^s$ was defined as $C <: D$ and $r \sqsubseteq s$. This made subtyping unnecessarily rigid. For example $\mathsf{Nil}^{\mathsf{rich}} <: \mathsf{Nil}^{\mathsf{poor}}$ did not hold because $\mathsf{rich} \sqsubseteq \mathsf{poor}$ did not hold due to the class Cons. The new subtyping relation is defined directly on refined types. However, the straightforward definition where $C^r <: D^s$ only depends on C and D is unsound. It is necessary to analyse the subclasses of C and D as well because resource usage is determined by the dynamic type of the expressions.

Definition 2 (Subtyping of RAJA types). *We define a preorder $<:$ on RAJA types C^r, D^s where $C <: D$ in $\mathscr{C}$ and $r, s \in \mathscr{V}$, as the largest relation $(C^r <: D^s)$ such that $C^r <: D^s \iff$ for each $E <: C$, $F <: D$ with $E <: F$:*

$$\Diamond(E^r) \geq \Diamond(F^s) \tag{2.1}$$

$$\forall a \in \mathsf{A}(F)\ .\ (F.a)^{\mathsf{A}^{\mathsf{get}}(E^r,a)} <: (F.a)^{\mathsf{A}^{\mathsf{get}}(F^s,a)} \tag{2.2}$$

$$\forall a \in \mathsf{A}(F)\ .\ (F.a)^{\mathsf{A}^{\mathsf{set}}(F^s,a)} <: (F.a)^{\mathsf{A}^{\mathsf{set}}(E^r,a)} \tag{2.3}$$

$$\forall m \in \mathsf{M}(F)\ .\ \forall \beta \in \mathsf{M}(F^s,m)\ .\ \exists \alpha \in \mathsf{M}(E^r,m)\ .\ (F.m)^{\alpha} <: (F.m)^{\beta} \tag{2.4}$$

where we extend $<:$ to monomorphic RAJA method types as follows:

Definition 3 (Subtyping of RAJA methods). *If $D.m = E_1, \ldots, E_j \to E_0$, $\alpha = r_1, \ldots, r_j \xrightarrow{p/q} r_0$ and $\beta = s_1, \ldots, s_j \xrightarrow{t/u} s_0$ then $(D.m)^{\alpha} <: (D.m)^{\beta}$ is defined as $p \leq t$ and $q \geq u$ and $E_0^{r_0} <: E_0^{s_0}$ and $E_i^{s_i} <: E_i^{r_i}$ for $i = 1, \ldots, j$.*

Sharing Relation The sharing relation $\curlyvee(\cdot\,|\,\cdot)$ is important for correctly using variables more than once. In a RAJA program, if a variable is to be used more than once, then the different occurrences must be given different types which are chosen such that the individual potentials assigned to each occurrence add up to the total potential available for that variable. For example if we have $l : \mathsf{List}^{\mathsf{rich}}$ we can use the variable l with the types $\mathsf{List}^{\mathsf{s}_1}$ and $\mathsf{List}^{\mathsf{s}_2}$ if $\curlyvee\big(\mathsf{List}^{\mathsf{rich}}\,|\mathsf{List}^{\mathsf{s}_1}, \mathsf{List}^{\mathsf{s}_2}\big)$ holds. In [HJ06] sharing was defined on views, i.e. $\curlyvee(r\,|s_1, \ldots, s_n)$ which is less flexible and precludes several examples.

Definition 4 (Sharing Relation). *We define the* sharing relation *between a single RAJA type C^r and a multiset of RAJA types $D^{s_1}, \ldots, D^{s_n}$ written*

$\curlyvee(C^r\,|D^{s_1},\ldots,D^{s_n})$ *as the largest relation* $\curlyvee$*, such that if* $\curlyvee(C^r\,|D^{s_1},\ldots,D^{s_n})$ *then for all* $E <: C$*,* $F <: D$ *with* $E <: F$*:*

$$\Diamond(E^r) \geq \sum_i \Diamond(F^{s_i}) \tag{2.5}$$

$$\forall i\,.\,E^r <: F^{s_i} \tag{2.6}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(F))\,.\,\curlyvee\Big((F.a)^{\mathsf{A}^{\mathsf{get}}(E^r,a)}\,\Big|(F.a)^{\mathsf{A}^{\mathsf{get}}(F^{s_1},a)},\ldots,(F.a)^{\mathsf{A}^{\mathsf{get}}(F^{s_n},a)}\Big) \tag{2.7}$$

We define sharing similarly to subtyping, so that the following can be proved: subtyping and sharing coincide when the multiset of RAJA types consists of only one element.

Lemma 1. $C^r <: C^s \iff \curlyvee(C^r\,|C^s)$

Typing RAJA The RAJA-typing judgment is formally defined by the rules in Figure 2. The type system allows us to derive assertions of the form $\Gamma \vdash^{n}_{n'} e : C^r$ where e is an expression or program phrase, C is an FJEU class, r is a view (so C^r is a refined type). Γ maps variables occurring in e to refined types; we often write Γ_x instead of $\Gamma(x)$. Finally n, n' are nonnegative numbers. The meaning of such a judgment is as follows. If e terminates successfully in some environment η and heap σ with unbounded memory resources available then it will also terminate successfully with a bounded freelist of size at least n plus the potential ascribed to η, σ with respect to the typings in Γ. Furthermore, the freelist size upon termination will be at least n' plus the potential of the result with respect to the view r.

The typing rules extend the typing rules of FJEU. The most interesting ones are $(\Diamond Share)$ and $(\Diamond Waste)$. First we notice that they are not syntax directed. Thus, they need to be eliminated when we come to implement the system in the next section. $(\Diamond Waste)$ corresponds to the rule of subsumption of subtyping systems and weakens context, type, and effect. Herein, $\Gamma <: \Theta$ means $\forall x \in \Theta\,.\,\Gamma_x <: \Theta_x$.

The purpose of the $(\Diamond Share)$ rule is to ensure that a variable can be used twice without duplication of potential. Suppose we have the following expression:

$$\Gamma, l\!:\!\mathsf{List}^{\mathsf{rich}} \vdash^{n}_{n'} \mathsf{let}\ nl = l.\mathsf{copy}()\ \mathsf{in}\ l.\mathsf{copy}() : \mathsf{List}^{\mathsf{poor}} \tag{2.8}$$

If we allow the second call to the copy method we would be creating objects without "paying" for it, which would be unsound. Since the method copy is only defined for the view rich, the only possibility of typing (2.8) would be that $\curlyvee\big(\mathsf{List}^{\mathsf{rich}}\,\big|\mathsf{List}^{\mathsf{rich}},\mathsf{List}^{\mathsf{rich}}\big)$ would hold, but it does not because $\Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) < \Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) + \Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big)$. Notice that the declarative type system as it is gives no procedure to find those intermediate views. To actually find them in order to implement the system is not trivial and will be discussed in the next section.

The judgment $\vdash m : \alpha\ \mathsf{ok}$ means that α is a valid RAJA type for a method m if the method body of m can be typed with the arguments, return type and effects as specified in α. Programs, then, are well-typed if all method bodies admit the announced type and, moreover, view and potential annotations are compatible with subtyping. Formally,

RAJA Typing $\boxed{\Gamma \vdash^{n}_{n'} e : C^r}$

$$\frac{}{\varnothing \vdash^{\Diamond(C^r)+1}_{0} \texttt{new}\ C : C^r}\ (\Diamond New) \qquad \frac{}{x{:}C^r \vdash^{0}_{\Diamond(C^r)+1} \texttt{free}(x) : E^s}\ (\Diamond Free)$$

$$\frac{C <: E}{x{:}E^r \vdash^{0}_{0} (C)x : C^r}\ (\Diamond Cast) \qquad \frac{}{\varnothing \vdash^{0}_{0} \mathsf{null} : C^r}\ (\Diamond Null) \qquad \frac{}{x{:}C^r \vdash^{0}_{0} x : C^r}\ (\Diamond Var)$$

$$\frac{s = \mathsf{A}^{\mathrm{get}}(C^r, a) \qquad D = C.a}{x{:}C^r \vdash^{0}_{0} x.a : D^s}\ (\Diamond Access) \qquad \frac{\mathsf{A}^{\mathrm{set}}(C^r, a) = s \qquad C.a = D}{x{:}C^r, y{:}D^s \vdash^{0}_{0} x.a\texttt{<-}y : C^r}\ (\Diamond Update)$$

$$\frac{\Gamma_1 \vdash^{n}_{n'} e_1 : D^s \qquad \Gamma_2, x{:}D^s \vdash^{n'}_{n''} e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ C\ x\ \texttt{=}\ e_1\ \mathsf{in}\ e_2 : C^r}\ (\Diamond Let)$$

$$\frac{\left(E_1^{q_1}, \ldots, E_j^{q_j} \xrightarrow{n/n'} E_0^{q_0}\right) \in \mathsf{M}(C^r, m)}{x{:}C^r, y_1{:}E_1^{q_1}, \ldots, y_j{:}E_j^{q_j} \vdash^{n}_{n'} x.m(y_1, \ldots, y_j) : E_0^{q_0}}\ (\Diamond Invocation)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash^{n}_{n'} e_1 : C^r \qquad \Gamma \vdash^{n}_{n'} e_2 : C^r}{\Gamma \vdash^{n}_{n'} \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : C^r}\ (\Diamond Conditional)$$

$$\frac{\curlyvee(D^s \,|D^{q_1}, \ldots, D^{q_n}) \qquad \Gamma, y_1{:}D^{q_1}, \ldots, y_n{:}D^{q_n} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y_1, \ldots, x/y_n] : C^r}\ (\Diamond Share)$$

$$\frac{n \geq u \quad n + u' \geq n' + u \quad \Theta \vdash^{u}_{u'} e : D^s \quad \Gamma <: \Theta \quad D^s <: C^r}{\Gamma \vdash^{n}_{n'} e : C^r}\ (\Diamond Waste)$$

RAJA Method Typing $\boxed{\vdash m : \alpha\ \mathsf{ok}}$

$$\frac{m \in \mathsf{M}(C) \qquad \alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in \mathsf{M}(C^r, m) \qquad \curlyvee(C^r \,|C^q, C^s)}{\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \vdash^{n + \Diamond(C^s)}_{n'} \mathsf{M}_{\mathsf{body}}(C, m) : E_0^{r_0}}\ (\Diamond MBody)$$
$$\vdash m : \alpha\ \mathsf{ok}$$

Fig. 2. Typing RAJA

Definition 5 (Well-typed RAJA-program). *A RAJA-program* $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot), \mathsf{M}(\cdot,\cdot))$ *is* well-typed *if for all* $C \in \mathscr{C}$ *and* $r \in \mathscr{V}$ *the following conditions are satisfied:*

1. $\mathsf{S}(C) = D \Rightarrow C^r <: D^r$
2. $\forall a \in \mathsf{A}(C)\ .\ (C.a)^{\mathsf{A}^{\mathrm{set}}(C^r,a)} <: (C.a)^{\mathsf{A}^{\mathrm{get}}(C^r,a)}$
3. $\forall m \in \mathsf{M}(C)\ .\ \forall \alpha \in \mathsf{M}(C^r, m)\ .\ \vdash m : \alpha\ \mathsf{ok}$

2.2 Algorithmic Views and Complete RAJA Programs

In this section we define algorithmic views which provide least upper and greatest lower bounds for subtyping restricted to refinements of a fixed FJEU type and also a formal addition operation on these refinements allowing us to infer the necessary type of a variable from the types of its (multiple) occurrences. Finally, they include operations to construct the intermediate views in method typings.

Recall the copy method of Example 2. We need one item of potential in order to create a $\mathsf{Cons}^{\mathsf{poor}}$ object . We said before that this object creation will be payed with the potential of this, but how exactly? In order to use the potential of the variable this of RAJA-class $\mathsf{Cons}^{\mathsf{rich}}$, we put it in the context with a modified type, for example, $\mathsf{Cons}^{\mathsf{rich}\dot{-}1}$, which is a view defined just like rich but with potential 0 everywhere. Moreover we find another view with potential 1, which we call $1(\mathsf{rich})$, such that $\curlyvee\Big(\mathsf{Cons}^{\mathsf{rich}} \,\Big|\, \mathsf{Cons}^{\mathsf{rich}\dot{-}1}, \mathsf{Cons}^{1(\mathsf{rich})}\Big)$ holds. Then we can derive: $\mathsf{this} : \mathsf{Cons}^{\mathsf{rich}\dot{-}1} \vdash^{\frac{1}{0}} \mathsf{let\ List\ res} = \mathsf{new\ Cons}^{\mathsf{poor}}\ \mathsf{in}\ \ldots\ \mathsf{in\ return\ res};$

The declarative rule gives no information about how to find the views q and s. In order to find them algorithmically, we will introduce special algorithmic views like $\mathsf{rich} \dot{-} 1$ and $1(\mathsf{rich})$.

Definition 6 (Algorithmic views). *Let* $\mathscr{R}$ *be a RAJA-program. We extend the given set of views* $\mathscr{V}$ *by algorithmic views*

$$\delta, \gamma ::= \ s_1 \vee s_2 \ \mid\ s_1 \wedge s_2 \ \mid\ s_1 + s_2 \ \mid\ s \dot{-} \mathsf{n} \ \mid\ \mathsf{n}(s) \qquad s, s_1, s_2 \in \mathscr{V}, n \in \mathbb{D}$$

by extending the given maps $\Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot), \mathsf{M}(\cdot,\cdot)$ *according to Fig. 3.*

Definition 7 (Complete RAJA-program). *A RAJA-program* $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot), \mathsf{M}(\cdot,\cdot))$ *is* complete *if the following conditions are satisfied. Let* $* \in \{\wedge, \vee, +\}$.

1. $s_1 * s_2 \in \mathscr{V}$*, for all* $s_1, s_2 \in \mathscr{V}$*.*
2. $s \dot{-} \mathsf{n}, \mathsf{n}(s) \in \mathscr{V}$*, for all* $s \in \mathscr{V}, n \in \mathbb{D}$*.*
3. *The annotation table of* $\mathscr{R}$ *satisfies the equations from Def. 6.*

Given a RAJA program $\mathscr{R}$ we can complete it with algorithmic views. $C^{s_1 \vee s_2}$ is the least upper bound of C^{s_1} and C^{s_2} and $C^{s_1 \wedge s_2}$ is the greatest lower bound of C^{s_1} and C^{s_2}. $C^{s_1+s_2}$ is defined such that $\curlyvee(C^s \,|C^{s_1}, C^{s_2})$ is equivalent to $C^s <: C^{s_1+s_2}$. This way we can deal only with subtyping instead of sharing, which is simpler and more intuitive. Finally, the views $\mathsf{n}(s)$ are neutral views of potential n and set-views like s. They are intended to be used together with the views $s \dot{-} \mathsf{n}$, which are nothing but the view s, with n units of potential stripped-off. This way, we get $\curlyvee\Big(C^s \,\Big|\, C^{s\dot{-}\mathsf{n}}, C^{\mathsf{n}(s)}\Big)$. These algorithmic views are

Let $C \in \mathscr{C}$, $a \in \mathsf{A}(C)$ and $m \in \mathsf{M}(C)$. We set:

$$
\begin{array}{ll}
\Diamond(C^{s_1 \wedge s_2}) & = \mathsf{max}(\Diamond(C^{s_1}), \Diamond(C^{s_2})) \\
\Diamond(C^{s_1 \vee s_2}) & = \mathsf{min}(\Diamond(C^{s_1}), \Diamond(C^{s_2})) \\
\Diamond(C^{s_1 + s_2}) & = \Diamond(C^{s_1}) + \Diamond(C^{s_2}) \\
\Diamond\left(C^{\mathsf{n}(s)}\right) & = n \\
\Diamond\left(C^{s \dot{-} \mathsf{n}}\right) & = \begin{cases} \Diamond(C^s) \dot{-} n & \Diamond(C^s) \geq n \\ 0 & \text{otherwise} \end{cases}
\end{array}
\qquad
\begin{array}{ll}
\mathsf{A}^{\mathrm{get}}(C^{s_1 \wedge s_2}, a) & = \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \wedge \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{get}}(C^{s_1 \vee s_2}, a) & = \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \vee \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{get}}(C^{s_1 + s_2}, a) & = \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) + \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{get}}\left(C^{\mathsf{n}(s)}, a\right) & = \mathsf{0}(s) \\
\mathsf{A}^{\mathrm{get}}\left(C^{s \dot{-} \mathsf{n}}, a\right) & = \mathsf{A}^{\mathrm{get}}(C^s, a)
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{M}(C^{s_1 \wedge s_2}, m) & = \mathsf{M}(C^{s_1}, m) \cup \mathsf{M}(C^{s_2}, m) \\
\mathsf{M}(C^{s_1 \vee s_2}, m) & = \mathsf{M}(C^{s_1}, m) \vee \mathsf{M}(C^{s_2}, m) \\
\mathsf{M}(C^{s_1 + s_2}, m) & = \mathsf{M}(C^{s_1}, m) \cup \mathsf{M}(C^{s_2}, m) \\
\mathsf{M}\left(C^{\mathsf{n}(s)}, m\right) & = \emptyset \\
\mathsf{M}\left(C^{s \dot{-} \mathsf{n}}, m\right) & = \mathsf{M}(C^s, m)
\end{array}
\qquad
\begin{array}{ll}
\mathsf{A}^{\mathrm{set}}(C^{s_1 \wedge s_2}, a) & = \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{set}}(C^{s_1 \vee s_2}, a) & = \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \wedge \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{set}}(C^{s_1 + s_2}, a) & = \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \\
\mathsf{A}^{\mathrm{set}}\left(C^{\mathsf{n}(s)}, a\right) & = \mathsf{A}^{\mathrm{set}}(C^s, a) \\
\mathsf{A}^{\mathrm{set}}\left(C^{s \dot{-} \mathsf{n}}, a\right) & = \mathsf{A}^{\mathrm{set}}(C^s, a)
\end{array}
$$

$$
\mathsf{M}(C^{s_1}, m) \vee \mathsf{M}(C^{s_2}, m) = \{(C.m)^{\alpha_1 \vee \alpha_2} \mid \alpha_1 \in \mathsf{M}(C^{s_1}, m), \alpha_2 \in \mathsf{M}(C^{s_2}, m)\}
$$

$$
(C.m)^{\alpha_1 \vee \alpha_2} = E_1^{p_1 \wedge q_1}, \ldots, E_j^{p_j \wedge q_j} \xrightarrow{\mathsf{max}(n,m)/\mathsf{min}(n',m')} E_0^{p_0 \vee q_0}
$$

Fig. 3. Definition of $\Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot,\cdot), \mathsf{M}(\cdot,\cdot)$ of algorithmic views

useful for implementing $\vdash m : \alpha\, \mathsf{ok}$. If we need to use n units of potential of the type C^s of this in the method body of a given method, we give this the type $C^{s \dot{-} \mathsf{n}}$ and use the potential of $C^{\mathsf{n}(s)}$ in the method.

Of course, we are free to use the algorithmic views from the beginning and in particular in the provided class and method typings. They may be seen as a shorthand for a longer table which includes them explicitly. We stress, though, that efficient type checking for *incomplete* programs is not possible with the techniques from this paper. We do not consider typechecking of incomplete programs to be of any practical relevance.

The following lemma summarizes the desirable order- and proof-theoretic properties of algorithmic views:

Lemma 2. *Let $C, D \in \mathscr{C}$ and $s, s_1, s_2, \ldots, s_n, q_1, q_2, \ldots, q_n \in \mathscr{V}$.*

1. *$C^{s_1 \vee s_2}$ is the least upper bound of C^{s_1} and C^{s_2}.*
2. *$C^{s_1 \wedge s_2}$ is the greatest lower bound of C^{s_i}.*
3. *$\curlyvee(C^s \,|C^{s_1}, \ldots, C^{s_n}) \iff C^s <: C^{s_1 + \ldots + s_n}$.*
4. *If $C^s <: C^{s_1 + \ldots + s_n}$ and $C^{s_i} <: C^{q_i}$ for all i, then $C^s <: C^{q_1 + \ldots + q_n}$.*
5. *$C^{s + \mathsf{0}(s)} = C^s$.*
6. *If $n \leq \Diamond(C^s)$ then $\curlyvee\left(C^s \,\middle|\, C^{s \dot{-} \mathsf{n}}, C^{\mathsf{n}(s)}\right)$. Moreover, $\curlyvee(C^s \,|C^{s_1}, C^{s_2})$ and $\Diamond(C^{s_2}) \geq n$ imply $C^{s \dot{-} \mathsf{n}} <: C^{s_1}$.*

Algorithmic typechecking now faces one more obstacle. Officially, one method can have infinitely many RAJA types. This does not compromise semantic type soundness, but must of course be restricted to finitely many to enable algorithmic type checking.

Moreover, the rule ($\Diamond Invocation$) chooses non-deterministically one monomorphic RAJA method type according to the given method call. In order for algorithmic typing to be efficient (not NP-complete) we need to make sure that there is an optimal such choice in any situation.

Definition 8. *If* $\alpha = r_1, \ldots, r_j \xrightarrow{p/p'} r_0$ *and* $\beta = s_1, \ldots, s_j \xrightarrow{n/n'} s_0$ *then* $\alpha \sqsubseteq \beta$ *iff* $p \leq n$ *and* $p - p' \leq n - n'$.

Definition 9. *A RAJA-program is* algorithmic *if it is finite, complete, and for all* C, r, m *the set* $\mathsf{M}(C^r, m)$ *is totally ordered by the ordering in Def. 8.*

From now on we assume that all RAJA-programs are algorithmic without explicit notice.

3 Algorithmic Typing of RAJA Programs

In this section we present an algorithm for typechecking RAJA programs. Algorithmic type-checking must consist of syntax directed rules, thus, the rules ($\Diamond Share$) and ($\Diamond Waste$) must be integrated in other rules. Instead of using ($\Diamond Waste$), we integrate subtyping in the rules.

The purpose of the ($\Diamond Share$) rule is to ensure that a variable can be used more than once without unsound incrementation of potential. The main challenge for implementing it is that it contains no information about how to find the views q_1 to q_n for the different occurrences. The current implementation does not include inference of these views. Instead, every variable occurrence has been annotated with the corresponding view, which can be an algorithmic view. The task of the type checker is then to check the correctness of the given sharing, or, more exactly, since, as we saw in last section, using algorithmic views a sharing task can be reduced into a subtyping task, the algorithm checks only subtyping. The inference of these intermediate views remains under investigation.

The computed resource annotations in rules ($\vdash Let$) and ($\vdash Cond.$) are a bit intricate. Ultimately, they are justified by soundness and completeness. Rule ($\vdash Let$) may be easier to understand if broken down into the two cases $m \geq n'$ and $m < n'$. In the latter case the output of the first computation suffices to satisfy the second one. In the former case extra input potential must be provided for the second computation. In rule ($\vdash Cond.$) we must cater for both computations, hence the max and the min. The adaptations $u - n$ and $u - m$ cater for the case where, say, $m \geq n$ units were provided due to the max, yet the first branch of the conditional was taken hence only n units "used" and vice versa.

In the rule ($\vdash Inv.$) we choose the minimal RAJA monomorphic type that satisfies the subtyping conditions. Since the algorithmic system considers only finite programs and the set of RAJA monomorphic types is totally ordered according to $\sqsubseteq$, every nonempty subset of $M(G^r, m)$ has a minimal element.

We define the judgment $\Delta^{\Psi} \vdash^{n}_{n'} e^{\circ} \leftleftarrows C^{\gamma}$ inductively by the rules in Figure 4, where Δ, e° and C^{γ} are inputs and Ψ, n and n' are outputs. Δ is an FJEU context, i.e. a map from variable names to FJEU types. Ψ is a map from variable names to algorithmic views. C^{γ} is an algorithmic RAJA type, which is an FJEU class refined with an algorithmic view and e° is an annotated FJEU expression.

Algorithmic RAJA Typing $\boxed{\Delta^{\Psi} \mid\!\!\frac{n}{n'}\ e^{\circ} \leftleftarrows C^{r}}$

$$\frac{D^{\gamma} <: C^{\gamma}}{\Delta^{\Psi_\emptyset} \mid\!\!\frac{\Diamond(D^{\gamma})+1}{0}\ \mathsf{new}\ D \leftleftarrows C^{\gamma}}\ (\vdash New) \qquad \frac{E^{q} <: C^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}E^{q} \mid\!\!\frac{0}{0}\ x^{q} \leftleftarrows C^{\gamma}}\ (\vdash Var)$$

$$\frac{}{\Delta^{\Psi_\emptyset}, x{:}C^{q} \mid\!\!\frac{0}{\Diamond(C^{q})+1}\ \mathsf{free}\,(x^{q}) \leftleftarrows E^{\gamma}}\ (\vdash Free)$$

$$\frac{D <: E\ \ (\text{or}\ \ E <: D) \qquad D^{q} <: C^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}E^{q} \mid\!\!\frac{0}{0}\ (D)x^{q} \leftleftarrows C^{\gamma}}\ (\vdash Cast) \qquad \frac{}{\Delta^{\Psi_\emptyset} \mid\!\!\frac{0}{0}\ \mathsf{null} \leftleftarrows C^{\gamma}}\ (\vdash Null)$$

$$\frac{\mathsf{A}^{\mathsf{get}}(C^{r}, a) = q \qquad C.a = E \qquad E^{q} <: D^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}C^{r} \mid\!\!\frac{0}{0}\ x^{r}.a \leftleftarrows D^{\gamma}}\ (\vdash Access)$$

$$\frac{\mathsf{A}^{\mathsf{set}}(E^{q}, a) = s \qquad E.a = D \qquad F^{p} <: D^{s} \qquad E^{q} <: C^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}E^{q}, {}^{+}\, y{:}F^{p} \mid\!\!\frac{0}{0}\ x^{q}.a \leftarrow y^{p} \leftleftarrows C^{\gamma}}\ (\vdash Update)$$

$$\frac{\Delta^{\Psi'} \mid\!\!\frac{n}{n'}\ e_1^{\circ} \leftleftarrows D^{\gamma_1} \quad \Delta^{\Psi''}, x{:}D^{\gamma_1} \mid\!\!\frac{m}{m'}\ e_2^{\circ} \leftleftarrows C^{\gamma_2}}{\Delta^{\Psi'+\Psi''} \mid\!\!\frac{\mathsf{max}(\,n, n+m-n'\,)}{\mathsf{max}(\,m', m'+n'-m\,)}\ \mathsf{let}\ D\,x = e_1^{\circ}\ \mathsf{in}\ e_2^{\circ} \leftleftarrows C^{\gamma_2}}\ (\vdash Let)$$

$$\frac{x \in \Delta \quad \Delta^{\Psi'} \mid\!\!\frac{n}{n'}\ e_1^{\circ} \leftleftarrows C^{\gamma} \quad \Delta^{\Psi''} \mid\!\!\frac{m}{m'}\ e_2^{\circ} \leftleftarrows C^{\gamma} \quad u = \mathsf{max}(\,m, n)}{\Delta^{\Psi' \wedge \Psi''} \mid\!\!\frac{u}{\mathsf{min}(\,n'+u-n, m'+u-m)}\ \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1^{\circ}\ \mathsf{else}\ e_2^{\circ} \leftleftarrows C^{\gamma}}\ (\vdash Cond.)$$

$$\frac{p/p' = \mathsf{arg\ min}\{\,(E_1^{q_1}, \ldots, E_j^{q_j} \xrightarrow{p/p'} E_0^{q_0}) \in M(G^{r}, m) \mid \forall i\,.\,F_i^{t_i} <: E_i^{q_i}, E_0^{q_0} <: C^{\gamma}\,\}}{\Delta^{\Psi_\emptyset}, x{:}G^{r}, {}^{+}\, y_1{:}F_1^{t_1}, {}^{+}\, \ldots, {}^{+}\, y_j{:}F_j^{t_j} \mid\!\!\frac{p}{p'}\ x^{r}.m\,(y_1^{t_1}, \ldots, y_j^{t_j}) \leftleftarrows C^{\gamma}}\ (\vdash Inv.)$$

Typecheck function

$$\mathsf{typecheck}(\Delta, e^{\circ}, C^{\gamma}) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^{\Psi} \mid\!\!\frac{n}{n'}\ e^{\circ} \leftleftarrows C^{\gamma} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

Algorithmic RAJA Method Typing $\boxed{\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in \mathsf{M}(C^{r}, m) \\ \mathtt{this}{:}C^{\beta},\ x_1{:}E_1^{\beta_1}, \ldots, x_j{:}E_j^{\beta_j} \mid\!\!\frac{u}{u'}\ \mathsf{M}_{\mathsf{body}}(C, m)^{\circ} \leftleftarrows E_0^{r_0} \\ E_i^{r_i} <: E_i^{\beta_i} \quad p = u \dot{-} n \quad u' \geq n' + u - (n+p) \quad \Diamond(C^{r}) \geq p \quad C^{r \dot{-} \mathsf{p}} <: C^{\beta}\end{array}}{\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}}$$

Fig. 4. Algorithmic RAJA Typing

The notation Δ^{Ψ} means that for every variable $x \in \Delta$, if $\Delta_x = C$ and $\Psi_x = \delta$ then $\Delta_x^{\Psi} = C^{\delta}$. We also use the notation $\Delta^{\Psi_1+\Psi_2}$ for meaning that if $\Delta_x^{\Psi_1} = C^{\delta_1}$ and $\Delta_x^{\Psi_2} = C^{\delta_2}$ then $\Delta_x^{\Psi_1+\Psi_2} = C^{\delta_1+\delta_2}$. The meaning of $\Delta^{\Psi_1 \wedge \Psi_2}$ is similar. We write $x{:}C^r, {}^+ y{:}D^s$ for the following two cases. The usual case is $x \neq y$ and then it means nothing but $x{:}C^r, y{:}D^s$. On the other hand, if $x = y$, then $C = D$ too, and the notation means $x{:}C^{r+s}$. We write $\Delta^{\Psi_\emptyset}$ for meaning $\Delta_x^{\Psi_\emptyset} = C^{0(s)}$ where $\Delta_x = C$ and s is one of the view annotations of x or any view if x is not used in the program. The idea is to return neutral views for variables that are not used in the given expression. Finally, let e° denote an annotated RAJA expression. In summary, we define the partial function $\mathsf{typecheck}(\Delta, e^\circ, C^\gamma)$ (Fig. 4).

Next, we define the algorithmic judgment $\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}$ based on algorithmic typing. (Fig. 4). The typechecking algorithm returns a greater context than the declared one. This has to be checked. Moreover, it calculates the space consumption u of the method body. If $u \leq n$ then n items are enough and we do not need any potential from this. Otherwise, we calculate how many items of potential we need from this, i.e. $p = u \dot{-} n$, and we of course have to check whether the potential of this is at least p. Finally, the amount of freelist units u' released by the expression should be at least $n' + u - (n + p)$.

In the following we show that the algorithmic typing system we just defined is correct w.r.t. the declarative typing system of RAJA. If Γ is a RAJA context, we write $|\Gamma|$ for meaning its underlying FJEU context.

Lemma 3 (Soundness of algorithmic RAJA typing).
If $\Delta^{\Psi} \ \vdash^{n}_{n'} \ e^\circ \leftleftarrows C^\gamma$ *then* $\Delta^{\Psi} \ \vdash^{n}_{n'} \ e : C^\gamma$.

Proof. By induction on algorithmic typing derivations, using the $(\Diamond Waste)$ rule and Lemma 2.

Lemma 4 (Soundness of algorithmic RAJA method typing). *Given a RAJA type* C^r*, a method* $m \in \mathsf{M}(C)$ *and a RAJA method type* $\alpha \in \mathsf{M}(C^r, m)$*, if* $\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}$ *then* $\vdash m : \alpha\ \mathsf{ok}$.

Proof. Follows by Lemma 3.

The completeness proof is a bit more complicated than the soundness proof. The reason for this is that we have eliminated the rules $(\Diamond Share)$ and $(\Diamond Waste)$ and we have to show that typing derivations that use these rules are still admissible in the algorithmic system. The following lemma states the admissibility of sharing in the algorithmic system.

Lemma 5 (Share). *Let* $\Delta^{\Psi}, y_1 : D^{\delta_1}, \ldots, y_n : D^{\delta_n} \ \vdash^{n}_{n'} \ e^\circ \leftleftarrows C^\gamma$*. Then* $\Delta^{\Psi}, x : D^{\delta} \ \vdash^{n}_{n'} \ e[x/y_1, \ldots, x/y_n]^\circ \leftleftarrows C^\gamma$ *where either* $\delta = \delta_1 + \ldots + \delta_n$ *or* $\delta = \delta_1 \wedge \ldots \wedge \delta_n$.

Proof. By induction on algorithmic typing derivations.

Lemma 6 (Waste). *Let* $\Lambda^{\Psi} \ \vdash^{u}_{u'} \ e^\circ \leftleftarrows D^\gamma$*,* $D^\gamma <: C^\delta$ *and* $\Delta <: \Lambda$ *then* $\Delta^{\Psi} \ \vdash^{w}_{w'} \ e^\circ \leftleftarrows C^\delta$ *for some* $w \leq u$ *and* $w' \geq u' + w - u$.

Proof. By induction on algorithmic typing derivations.

Lemma 7 (Completeness of algorithmic RAJA typing).
If $\Gamma \vdash^{n}_{n'} e : C^r$ *then there is an annotated version* e° *of the expression* e *with* $|\Gamma|^\Psi \vdash^{u}_{u'} e^\circ \leftleftarrows C^r$ *for some* $u \leq n$ *and* $u' \geq n' + u - n$ *so that* $\Gamma <: |\Gamma|^\Psi$.

Proof. By induction on typing derivations, using Lemma 5 and 6.

Lemma 8 (Completeness of algorithmic RAJA method typing). *Given a RAJA type* C^r, *a method* $m \in \mathsf{M}(C)$ *and a RAJA method type* $\alpha \in \mathsf{M}(C^r, m)$, *if* $\vdash m : \alpha$ ok *then* $\vdash_{\mathsf{a}} m : \alpha$ ok.

Proof. Follows by Lemma 7.

The statements relating to polynomial time below make the assumption that the size of method typings, i.e. $|\mathsf{M}(C^r, m)|$ is constant. Otherwise the definition of $\mathsf{M}(C^{r_1}, m) \vee \mathsf{M}(C^{r_2}, m)$ may lead to exponential blowup.

Lemma 9 (Efficiency of algorithmic RAJA typing).
$\Delta^\Psi \vdash^{n}_{n'} e^\circ \leftleftarrows C^r$ *is decidable in polynomial time.*

Proof (sketch). The syntax-directed backwards application of the algorithmic typing rules produces a linear number of subtyping and sharing constraints. Furthermore, the algorithmic view expressions occurring in these constraints are themselves of linear size. It then suffices to restrict attention to the views that occur as subexpressions of the ones appearing in the constraints. Their number is therefore polynomial in the size of the program. A complete table of the subtyping and sharing judgments for this relevant subset can then be computed iteratively in polynomial time. In practice, a goal-directed implementation performs even better.

Lemma 10. *Given a RAJA class* C, *a view* r, *a method* $m \in \mathsf{M}(C)$ *and a RAJA method type* $\alpha \in \mathsf{M}(C^r, m)$, $\vdash m : \alpha$ ok *is decidable.*

Proof. Follows by Lemmas 4, 8 and 9.

Theorem 1 (Efficiency of RAJA typing). *Given a RAJA -Program* $\mathscr{R}$, *its well-typedness is decidable in polynomial time.*

4 Related work

Since [HJ06] several authors have made contributions towards costing heap consumption of object-oriented programs. [MP07] uses methods from abstract interpretation and term rewriting (quasi interpretations) to estimate the size of data structures and thus indirectly heap consumption. The approach is promising, but aliasing does not seem to have been taken into account properly and not many examples are given. The interpretation of methods must be provided manually.

COSTA [AAG+07] is similar in that it assigns cost functions to methods and program parts. These refer directly to heap consumption and are given as solutions of automatically constructed recurrence systems. The main contribution

of COSTA is an improved solver for these recurrences. COSTA is not as general as RAJA which, however, is not fully automatic.

Another promising fully automatic system is [GMC09] which works by instrumenting code with resource-counting, integer-valued "ghost"-variables and using modern tools from static analysis for estimating their range of values. The examples given are stunning, but do not involve dynamically allocated data structures. With further progress with automatic analysis of arithmetic relationships between integer variables systems like SPEED may eventually render type-based analyses obsolete. More likely, however, is a combination of the two.

Finally, Java(X) [DTW07] is a type system quite similar to RAJA and developed independently which has, however, a different purpose, namely ensuring the correct usage of resources like files etc. according to a specified protocol. The paper [DTW07] does not present algorithmic type checking, let alone automatic type inference; it is likely that the algorithmic system presented here could be adapted to Java(X).

5 Conclusions

We have provided a type checking algorithm for RAJA programs and proved its correctness and efficiency in the sense of polynomial-time computability. In order to do this, we introduced algorithmic views which render the subtyping lattice more well behaved and could also be a useful addition to the declarative system which is exposed to the programmer. In this way, we were able to get rid of most type annotations in method bodies although we still have to indicate the types of multiple occurrences of a variable, i.e., how the potential belonging to the variable is to be split among the different occurrences.

The algorithmic typechecking and the implementation allow us to investigate larger examples which might prompt further extensions to RAJA. In particular, we would like to investigate the typability of the Iterator pattern and more challengingly patterns involving callbacks like Observer. From a pragmatic viewpoint, polymorphic quantification over views could be a useful extension, too.

Of course, full-blown type inference is also on our agenda, thus potentially rendering RAJA into a push-button analysis.

Acknowledgment We acknowledge support by the EU integrated project MOBIUS IST 15905. We thank Andreas Abel, Lennart Beringer and Steffen Jost for valuable comments.

References

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2007.

[Cam08] Brian Campbell. *Type-based amortized stack memory prediction*. PhD thesis, University of Edinburgh, 2008.

[DTW07] Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with java(X). In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 550–574. Springer, 2007.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 171–183, New York, January 1998. Association for Computing Machinery.

[GL98] Gustavo Gómez and Yanhong A. Liu. Automatic accurate cost-bound analysis for high-level languages. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada*. Springer, 1998. LNCS 1474.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 127–139. ACM, 2009.

[Gro01] Bernd Grobauer. *Topics in Semantics-based Program Manipulation*. PhD thesis, BRICS Aarhus, 2001.

[HBH+07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In *7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, Proceedings*, pages 13–18, 2007.

[HDF+05] Kevin Hammond, Roy Dyckhoff, Christian Ferdinand, Reinhold Heckmann, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert F. Pointon, Norman Scaife, Jocelyn Srot, and Andy Wallace. The embounded project (project start paper). In Marko C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 195–210. Intellect, 2005.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[HJR] Martin Hofmann, Steffen Jost, and Dulma Rodriguez. Type-based amortised heap space analysis. (complete soundness proof). In http://raja.tcs.ifi.lmu.de/download/files/rajaSoundProof.pdf.

[HP99] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space:, June 21 1999.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[MP07] Jean-Yves Marion and Romain Péchoux. Resource control of object-oriented programs. *CoRR*, abs/0706.2293, 2007. informal publication.

[NCQR05] Huu Hai Nguyen, Wei Ngan Chin, Shengchao Qin, and Martin C. Rinard. Memory usage inference for object-oriented programs. January 2005.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[raj] http://raja.tcs.ifi.lmu.de.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

Membership Checking in Greatest Fixponts Revisited

Martin Hofmann
LMU Munich
mhofmann@ifi.lmu.de

Dulma Rodriguez
LMU Munich
rodrigue@ifi.lmu.de

Abstract

Pierce (2002) presents an efficient algorithm for computing membership in the greatest fixpoint of *invertible* operators in a goal-directed way. In this paper we provide a new proof of correctness for it based on coinduction. Moreover, we extend the algorithm for computing membership in the gfp of arbitrary monotone operators and prove this extension correct in a very similar way.

1 Introduction

We are interested in computing membership in the greatest fixpoint of a monotone operator on the powerset of some given set. Rather than computing the entire fixpoint by Knaster-Tarski iteration we want to depart from a given goal. This may be advantageous if the size of the underlying set or of the greatest fixpoint is large compared to the portion relevant for determining membership of a particular element. For a concrete example consider the operator $\mathcal{F}(X) = \{x \mid x+1 \mod 5 \in X\}$ on the powerset of $G = \{0,\ldots,2^{100}\}$. Obviously, the largest fixpoint consists of G itself; determining this by Knaster-Tarski iteration is infeasible though. If we only want to check whether a particular element, say 23 is in the gfp we can commence with the goal $23 \in \mathsf{gfp}?$. This leads to the sequence of subgoals $4 \in \mathsf{gfp}?$, $0 \in \mathsf{gfp}?$, $1 \in \mathsf{gfp}?$, $2 \in \mathsf{gfp}?$, $3 \in \mathsf{gfp}?$, $4 \in \mathsf{gfp}?$ at which point we are done because we have discovered a loop in the sequence of subgoals that have arisen.

We are specially interested in deciding subtyping for RAJA types. The RAJA system is a refinement of an extension of Featherweight Java (FJ) [IPW01] with attribute update (FJEU), with the goal of statically analysing the heap space consumption of object-oriented programs. The system has been first described by Hofmann and Jost in [HJ06]. Recently, the current authors analysed algorithmic typing of RAJA programs [HR09]. Briefly, RAJA types are FJEU classes refined with a possibly infinite set of *views*. Subtyping for RAJA types is defined as the greatest fixpoint of a monotone operator, similarly to the definitions of subtyping for other recursive types like tree types or μ-types [Pie02, Chapter 21].

Subtyping algorithms for recursive types have been widely studied in the past. Amadio and Cardelli gave the first subtyping algorithm for recursive types [AC93]. Brandt and Henglein's [BH98] showed the underlying coinductive nature of Amadio and Cardelli's algorithm. In [Pie02, Chapter 21] Pierce gives an overview of many algorithms for membership checking for greatest fixed points and how they can be used to decide subtyping for recursive types.

RAJA subtyping, however, is a bit more complicated than most of the other definitions of subtyping for recursive types because in RAJA methods can have many different types. Therefore, in order to check that a RAJA type C^r is a subtype of a RAJA type D^s we need to check that for a given method m for all its method types in D^s there is a method type in C^r with some properties. This causes that the *support* of a given goal is not a set of subgoals as usual but a boolean combination of subgoals.

In this paper we will extend the efficient algorithm for membership checking for greatest fixed points described in [Pie02, Chapter 21.6] to a more general version where the support of a given goal is a positive boolean expression. Moreover, we provide a new proof of correctness for both algorithms. We found the proof in [Pie02, Chapter 21.6] difficult to extend and provide therefore a more abstract coinductive proof which can be easily adapted to the new algorithm.

Contents. In Section 2 we describe and prove correct an algorithm for membership checking in greatest fixed points of monotone operators closed under intersection. In Section 3 we extend the algorithm to arbitrary monotone operators. In Section 4 we instantiate the second algorithm in order to decide subtyping for the RAJA system.

2 Invertible Operators

Let G be a set. We let $\mathcal{P}(G)$ denote the powerset of G and $\mathcal{PF}(G)$ denote the set of finite subsets of G. If $\mathcal{F}:\mathcal{P}(G)\to\mathcal{P}(G)$ is a monotone operator we write $\mathsf{gfp}(\mathcal{F})$ for its greatest fixpoint. We have $\mathsf{gfp}(\mathcal{F})=\mathcal{F}(\mathsf{gfp}(\mathcal{F}))$ and whenever $X\subseteq\mathcal{F}(X)$ then $X\subseteq\mathsf{gfp}(\mathcal{F})$. The latter principle is called coinduction. We may also use the notation $\nu X.\mathcal{F}(X)$ for $\mathsf{gfp}(\mathcal{F})$.

In the following we review a goal-directed algorithm for membership checking for greatest fixed points described in [Pie02, Chapter 21.6]. This algorithm works only for a special kind of operators, *invertible operators*, which we now characterize.

A given element $g\in G$ can be generated by a monotone operator $\mathcal{F}$ in many ways, which means that there can be more than one set $X\subseteq G$ such that $g\in\mathcal{F}(X)$. We call any such set a *generating* set for g. We focus here on the class of invertible operators, where each g has at most one minimal generating set.

Definition 2.1. *A monotone operator $\mathcal{F}$ is said to be* invertible *if, for all $g\in G$, the collection of sets*

$$G_g=\{X\subseteq G\mid g\in\mathcal{F}(X)\}$$

is either empty or contains a unique finite member that is a subset of all the others.

When F is invertible, the partial function $\mathsf{support}_{\mathcal{F}}:G\to\mathcal{PF}(G)$ is defined like this:

$$\mathsf{support}_{\mathcal{F}}(g)=\begin{cases}X & \text{if } X\in G_g \text{ and } \forall X'\in G_g.\,X\subseteq X'\\ \uparrow & \text{if } G_g=\emptyset\end{cases}$$

That is, the support of a goal g is the least generating set X for g, or undefined if g is not supported in $\mathcal{F}$.

Definition 2.2. *Let G be a set, $A\subseteq G$, $f:G\to\mathcal{PF}(G)$. A monotone operator $\mathcal{F}_{f,A}$ is defined by*

$$\begin{array}{lcl}\mathcal{F}_{f,A} & : & \mathcal{P}(G)\to\mathcal{P}(G)\\ \mathcal{F}_{f,A}(X) & = & \{g\mid g\in A\wedge f(g)\subseteq X\}\end{array}$$

Then, the support of a goal is given by the function f and it is only defined for elements $g\in A$:

$$\mathsf{support}_{\mathcal{F}_{f,A}}(g)=\begin{cases}f(g) & \text{if } g\in A\\ \uparrow & \text{otherwise}\end{cases}$$

The following result seems to be folklore, well-known e.g. in the field of predicate transformers. The operators $\mathcal{F}_{f,A}$ are equivalent to invertible operators and to monotone operators closed under intersection where every goal has a finite support.

Theorem 2.3. *Let $\mathcal{F}:\mathcal{P}(G)\to\mathcal{P}(G)$ be a monotone operator. The following are equivalent:*

1. *There exists f,A such that $\mathcal{F}=\mathcal{F}_{f,A}$.*
2. *For each $g\in\mathcal{F}(G)$ there exists a finite support set $S\in\mathcal{PF}(G)$ such that $g\in\mathcal{F}(S)$ and for all $X_1,X_2\in\mathcal{PF}(G)$ one has $\mathcal{F}(X_1\cap X_2)=\mathcal{F}(X_1)\cap\mathcal{F}(X_2)$.*
3. *$\mathcal{F}$ is invertible.*

Membership checking

Figure 1 shows an algorithm for membership checking in the greatest fixed point of $\mathcal{F}_{f,A}$. The idea of this membership algorithm is to run $\mathcal{F}$ backwards: to check membership for an element g, we need to ask how g could have been generated by $\mathcal{F}$. The advantage of an invertible $\mathcal{F}$ is that there is at most one way to generate a given g. We have to be careful though, a goal g might be supported e.g. by the same goal g. If we do not detect these kind of loops, the algorithm will not terminate. Therefore we keep a set of assumptions U that is empty at the beginning and that will be incremented with every goal we handle. This way we are able to detect a loop if we check whether the current goal is a member of the set of assumptions, in which case we finish with a positive answer. The following algorithm takes a set of assumptions U as an argument and returns another set of assumptions as a result. This allows it to record

Algorithm 1.

$$
\begin{array}{lcl}
\mathsf{test} & : & G \times \mathcal{P}(G) \to \mathcal{P}(G)_\perp \\
\mathsf{test}(g,U) & = & \mathsf{if}\ g \in U\ \mathsf{then}\ U \\
 & & \mathsf{else\ if}\ g \notin A\ \mathsf{then\ fail} \\
 & & \mathsf{else} \\
 & & \quad \mathsf{let}\ \{h_1,\ldots,h_n\} = f(g)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ V_1 = \mathsf{test}(h_1, U \cup \{g\})\ \mathsf{in} \\
 & & \quad \mathsf{let}\ V_2 = \mathsf{test}(h_2, V_1)\ \mathsf{in} \\
 & & \quad \ldots \\
 & & \quad \mathsf{let}\ V_n = \mathsf{test}(h_n, V_{n-1})\ \mathsf{in} \\
 & & \quad\ V_n
\end{array}
$$

Figure 1: Algorithm for membership checking of greatest fixed points.

the subtyping assumptions that have been generated during completed recursive calls and reuse them in later calls. For failure we use the convention: if an expression B fails, then $\mathsf{let}\ A = B\ \mathsf{in}\ C$ also fails.

This algorithm has been described and proved correct in [Pie02, Chapter 21.6]. In [SC05], Costa Seco and Caires have used it as well for defining subtyping for a class-based object oriented language where classes are first class polymorphic values. We provide here a more abstract correctness proof based on coinduction.

Theorem 2.4. *1. if G is a finite set the* $\mathsf{test}(g,U)$ *terminates.*

2. $\mathsf{test}(g,\emptyset) = V \iff g \in \nu X.\mathcal{F}_{f,A}(X)$.

Proof. 1. Termination of the algorithm follows using $|G \setminus U|$ as a ranking function.

2. Let $\mathcal{N}(U) := \nu X.\{h \mid h \in U \vee (h \in A \wedge f(h) \subseteq X)\}$. Note that $\mathcal{N}(U) = \nu X.U \cup \mathcal{F}_{F,A}(X)$. Consequently, $\mathcal{N}(\emptyset) = \nu X.\mathcal{F}_{f,A}(X)$. The goal follows then from the more general results:

   (a) $\mathsf{test}(g,U) = V \Rightarrow g \in \mathcal{N}(U)$ and $U \subseteq V \subseteq \mathcal{N}(U)$.

   (b) $\mathsf{test}(g,U) = \mathsf{fail} \Rightarrow g \notin \mathcal{N}(U)$.

   which we prove simultaneously by induction on the runtime of the computation of $\mathsf{test}(g,U)$.

   Case $g \in U$. Then $\mathsf{test}(g,U) = U$ by definition and $g \in \mathcal{N}(U)$ since $U \subseteq \mathcal{N}(U)$.

   Case $g \notin U$ and $g \notin A$. Then $\mathsf{test}(g,U) = \mathsf{fail}$ and $g \notin \mathcal{N}(U)$ since $\mathcal{N}(U) \subseteq U \cup A$.

   Case $g \notin U$ and $g \in A$. We consider the representative case $f(g) = \{h_1, h_2\}$.

      Case $\mathsf{test}(h_1, U \cup \{g\}) = V_1$ and $\mathsf{test}(h_2, V_1) = V_2$.
      Then by induction hypothesis we get $h_1 \in \mathcal{N}(U \cup \{g\})$ and $U \cup \{g\} \subseteq V_1 \subseteq \mathcal{N}(U \cup \{g\})$ and $h_2 \in \mathcal{N}(V_1)$ and $V_1 \subseteq V_2 \subseteq \mathcal{N}(V_1)$. From monotonicity of $\mathcal{N}(.)$ then follows $\mathcal{N}(V_1) \subseteq \mathcal{N}(\mathcal{N}(U \cup \{g\})) = \mathcal{N}(U \cup \{g\})$ easily [1], hence, we get $f(g) \subseteq \mathcal{N}(U \cup \{g\})$ (*).
      Next we claim that $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. One direction is clear by monotonicity of $\mathcal{N}(.)$. For the other direction we use coinduction with $X_0 = \mathcal{N}(U \cup \{g\})$. To conclude $X_0 \subseteq \mathcal{N}(U)$ we thus have to prove $X_0 \subseteq U \cup \{h \mid h \in A \wedge f(g) \subseteq X_0\}$ which we now do.
      Pick $h \in X_0 = \mathcal{N}(U \cup \{g\})$.
      From the definition of $\mathcal{N}(.)$ we get that $h \in U$ or $h = g$ or $f(g) \subseteq X_0$. The first and third case immediately yield the desired result. In the second case ($g = h$) we get $f(g) \subseteq X_0$ from (*). So we proved $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. Then we have $f(g) \subseteq \mathcal{N}(U)$ and $g \in A$, thus, we get the desired $g \in \mathcal{N}(U)$. Moreover, we get $U \subseteq U \cup \{g\} \subseteq \mathcal{N}(U \cup \{g\}) \subseteq \mathcal{N}(U)$.

      Case $\mathsf{test}(h_i, U \cup \{g\}) = \mathsf{fail}$ for some i. Then $g \notin \mathcal{N}(U)$ follows easily by I.H.

□

[1] $\mathcal{N}(\mathcal{N}(U)) = \mathcal{N}(U)$ follows by monotonicity of $\mathcal{N}(.)$ and coinduction.

3 Arbitrary Monotone Operators

In this section we extend the previous algorithm to an algorithm for membership checking in the greatest fixpoint of not necessarily invertible monotone operators, where the support of a given goal is the meaning of some positive boolean expression. As mentioned in the introduction, this extension is motivated by the subtyping relation of the RAJA system. In the following we describe formally positive boolean expressions and their meaning.

Definition 3.1. Positive boolean expressions *over G are defined by the grammar*

$$e ::= \mathsf{tt} \mid \mathsf{ff} \mid g \mid e_1 \wedge e_2 \mid e_1 \vee e_2$$

where g ranges over elements of G. Let $\mathsf{PBool}(G)$ *be the set of positive boolean expressions over G.*

Positive boolean expressions denote predicates on $\mathcal{P}(G)$. In particular, g denotes $\{X \mid g \in X\}$. Formally, if $X \subseteq G$ we define the *meaning* $[\![e]\!]^X : \mathsf{bool}$ as follows:

$$\begin{array}{lcl} [\![\mathsf{tt}]\!]^X & = & \mathsf{tt} \\ [\![\mathsf{ff}]\!]^X & = & \mathsf{ff} \\ [\![g]\!]^X & = & g \in X \end{array} \qquad \begin{array}{lcl} [\![e_1 \wedge e_2]\!]^X & = & [\![e_1]\!]^X \wedge [\![e_2]\!]^X \\ [\![e_1 \vee e_2]\!]^X & = & [\![e_1]\!]^X \vee [\![e_2]\!]^X \end{array}$$

Example 3.2. *Let* $G = \{a,b,c,d\}$ *and* $e = a \wedge (b \vee c)$*, then* $[\![e]\!]^{\{a,b\}} = \mathsf{tt}$ *and* $[\![e]\!]^{\{b,c\}} = \mathsf{ff}$.

Note that $X \subseteq Y$ implies $[\![e]\!]^X \Rightarrow [\![e]\!]^Y$.

Definition 3.3. *Let* $f : G \to \mathsf{PBool}(G)$ *be a boolean operator. Then we obtain a monotone operator* $\mathcal{F}_f$ *as follows:*

$$\begin{array}{lcl} \mathcal{F}_f & : & \mathcal{P}(G) \to \mathcal{P}(G) \\ X & \mapsto & \{g \mid [\![f(g)]\!]^X = \mathsf{tt}\} \end{array}$$

Next we prove constructively that, whenever a set G is finite, we can provide a boolean operator for any monotone operator over G. We notice though that the so constructed boolean operator might be very big, hence, applying the algorithm we are about to describe would be very inefficient.

Theorem 3.4. *If G is a finite set and* $\mathcal{F} : \mathcal{P}(G) \to \mathcal{P}(G)$ *then there exists* $f : G \to \mathsf{PBool}(G)$ *such that* $\mathcal{F} = \mathcal{F}_f$.

Proof. For each (finite) subset $X = \{g_1, \ldots, g_k\} \subseteq G$ define $\bigwedge X := g_1 \wedge \ldots \wedge g_k$. We have $[\![\bigwedge X]\!]^Y = \mathsf{tt} \iff X \subseteq Y$. Given g let $X_1 \ldots X_k$ be an enumeration of the subsets X such that $g \in \mathcal{F}(X)$. We then put $f(g) = \bigwedge X_1 \ \vee \ldots \vee \ \bigwedge X_n$. Now $g \in \mathcal{F}(X) \Rightarrow X = X_i$ for some $i \ \Rightarrow [\![\bigwedge X_i]\!]^X = \mathsf{tt} \Rightarrow [\![f(g)]\!]^X = \mathsf{tt}$. Conversely $[\![f(g)]\!]^X = \mathsf{tt} \Rightarrow X_i \subseteq X$ for some $i \ \Rightarrow g \in \mathcal{F}(X_i) \Rightarrow g \in \mathcal{F}(X)$ by monotonicity. □

For invertible operators we can provide a boolean operator directly. Given $f : G \to \mathcal{P}(G)$ as in the last section and $A \subseteq G$, define $\tilde{f}$ as follows:

$$\tilde{f}(g) = \begin{cases} \mathsf{ff} & \text{if } g \notin A \\ \bigwedge f(g) & \text{if } g \in A \end{cases}$$

Then $[\![\tilde{f}(g)]\!]^X = \mathsf{tt} \iff g \in A \wedge f(g) \subseteq X$, hence, $\mathcal{F}_{\tilde{f}}(X) = \mathcal{F}_{f,A}(X)$.

Membership checking

Figure 2 shows a new algorithm for membership in the gfp of arbitrary monotone operators whenever a boolean operator $f : G \to \mathsf{PBool}(G)$ is given. *Algorithm 2* takes a set of assumptions U as an argument and returns another set of assumptions and a boolean as a result. The difference to the first algorithm is that if the meaning of the support of a goal is tt, then the new computed set of assumptions will be returned; otherwise it will be dropped. Moreover, ff branches do not lead immediately to rejection. They can lead to a positive answer if combined by "or" with a tt branch. In the following we prove correctness

Algorithm 2. Let $* \in \{\wedge, \vee\}$:

$$
\begin{array}{rcl}
\mathsf{test} & : & \mathsf{PBool}(G) \times \mathcal{P}(G) \to \mathsf{bool} \times \mathcal{P}(G) \\
\mathsf{test}(e_1 * e_2, U) & = & \mathsf{let}\ (b_1, V_1) = \mathsf{test}(e_1, U)\ \mathsf{in} \\
 & & \mathsf{let}\ (b_2, V_2) = \mathsf{test}(e_2, V_1)\ \mathsf{in} \\
 & & (b_1 * b_2, V_2) \\
\mathsf{test}(g, U) & = & \mathsf{if}\ g \in U\ \mathsf{then}\ (\mathsf{tt}, U) \\
 & & \mathsf{else\ let}\ (b, V) = \mathsf{test}(f(g), U \cup \{g\})\ \mathsf{in} \\
 & & \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{tt}, V)\ \mathsf{else}\ (\mathsf{ff}, U)
\end{array}
$$

Figure 2: Algorithm for membership checking in the gfp of arbitrary monotone operators.

and termination of the algorithm. If the basic set is finite the algorithm will terminate and the result will be correct. Otherwise, even if the basic set is infinite, if the computation of the support of a goal do not lead to an infinite chain of new goals, then the algorithm will terminate as well with a correct answer.

Theorem 3.5. *Let* $f : G \to \mathsf{PBool}(G)$ *and* test *defined as above. Let* $\mathcal{N}(U) = \nu X . U \cup \mathcal{F}_f(X)$.

1. *If* $\mathsf{test}(e, U) = (b, V)$ *then* $[\![e]\!]^{\mathcal{N}(U)} = b$ *and* $U \subseteq V \subseteq \mathcal{N}(U)$.
2. *If for each g there exists a finite set S such that* $f(S) \subseteq \mathsf{PBool}(S)$ *and* $g \in S$ *then* $\mathsf{test}(g, \emptyset)$ *terminates.*

Proof. 2. follows using $|S \setminus U|$ as a ranking function. For 1. we induct on the runtime of $\mathsf{test}(e, U)$ and – subordinately – on the structure of e. We note that for all $U \subseteq G$ we have $U \subseteq \mathcal{N}(U)$, $\mathcal{N}(U) = \mathcal{N}(\mathcal{N}(U))$.

Case $e = e_1 * e_2$. Write $(b_1, V_1) = \mathsf{test}(e_1, U)$ and $(b_2, V_2) = \mathsf{test}(e_2, V_1)$.

Inductively, we have $b_1 = [\![e_1]\!]^{\mathcal{N}(U)}$ and $U \subseteq V_1 \subseteq \mathcal{N}(U)$. Therefore, $\mathcal{N}(U) \subseteq \mathcal{N}(V_1) \subseteq \mathcal{N}(\mathcal{N}(U)) = \mathcal{N}(U)$, and thus $\mathcal{N}(V_1) = \mathcal{N}(U)$. It follows that $b_2 = [\![e_2]\!]^{\mathcal{N}(U)}$ and $U \subseteq V_1 \subseteq V_2 \subseteq \mathcal{N}(U)$. The claim then follows.

Case $e = g$.

Case $g \in U$. Then $\mathsf{test}(g, U) = (\mathsf{tt}, U)$ and obviously $[\![g]\!]^{\mathcal{N}(U)} = \mathsf{tt}$ and $U \subseteq \mathcal{N}(U)$.

Case $g \notin U$. Write $(b, V) = \mathsf{test}(f(g), U \cup \{g\})$. Inductively, we have $U \cup \{g\} \subseteq V \subseteq \mathcal{N}(U \cup \{g\})$ and $b = [\![f(g)]\!]^{\mathcal{N}(U \cup \{g\})}$.

We claim that $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. One direction is clear by monotonicity of $\mathcal{N}(.)$. For the other direction we use coinduction with $X = \mathcal{N}(U \cup \{g\})$. To conclude $X \subseteq \mathcal{N}(U)$ we have to prove $X \subseteq U \cup \{h \mid [\![f(h)]\!]^X = \mathsf{tt}\}$ which we now do.

Pick $h \in X = \mathcal{N}(U \cup \{g\})$.

From the definition of $\mathcal{N}(.)$ we get that $h \in U$ or $h = g$ or $[\![f(h)]\!]^X = \mathsf{tt}$. The first and third case immediately yield the desired result. In the second case ($g = h$) we get $[\![f(h)]\!]^X = \mathsf{tt}$ from the induction hypothesis. So we proved $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. The result is now direct from the definitions. □

Corollary 3.6. $\mathsf{test}(e, \emptyset) = (b, \_)$ *iff* $[\![e]\!]^{\mathsf{gfp}(\mathcal{F}_f)} = b$.

4 Applications

In this section we consider a special application of the last algorithm. As we already mentioned we are specially interested in computing subtyping for RAJA types. In the following we give a brief and simplified introduction to the RAJA system and show how to instantiate the generic *Algorithm* 2 to gain a RAJA subtyping algorithm.

RAJA programs are annotated FJEU programs, created with the goal of statically analysing their heap space consumption. An FJEU program $\mathscr{C}$ is a partial finite map from class names to class definitions. Classes contain attributes and methods. The RAJA type system is a refinement of the FJEU type system. A *refined (class) type* consists of a class C and a *view* r and is written C^r. The meaning of views is

given by three maps $\Diamond()$, defining potentials, A, defining views of attributes, and M, defining refined method types. More precisely, $\Diamond() : \text{Class} \times \text{View} \to \mathbb{Q}^+$ assigns each class its potential according to the employed view. Next, $\mathsf{A} : \text{Class} \times \text{View} \times \text{Field} \to \text{View}$ determines the refined types of the fields. Finally, $\mathsf{M} : \text{Class} \times \text{View} \times \text{Method} \to \mathcal{P}(\text{Views of Arguments} \to \text{View of Result})$ assigns refined types to methods. We allow polymorphism in the sense that one method may have more than one (or no) refined typing. For more details and concrete examples we refer to [HJ06, HR09].

Now we describe a simplified version of subtyping for RAJA types. The simplification disregards subclasses and potentials but shows the need for going beyond invertible operators. Let RT be the set of RAJA types. We define a monotone operator $\mathcal{F} : \mathcal{P}(\mathsf{RT} \times \mathsf{RT}) \to \mathcal{P}(\mathsf{RT} \times \mathsf{RT})$ as follows:

$$\begin{array}{rcll} \mathcal{F}(X) & = & \{\,(C^r, D^s) & \mid \quad \forall \text{ attributes } a \;.\; \mathsf{A}(C^r,a) = E^p,\, \mathsf{A}(D^s,a) = E^q \;.\; (E^p,E^q) \in X \\ & & & \quad \forall \text{ methods } \; m \;.\; \forall\, (E_1^{\beta_1},\ldots,E_j^{\beta_j} \to E_0^{\beta_0}) \in \mathsf{M}(D^s,m) \;. \\ & & & \qquad \exists\, (E_1^{\alpha_1},\ldots,E_j^{\alpha_j} \to E_0^{\alpha_0}) \in \mathsf{M}(C^r,m) \;. \\ & & & \qquad\quad (E_1^{\beta_1},E_1^{\alpha_1}) \in X, \ldots, (E_j^{\beta_j},E_j^{\alpha_j}) \in X, (E_0^{\alpha_0},E_0^{\beta_0}) \in X \,\} \end{array}$$

Then $C^r <: D^s \iff (C^r,D^s) \in \nu X.\mathcal{F}(X)$. Now, in order to apply *Algorithm* 2, we define a function $f : \mathsf{RT} \times \mathsf{RT} \to \mathsf{PBool}(\mathsf{RT} \times \mathsf{RT})$ so that $\mathcal{F}(X) = \mathcal{F}_f(X)$:

$$\begin{array}{rcl} f(C^r,D^s) & = & \bigwedge_a (E^p,E^q) \wedge \\ & & \bigwedge_m \bigwedge_{E_1^{\beta_1},\ldots,E_j^{\beta_j} \to E_0^{\beta_0}} \bigvee_{E_1^{\alpha_1},\ldots,E_j^{\alpha_j} \to E_0^{\alpha_0}} (E_1^{\beta_1},E_1^{\alpha_1}) \wedge \ldots \wedge (E_j^{\beta_j},E_j^{\alpha_j}) \wedge (E_0^{\alpha_0},E_0^{\beta_0}) \end{array}$$

5 Conclusions

In this paper we extended the algorithm for membership checking for greatest fixed points described in [Pie02, Chapter 21.6] to a more general version where the support of a given goal is a positive boolean expression. For finite sets this generalization encompasses all monotone operators. Next, we provided a new coinductive correctness proof for both algorithms. Finally, we instantiated the general membership algorithm in order to compute subtyping for RAJA types in a goal-directed way.

We believe that our new algorithm can be useful for computing subtyping for other refinement systems that also provide multiple types to methods. As part of a prototype implementation of the RAJA system the algorithm has been implemented in Ocaml and we work currently in a formalization of its correctness proof in the theorem prover Coq.

Acknowledgments. We acknowledge support by the EU integrated project MOBIUS IST 15905 and by the DFG Graduiertenkolleg 1480 Programm- und Modell-Analyse (PUMA). We also thank Andreas Abel for valuable comments.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *TOPLAS*, 15(4):575–631, 1993.

[BH98] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In *ESOP'06*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[HR09] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *CSL'09*, LNCS. Springer, 2009.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *TOPLAS*, 23(3):396–450, 2001.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[SC05] João Costa Seco and Luís Caires. Subtyping first-class polymorphic components. In *ESOP'05*, volume 3444 of *LNCS*, pages 342–356. Springer, 2005.

REM 2007

Monitoring External Resources in Java MIDP

David Aspinall Patrick Maier[1] Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

Abstract

We present a Java library for mobile phones which tracks and controls at runtime the use of potentially costly resources, such as premium rate text messages. This improves on the existing framework (MIDP — the Mobile Information Device Profile [6]), where for example every text message must be authorised explicitly by the user as it is sent. Our resource management library supports richer protocols, like advance reservation and bulk messaging, while maintaining the security guarantee that attempted resource abuse is trapped.

Keywords: Runtime Monitoring, Resource Control, Java MIDP, Security.

1 Introduction

Modern mobile phones are powerful computers. Their primary task, providing mobile wireless telephone services, is comparatively losing importance as they are being used for a range of other applications, from personal information managers to web browsers, from media players to games. Most of these applications access the network [2], either because it is integral to their functionality (e. g. web browsers, online games), or because networking is adding desired features (e. g. playing streaming media or synchronising diaries).

The cost of the standard computational resources, like execution time or memory space, is determined solely by the computational device (i. e. the hardware of the mobile phone) itself. The cost of network access, however, is determined by external entities, e. g. the business model of the phone operator, which is why we classify network access as an *external resource*. Moreover, it is a resource the spending of which users generally would like to control tightly because it costs them money. The last point actually goes double: If network access is maliciously exploited it could be very expensive, but even if it is not exploited, users care about each 10p [3], i. e. they want to know the exact cost beforehand.

In MIDP [6], the current standard framework for Java applications on mobile phones, monitoring external resources, like communication via text message, is left to the user, as

[1] Email: pmaier@inf.ed.ac.uk

[2] Refers to the operator's mobile phone network; access to other networks (like the Internet) is routed through this one.

[3] The standard cost of sending a text message in the United Kingdom.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

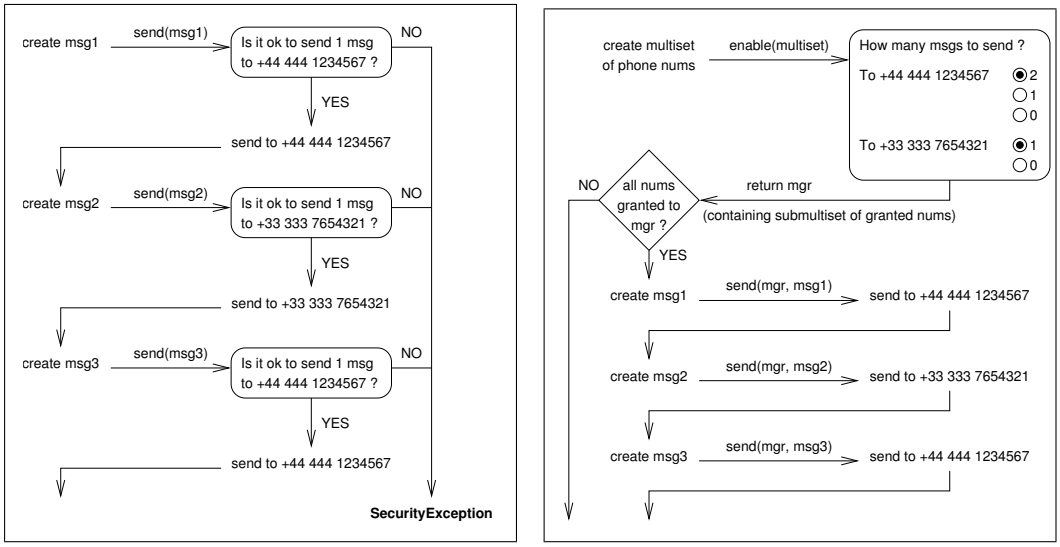


Fig. 1. Transaction sending 3 text messages; in MIDP 2.0 (left) and with explicit resource management (right).

illustrated by the flowchart on the left hand side of Figure 1. For each of the three messages, the application pauses to ask the user for authorisation before sending. This one-shot authorisation is clearly prohibitive for applications wishing to send many messages because users will get annoyed by the many pop-up screens, which malicious applications may exploit to trick users into authorising messages to premium rate numbers. Such social engineering attacks [12] have been reported in the wild [14]. Yet, even if an application sends only few messages, one-shot authorisation can lead to undesirable results, like transactions aborted midway by an exception because the user stops authorising messages (see the left hand side of Figure 1).

We propose explicit accounting and monitoring of external resources to better protect the user from accidental or malicious resource abuse. Our approach revolves around *resource manager* objects, which keep an account of which external resources an application is granted to use and how often. The right hand side of Figure 1 illustrates this on the messaging example. Before sending messages, the application computes a multiset of phone numbers encoding how many messages it will send to which recipients. In a single authorisation dialogue the user then gets to decide how many messages the applications may send to whom. This information (a submultiset of the multiset of requested numbers) is stored in a resource manager. The application only proceeds if all the requested numbers have actually been granted, in which case it calls instrumented methods for sending the messages, taking an extra resource manager argument, which monitors the resources being spent (and would abort the application if it was overspending).

Explicit resource management has additional benefits besides runtime monitoring. It forces the application to determine early on how many resources to request. It provides a clear user interface by centralising the choice of which of the requested resource to grant into a single dialogue. Plus, it enables the application to react flexibly to the amount of resources it has been granted, i. e. the application can choose whether it is feasible to continue with the resources granted or whether it has to abort because of insufficient resources.

The rest of this paper is structured as follows. Section 2 gathers some facts about MIDP which are relevant to us. Section 3 introduces the resource management library,

which Section 4 extends by adding policies. Section 5 describes the security properties that library guarantees and outlines a deployment scenario. Section 6 discusses related work, and Section 7 concludes.

2 Background: The MIDP Security Model

The *Mobile Information Device Profile* (MIDP, current version 2.0 [6]) is the current standard framework for Java applets (also called *MIDlets*) on networked mobile devices. MIDP builds upon the *Connected Limited Device Configuration* (CLDC, current version 1.1 [7]). Together, CLDC and MIDP, which are part of the *Java Micro Edition Platform* (Java ME), define a set of APIs for programming small devices like phones and PDAs. With security in mind, they restrict Java in several ways. In particular, reflection and custom class loading are not supported; all of a MIDlet's classes must be loaded from a single JAR using the standard CLDC class loader, which renders possible to statically check the MIDlet's classes for certain properties (see Section 5.4).

As of MIDP 2.0, access to sensitive APIs and functions (e. g. for sending text messages) is regulated by a permission-based security model. MIDlets are bound to *protection domains* based on whether and by whom they are signed (where a signature expresses the signer's trust in the MIDlet but does not provide any guarantees about the code itself). Each protection domain holds a set of *permissions*, each of which is either flagged as *Allowed* or *User*. The former grants unconditional access whereas the latter requires access to be authorised by the user. How often this authorisation has to be obtained depends on whether a *User* permission is flagged as *Blanket*, *Session* or *OneShot*; the latter requires authorisation for every single access.

According to the MIDP specification, only MIDlets signed by the device manufacturer or the network operator may obtain unconditional access to cost-sensitive functions (e. g. for sending text messages). The protection domains for other MIDlets must insist on OneShot authorisation for access to these functions. As a consequence, MIDlets wishing to use messaging more than just occasionally are faced with the choice of either having to be signed by the operator (or manufacturer) or having to annoy their users with lots of authorisation screens.

3 Basic Resource Management API

This section presents an API for monitoring the use of external resources. The API introduces special objects, called *resource managers*, which encapsulate multisets of resources that a MIDlet may legally use (according to the user's approval) and which are passed as arguments into instrumented MIDP methods that actually use the resources. These methods, e. g. the method for sending text messages, check the resource manager before consuming the resources. If the required resources are not present, the instrumented methods abort the MIDlet with a runtime error.

3.1 Resource Managers

Figure 2 shows a class diagram of resource management package. The core of the API is the final class `ResManager`, which encapsulates a multiset of resources and whose meth-

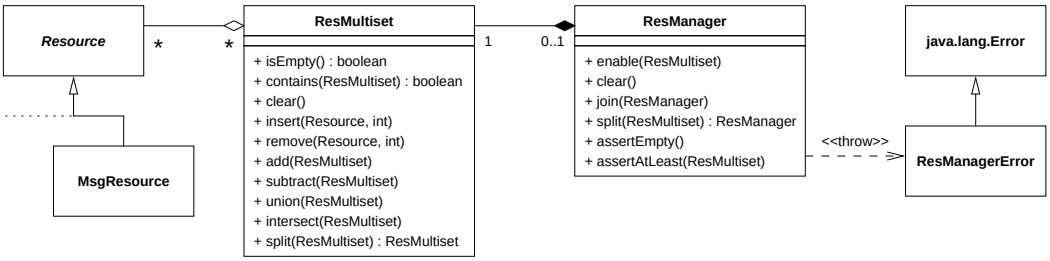


Fig. 2. UML class diagram of the basic resource management API. All terminal (w. r. t. generalisation) classes are final.

ods are explained below. The final class `ResMultiset` provides modifiable multisets of resources, with the usual operations on multisets, including multiset intersection, sum and inclusion. Internally, multisets are realised by hash tables, mapping resources to multiplicities (which may be infinite). Every `ResMultiset` object encapsulates its mutable state, so that it cannot be changed other than by calling its public methods. The abstract class `Resource` serves as an abstract type for resources; actual resources (e. g. the class `MsgResource` representing the permission to send one text message to a given phone number) must be final subclasses. Being used as keys in hash tables, resources must abide by the following contract: They must be immutable objects, and resources constructed from the same arguments must be indistinguishable by the `equals` method.

The class `ResManager` encapsulates a multiset of resources via a private field `rs` of type `ResMultiset`. All public methods are synchronised to avoid races in case different threads access the same resource manager. The table below lists the methods with a JML-style [4] semantics, where the symbols $\subseteq$, $\uplus$ and $\cap$ stand for multiset inclusion, sum and intersection, respectively.

	requires	**ensures**	**modifies**
ResManager()	*true*	this.rs $= \emptyset$	this.rs
void enable(ResMultiset req)	*true*	this.rs $\uplus$ req $=$ \old(this.rs) $\uplus$ \old(req) $\wedge$ req $\subseteq$ \old(req)	this.rs, req
void clear()	*true*	this.rs $= \emptyset$	this.rs
void join(ResManager mgr)	*true*	this.rs $=$ \old(this.rs) $\uplus$ \old(mgr.rs) $\wedge$ mgr.rs $= \emptyset$	this.rs, mgr.rs
ResManager split(ResMultiset bound)	*true*	\fresh(\result) $\wedge$ \result.rs $=$ \old(this.rs) $\cap$ bound $\wedge$ \result.rs $\uplus$ this.rs $=$ \old(this.rs)	this.rs
void assertEmpty()	this.rs $= \emptyset$	*true*	\nothing
void assertAtLeast(ResMultiset bound)	bound $\subseteq$ this.rs	*true*	\nothing

The `enable` method takes a multiset `req` of requested resources and lets the user decide (in a pop-up dialogue) how many of these resources to add to the manager's multiset `rs`. As a side effect, `enable` modifies its argument `req`; upon return from `enable`, the MIDlet should check `req` to learn which of the requested resources it is being denied; in particular, if `req` is empty then all of the requested resources have been granted.

The methods `clear`, `split` and `join` provide some control over the contents of a resource manager, by consuming all its resources, transferring some resources to a new

[4] The \operators generally bear the same meaning as in JML [11], except that `\old(e)` refers to the pre-state of expression `e` in the *pre*-state of the heap.

```
void sendBulk(MessageConnection conn,
              Message msg,
              PhonebookEntry[] grp)
{
  ResMultiset rs = new ResMultiset();
  for (int i=0; i < grp.length; i++) {
    String num = grp[i].getMobileNum();
    rs.insert(new MsgResource(num), 1);
  }

  ResManager mgr = new ResManager();
  mgr.enable(rs);

  if (rs.isEmpty()) {
    for (int i=0; i < grp.length; i++) {
      String num = grp[i].getMobileNum();
      msg.setAddress(num);
      conn.send(mgr, msg);
    }
    mgr.assertEmpty();
  }
  else mgr.clear();
}
```

```
public void send(ResManager mgr, Message msg)
throws IOException, InterruptedIOException
{
  synchorized (msg) {
    String num = msg.getAddress();
    ResMultiset rs = new ResMultiset();
    rs.insert(new MsgResource(num), 1);

    ResManager local_mgr = mgr.split(rs);
    local_mgr.assertAtLeast(rs);

    try {
      send(msg);
      local_mgr.clear(); local_mgr = null;
    } catch (InterruptedIOException e) {
      local_mgr.clear(); local_mgr = null;
      throw e;
    } catch (IOException e) {
      mgr.join(local_mgr); local_mgr = null;
      throw e;
    }
  }
}
```

Fig. 3. Bulk messaging example, left: MIDlet code, right: instrumented MIDP method.

manager, or joining the resources in two managers, respectively. Thanks to `split` and `join`, the MIDlet may keep resource managers thread local, avoiding contention over shared managers.

The assertion methods check whether their preconditions hold. If so they behave like no-ops, otherwise they throw an instance of `ResManagerError`. The latter case must be seen as a violation of the MIDlet's own logic (much like failing an assertion), and the MIDlet should not be allowed attempts at repairing the situation (by catching the error), which is why `ResManagerError` extends `java.lang.Error` rather than `java.lang.Exception`.

3.2 Example: Bulk Messaging MIDlet

We illustrate the use of resource managers by an example application built on top of the *Wireless Messaging API* (WMA, current version 2.0 [8]), a bulk messaging MIDlet, which lets the user send a text message to a group of recipients from his phone book. Figure 3 (left column) shows the MIDlet's method that actually sends the message. The method takes an (already open) message connection, a message and a group of recipients (represented as array of phone book entries). First, the MIDlet builds up a multiset of resources `rs` by iterating over the group of recipients and for each one, extracting the mobile phone number, converting it into a resource by constructing an instance of `MsgResource`, and adding one occurrence of that instance to the multiset. Next, the MIDlet creates an empty resource manager `mgr` and enables it to use the resources in the multiset `rs`. This will pop up a confirmation dialogue box where the user can approve or deny the planned resource usage, modifying `rs` as a side effect. Only if the user approves of all messages to be sent, i. e. if `enable` returns its argument `rs` empty, does the code proceed to the actual send loop. The send loop again iterates over the group of recipients, extracting for each one the mobile phone number, setting the address field of the message and sending the message using the instrumented `send` method, see below. After the loop, `assertEmpty` checks that the resource manager `mgr` is really empty, i. e. all enabled resources have been used. (Instead of checking, the manager could have been cleared explicitly, like in the else branch, to prevent unintended later use of left-over resources.)

3.3 Instrumented Methods

Resources are consumed by specific methods, e. g. in the case of messaging by the method `send(Message)` declared in the WMA interface `MessageConnection`. To monitor whether these methods consume only resources that have been granted, we wrap them with instrumentation code checking whether a given resource manager holds the required resources. These instrumented methods are declared in sub-packages of the resource management package.

To instrument messaging, we have to augment MIDP and WMA in three places. We supplement the WMA interface `MessageConnection` with a new wrapper method `send(ResManager,Message)`, provide a class which implements this extended interface, and revise the MIDP method `Connector.open` to return the new class.

The code for the wrapper method is shown on the right-hand side of Figure 3. It extracts the phone number `num` from the message and constructs a multiset `rs` containing a single occurrence of the resource corresponding to `num`. Then it splits the resources in `rs` off from the resource manager `mgr` and stores them in the new local resource manager `local_mgr`, which is checked for containing at least the resources in `rs`. If this check fails a `ResManagerError` will be thrown, aborting the calling MIDlet; if the check succeeds we know that `local_mgr` holds exactly the resources in `rs`. Finally, the message is actually sent by calling the uninstrumented `send` method. [5] Clearing `local_mgr` and nulling the reference afterwards is not strictly necessary but considered good practise; it signals that the resources in the local manager are now used up and that the manager itself is ready to be reclaimed by garbage collection.

In case of a send failure, the event that actually spends the resources (i. e. delivering the text message to the operator's network) may or may not have happened yet. We assume that an `IOException` is thrown before actually sending the message (e. g. because the connection to the operator's network is down), so the resources are not yet consumed, and the handler can return them to the caller (by joining the local manager to `mgr`) before propagating the exception. However, if an `InterruptedIOException` is raised, we do not know whether the send event has already happened, so we assume that the resources are already spent. In this case, the handler consumes the resources (by clearing the local manager) before propagating the exception.

Note that the instrumented `send` method method must synchronise on `msg`, which is accessed twice, but there is no need to synchronise on `mgr` (for there are no data dependencies between the first and second access) or on **`this`** (for it is accessed only once).

3.4 Runtime Overhead

Monitoring of external resources does cause some runtime overhead. In terms of execution time, the overhead is negligible, as very little time is spent on the instrumentation compared to what is spent on actually consuming the resource (e. g. transmitting a message). Due to the hash table based implementation of multisets, all operations on resource managers take (at most) linear time w. r. t. to the size of the multisets involved. In fact, the overhead of the instrumented send method in Figure 3 is constant because the argument of `assertAtLeast` is a singleton multiset.

[5] Depending on the MIDlet's protection domain, the uninstrumented `send` method may again ask the user to authorise sending the message; Section 5.4 addresses this shortfall.

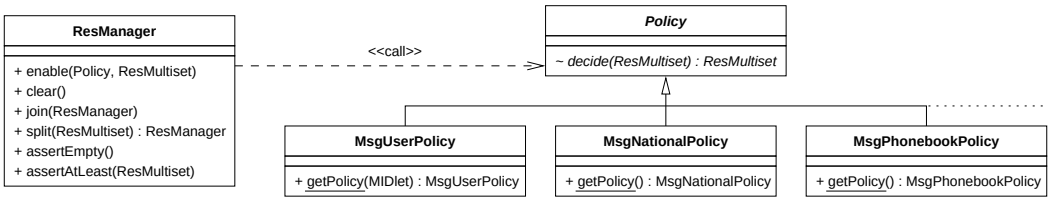


Fig. 4. UML class diagram of policy extension of resource management API. All terminal classes are final.

In terms of memory, the overhead may be more severe, particularly on small devices, because of the memory requirements of the hash tables. Additionally, resource monitoring puts a higher strain on garbage collection because the instrumentation code temporarily allocates resources, multisets and managers. If runtime checking is not necessary or desired, it can be switched off by "erasing" resource managers (see Section 5.2), which reduces the memory overhead significantly.

3.5 Extensibility

By design, the resource management API is extensible. Monitoring new resources (e. g. the number of bytes sent over a TCP/IP connection, or the space available in the persistent record store) simply amounts to adding new resource types plus adding the appropriate instrumentation. New resource types are added by extending the abstract class `Resource` with final subclasses, which abide by the contract on resources. Instrumented methods, which monitor the new resources before calling the corresponding uninstrumented methods, are added to sub-packages of the resource management package.

4 Extending the API with Flexible Policies

So far, the `enable` method involves the user, who is selecting to-be-added resources in a pop-up dialogue. That is, the user is acting as a *policy oracle* deciding which resources to grant and which to deny. In this section, we extend the API to include more flexible policy oracles, not just the user.

4.1 Changes to the API

Figure 4 shows the class diagram of the extension. It adds an abstract class `Policy` providing an abstract, package private method `decide` for deciding which resources to grant and which to deny. The table below shows the formal, non-deterministic semantics of `decide`; granted resources are returned in a new multiset, denied resources are returned via the modified argument.

	requires	**ensures**	**modifies**
ResMultiset decide(ResMultiset req)	$true$	\fresh(\result) $\wedge$ \old(req) = req $\uplus$ \result	req

Actual policies must be final subclasses of `Policy` and must provide a package private implementation of `decide`. The latter requirement ensures that `decide` can be called by the resource management library only, not directly by MIDlets themselves. For a MIDlet to gain access to policies, each subclass of `Policy` provides a static `getPolicy` method

which hands out the requested policy (i. e. an instance of the respective class) or **null** if the calling MIDlet is not authorised to use the requested policy.

MIDlets can only pass policies as arguments to other methods, in particular to the `enable` method of class `ResManager`, which consults its policy argument as an oracle to decide which resources to grant and which to deny, and which interprets a **null** argument as the deny-all policy, see the implementation below. Note that `enable` defers synchronisation on **this** as long as possible (i. e. until accessing the manager's encapsulated multiset `rs`) to avoid locking the manager during a call to `decide`, which may block for a long time (e. g. if the policy consults the user).

```
public void enable(Policy p, ResMultiset req) {
  if (p == null) return;
  synchronized (req) {
    ResMultiset granted = p.decide(req);
    synchronized (this) { rs.add(granted); }
  }
}
```

4.2 *Use of Policies in MIDlets*

The basic resource management API knew only one implicit policy: ask the user. Yet, typically each resource type has its own policy or policies. The policies for `MsgResource` include a `MsgUserPolicy`, which behaves like the implicit policy of the basic API, asking the user how many messages to send to which phone numbers. To use this policy, the call `mgr.enable(rs)` in the bulk messaging MIDlet (Figure 3) must be replaced by `mgr.enable(MsgUserPolicy.getPolicy(this), rs)`.[6]

There could be other policies for `MsgResource`, e. g. a `MsgNationalPolicy`, which grants only messages to national phone numbers. This policy could be combined with `MsgUserPolicy` by chaining calls to `enable` as in the following code snippet.

```
mgr.enable(MsgNationalPolicy.getPolicy(), rs);
mgr.enable(MsgUserPolicy.getPolicy(this), rs);
```

The first call enables all requested messages to national numbers, without asking the user. The second call asks the user to authorise the messages to the remaining (international) numbers. In the end, `rs` contains only those international numbers that the user has denied.

Another interesting policy for messaging could be a `MsgPhonebookPolicy`, which automatically grants all messages to numbers in the user's phone book. If the bulk messaging MIDlet used this policy, the user would not have to confirm anything. In return, the MIDlet could maliciously send more messages than the user intended, but only to phone numbers in the user's phone book, not to premium rate numbers (unless the MIDlet was allowed to modify the phone book).

4.3 *Extensibility*

By design of the API, adding new policies simply amounts to extending the abstract class `Policy` with final subclasses, which abide by the contract on policies: No public fields and methods (in particular, `decide` must be package private) except the static `getPolicy` methods, and the implementation of `decide` must agree with the formal semantics as shown in the table in Section 4.1.

[6] `MsgUserPolicy.getPolicy` requires an argument of type `MIDlet` so that the policy can access the MIDlet's screen.

5 Security Properties of Explicit Resource Management

This section informally summarises and motivates the security guarantees provided by the resource management API and a trusted library implementing it.

5.1 No Abuse of Resources

Property 1 *MIDlets using the resource management API cannot consume more resources than granted; any attempt to do so will result in the MIDlet being aborted before the abuse happens.*

The property holds for two reasons.

(i) Before performing any actions, the instrumented methods, e. g. the `send` method from Section 3.3, check their `ResManager` argument for the required resources and throw a `ResManagerError` (which will abort the MIDlet) if there aren't enough. If there are enough resources, the instrumentation deduces the required amount from the resource manager, even if the underlying uninstrumented method throws an exception.

(ii) The implementation of the resource management API ensures that policies cannot be bypassed. Resources may be moved back and forth between managers by the methods `split` and `join`, but there is no way to sneak new resources into the managers other than by calling `enable`, in which case a policy gets to decide which resources to grant and which to deny. Furthermore, the implementation confines the multiset held by a manager, i. e. it ensures that there are no pointers from outside a manager into its mutable state, hence a manager's multiset cannot be modified from the outside.

Of course, the above argument assumes that the MIDlet does not bypass or subvert the resource management library itself; see Section 5.4 on how to ensure this.

5.2 Erasure

Tracking the use of resources with resource managers does induce some overhead, mainly in terms of the memory required for storing the multisets. On small devices, one might want to avoid this overhead if a MIDlet is known to be *resource safe*, i. e. if it cannot ever throw a `ResManagerError`. In this case, resource managers can be "erased".

Erasure cannot be performed as a simple source code transformation removing all occurrences of resource managers from a MIDlet, for two reasons. First, MIDlets must be able to access resource managers in order to call the `enable` method, even after erasure, to let a policy decide which resources to grant. Second, resource managers may appear in conditions like `(mgr1 == mgr2)`, from where they cannot be removed unless the condition can be evaluated statically. What can be done, however, is a "soft" erasure, which keeps the managers themselves in place but erases their multisets, resulting in very lightweight *erased* resource managers.

Soft erasure can be achieved by retaining the public interface of class `ResManager` but replacing its implementation with a stateless dummy implementation. More precisely, erasure removes the private field `rs` (storing the manager's multiset), which turns all public methods into no-ops, except for `split` and `enable`. The latter still calls the policy and reports the denied resources back to the MIDlet, whereas the former creates a fresh (erased) manager.

Property 2 *If a MIDlet is resource safe then erasing the resource managers does not change its observable behaviour.*

The property holds because by design of the resource management API, the value of a resource manager can only affect the values of other resource managers; it cannot affect the values of other types.

Note that an optimiser can eliminate all of the calls on erased resource managers, except calls to `enable`, by inlining. As a result, resource managers may become unused and can be optimised away. In fact, a clever optimiser could optimise away the entire instrumentation code from the instrumented `send` method in Figure 3, leaving just the call of the uninstrumented method.

5.3 Information Flow Security

It may seem as if resource managers could infringe information flow security. Is it not possible that sensitive data (e. g. phone numbers from the address book) leaks from a manager while it is passed from method to method? We argue that at least for resource safe MIDlets, this is not the case.

Property 3 *If a MIDlet is resource safe then its resource managers do not leak information.*

This is a corollary of Property 2. If a MIDlet is resource safe, the resource managers can be erased without changing the MIDlet's observable behaviour. Yet, erased resource managers are stateless, so they cannot leak information. Hence, no leakage is observable.

5.4 Secure Deployment

As mentioned in Section 5.1, the security guarantees do not only depend on the correctness of the resource management library itself but also on the MIDlet correctly using the API (i. e. not bypassing or subverting the library).

Property 4 *Correct use of the resource management API can be checked statically by inspecting the MIDlet's JAR only.*

The property holds due to the restrictions imposed by CLDC and MIDP (see Section 2), which imply that all of the MIDlet's classes are statically known (since all classes must be loaded from a single JAR) and the signature of each method call is statically known (since reflection is not supported). Thus, the following properties of the MIDlet's class files can be statically checked.

- The MIDlet does not bypass the instrumentation. More precisely, if the MIDlet allocates a particular resource type (e. g. `MsgResource`) then it does not call uninstrumented methods for consuming resources of that type (e. g. the method `send(Message)` declared in the WMA interface `MessageConnection`).
- The MIDlet does not suppress failing assertions. More precisely, it does not catch `ResManagerError` or any of its superclasses.
- The MIDlet does not pass policies of its own to the `enable` method. More precisely, none of the MIDlet's classes extend the abstract class `Policy`.
- The MIDlet does not subvert the implementation of resource multisets by adding re-

source types of its own.[7] More precisely, none of the MIDlet's classes extend the abstract class `Resource`.

- The MIDlet does not exploit non-public methods (e. g. `decide`) of the resource management library. More precisely, none of the MIDlet's classes are declared to be part of the packages that constitute the resource management library.

The correctness of the resource management library itself cannot be checked easily, hence the library (including the instrumented methods) has to be trusted. Yet, as MIDP does not support the download of trusted libraries, MIDlets using the resource management API have to provide the library as part of their own JAR. To establish trust in the library, a trustworthy third party (e. g. the network operator) should vouch for it by signing the MIDlet. In detail, the deployment process should comprise the following steps.

(i) In the MIDlet's JAR, the signer replaces the untrusted resource management library with its own trusted implementation.

(ii) The signer checks for correct use of the resource management API by checking the above properties.

(iii) The signer signs and deploys the MIDlet (possibly after it passed other checks, too).

The signer may choose to erase resource managers by replacing the resource management library with the library for erased managers (see Section 5.2) if there is additional confidence in the MIDlet's resource safety (where this confidence may have been gained by type checking, extended static checking, interactive verification or extensive testing). Of course, Property 1 is not guaranteed by the library for erased managers.

There is a reason, why MIDlets should be signed by the network operator (or device manufacturer) rather than just by any trusted third party. For otherwise, the MIDP specification (see Section 2) demands that the uninstrumented methods which are called by the instrumented ones do still pop-up authorisation screens, despite the fact that the user (or the policy) has already approved all of the resources held by resource managers.

As an alternative deployment scenario, the resource management library could be integrated into future versions of MIDP. In this case, the MIDP class loader would have to check for correct use of the API, rendering unnecessary the requirement that MIDlets be signed by the network operator.

6 Related Work

Runtime monitoring to increase software reliability is at the heart of the Java language [4] with its mandatory runtime checking of array bounds and null pointer dereferences. Several frameworks have been proposed for enhancing Java with runtime monitoring of resource consumption, for example JRes [3], J-Seal [1] and J-RAF [10]. Real-time Java (RTSJ [5]) provides resource monitoring as part of its support for real-time applications. These frameworks monitor specific resources (CPU, memory, network bandwidth, threads), relying on instrumentation of either the JVM (for CPU time), low level system classes (for memory and network bandwidth) and the bytecode itself (for memory and instruction counting). Where our resource management API is designed to enforce security, these frameworks

[7] The hash table based implementation of multisets may fail to function correctly if resources are added that breach the contract that Java imposes on the `equals` and `hashCode` methods.

were developed to support resource aware applications, which can adapt their behaviour in response to resource fluctuation, for example by trading precision for time (by returning an imprecise result to meet a deadline), or time for memory (by caching less to reduce memory consumption).

Runtime monitoring can be used to check whether a program meets a safety property specified in a propositional temporal logic. Tools like JPaX [9] compile a specification into a finite automaton which runs in parallel with the program, observing its behaviour. This kind of temporal specification can express resource protocols like authorise-before-use but is not expressive enough to capture protocols that involve counting potentially unbounded resources.

Schneider [13] advocates a similar use of (not necessarily finite) automata for enforcing security policies at runtime. [15] extends this by allowing an application to query the policy for compliance with a planned sequence of actions. Thus, the application can react gracefully to the policy's decisions; our resource managers provide a similar policy query feature through the `enable` method.

7 Conclusion

We have designed a Java library for tracking and monitoring the use of external resources on MIDP mobile phones (e. g. sending text messages). The library improves the flexibility of runtime monitoring in MIDP (which previously was in the hands of the user), providing a clear user interface and flexible policies while maintaining the security guarantee that any attempt to abuse resources will be trapped.

Our technical contribution is an API for fine-grained accounting of external resources, where fine-grained accounting is achieved by resource managers tracking not just a fixed set of resources but an input-dependent unbounded set (e. g. phone numbers from the user's address book). The API is extensible, admitting to add new resource types and new policies by extending the class hierarchy. Moreover, we have outlined how a trusted library implementing the API can be deployed to MIDP phones as part of a potentially malicious application in such a way that the application cannot subvert the security guarantee (turning the application into a less malicious one). Finally, resource monitoring can be switched off by "erasing" resource managers, which reduces the overhead without changing the observable behaviour of resource safe applications (and we are working on a type system for certifying resource safety [2, chapter 3.3]).

Acknowledgements

This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein. Ian Stark was also supported by an Advanced Research Fellowship from the UK Engineering and Physical Sciences Research Council, EPSRC project GR/R76950/01.

References

[1] Walter Binder, Jarle Hulaas, and Alex Villazón. Portable resource control in Java: The J-SEAL2 approach. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 139–155, 2001.

[2] Mobius Consortium. Deliverable 2.1: Intermediate report on type systems. Available online from http://mobius.inria.fr, September 2006.

[3] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 21–35, 1998.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley Publishing Company, 2005.

[5] JSR 1 Expert Group. JSR 1: Real-time specification for Java. Java specification request, Java Community Process, January 2002.

[6] JSR 118 Expert Group. JSR 118: Mobile information device profile 2.0. Java specification request, Java Community Process, November 2002.

[7] JSR 139 Expert Group. JSR 139: Connected limited device configuration 1.1. Java specification request, Java Community Process, March 2003.

[8] JSR 205 Expert Group. JSR 205: Wireless messaging API 2.0. Java specification request, Java Community Process, June 2004.

[9] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. *Electr.Notes Theor. Comput. Sci.*, 55(2):200–217, 2001.

[10] Jarle Hulaas and Walter Binder. Program transformations for portable CPU accounting and control in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 169–177, 2004.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, July 2007. In Progress. Available from http://www.jmlspecs.org.

[12] Kevin D. Mitnick and William L. Simon. *The Art of Deception*. John Wiley and Sons, Inc., 2002.

[13] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[14] Unknown. Redbrowser.A, February 2006. J2ME trojan. Identified as Redbrowser.A (F-Secure), J2ME/Redbrowser.a (McAfee), Trojan.Redbrowser.A (Symantec), Trojan-SMS.J2ME.Redbrowser.a (Kaspersky Lab).

[15] Dries Vanoverberghe and Frank Piessens. Supporting security monitor-aware development. In *International Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2007.

Safety Guarantees from Explicit Resource Management

David Aspinall, Patrick Maier, and Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh, Scotland
{David.Aspinall,Patrick.Maier,Ian.Stark}@ed.ac.uk

Abstract. We present a language and a program analysis that certifies the safe use of flexible resource management idioms, in particular advance reservation or "block booking" of costly resources. This builds on previous work with *resource managers* that carry out runtime safety checks, by showing how to assist these with compile-time checks. We give a small ANF-style language with explicit resource managers, and introduce a type and effect system that captures their runtime behaviour. In this setting, we identify a notion of *dynamic safety* for running code, and show that dynamically safe code may be executed without runtime checks. We show a similar *static safety* property for type-safe code, and prove that static safety implies dynamic safety. The consequence is that typechecked code can be executed without runtime instrumentation, and is guaranteed to make only appropriate use of resources.

1 Introduction

Safe management of resources is a crucial aspect of software correctness. Bad resource management impacts reliability and security. The more expensive a resource or the more complex its usage pattern, the more important is good management. For example, a media player could crash badly, leaving the hardware in a messy state, if its memory management was governed by the overly optimistic assumption that every request for memory will succeed. Malware on a mobile phone can defraud an unaware user by maliciously sending text messages to premium rate numbers, if there is no effective management of network access [12]. On current mobile platforms such as Java MIDP 2.0, management of network access is commonly left to the user, but users can easily be deceived by social engineering attacks.

Unfortunately, current programming languages do not provide special mechanisms for resource management. Therefore, programmers can only hope that their applications are resource safe, or use necessarily imprecise analyses to try to show this. For example, there are type systems that over-approximate (hopefully tightly) the memory requirements of an application [6], and static analyses that over-approximate the number of text messages being sent by an application [7].

These approaches may fail if a dynamic set of resources must be managed, as with *bulk messaging* where the user wants to send a text message to a number of recipients selected from an address book. Because of the cost of sending text messages, the user must authorise each recipient (i. e., their phone number) explicitly. This could happen individually, just before each message is being sent, or collectively, before sending the

first message. Collective authorisation, or *block booking* of resources, is preferable but requires detailed resource management, keeping track of the (multi-)set of authorised resources – in this case the permitted phone numbers.

In this paper, we present a language-based mechanism that provides programmers with a safe way to control complex resource usage patterns using a notion of *resource manager*. Figure 1 shows the code of a bulk messaging application using resource managers in our intermediate-level functional programming language. The language and functions used will be explained in full detail in Section 2; for now, we just give an outline of operation. The function send_bulk calls send_msgs to send the message msg to the phone numbers stored in the array nums. Along with these two arguments send_msgs takes a resource manager m' which encapsulates the resources that have been authorised (during the call to enable) to send the messages. For each phone number in nums, send_msgs calls the wrapper function send_msg, passing along a resource manager. Prior to calling the primitive send function prim_send_msg, the wrapper checks (using assertAtLeast) whether its input manager m contains the resource required to send a message to num; if the resource is not present, the program will abort with a runtime error, otherwise send_msg removes the resource from the manager (using split), and returns the modified manager as m'.

The bulk messaging application is (dynamically) resource safe by construction, as the resource managers will trap attempts to abuse resources. The resource manager abstraction works in tandem with a static analysis, so that programs which can be proved resource safe statically can be treated more efficiently at runtime by removing the dynamic accounting code. In Section 3.2, we prove resource safety statically for the bulk messaging application.

Our contribution is two-fold. In Section 2, we develop a functional programming language for coding complex resource idioms, such as block booking resources in the bulk messaging application. The language is essentially a first-order functional language in administrative normal form (ANF) [10] with a novel type system serving two purposes. First, the type system names input and output parameters of functions and avoids shadowing of previously bound names, thus admitting to view functions as relations (expressed by logical formulae) between their input and output parameters. Second, the language includes a special, linear type for resource managers, where linearity serves as a means of introducing stateful objects into an otherwise pure functional language. Resource managers track what resources a program is allowed to use, and the operational semantics causes the program to go wrong (i. e., abort with a runtime error) as soon as it attempts to abuse resources. This induces a notion of *dynamic resource safety*, which holds if a program never attempts to abuse resources. In this case, accounting is not necessary. As our first result, we show that erasing resource managers does not alter the semantics of dynamically resource safe programs.

Decisions about which resources programs may use are typically guided by *resource policies*. From the point of view of a program, a policy is simply an oracle determining what resources to grant; and we abstract this as a non-deterministic operation on resource managers. This covers many concrete policy mechanisms, both static (e. g., Java-style policy files) or dynamic (e. g., user interaction); see [3] for more on the interaction of resource managers and policies.

```
send_bulk ::
λ let (r) = res_from_nums (nums) in
  let (m) = init () in
  let (m',r') = enable (m,r) in
  let (n) = size (r') in
  if n then let () = consume (m') in
            ret ()
       else let (m'') = send_msgs (msg,nums,m') in
            let (m''') = assertEmpty (m'') in
            let () = consume (m''') in
            ret () :
(msg:str, nums:str[]) → ()

res_from_nums ::
λ let (i) = length (nums) in
  let (r) = empty () in
  let (r') = res_from_nums' (nums,r,i) in
  ret (r') :
(nums:str[]) → (r':res{})

res_from_nums' ::
λ if i then let (i') = sub (i,1) in
            let (num) = read (nums,i') in
            let (c) = fromstr (num) in
            let (r_c) = single (c,1) in
            let (r'') = sum (r, r_c) in
            let (r') = res_from_nums' (nums,r'',i') in
            ret (r')
       else let (r') = id (r) in
            ret (r') :
(nums:str[], r:res{}, i:int) → (r':res{})

send_msgs ::
λ let (i) = length (nums) in
  let (m') = send_msgs' (msg,nums,m,i) in
  ret (m') :
(msg:str, nums:str[], m:mgr) → (m':mgr)

send_msgs' ::
λ if i then let (i') = sub (i,1) in
            let (num) = read (nums,i') in
            let (m'') = send_msg (msg,num,m) in
            let (m') = send_msgs' (msg,nums,m'',i') in
            ret (m')
       else let (m') = id (m) in
            ret (m') :
(msg:str, nums:str[], m:mgr, i:int) → (m':mgr)

send_msg ::
λ let (c) = fromstr (num) in
  let (r) = single (c,1) in
  let (m',m_r) = split (m,r) in
  let (m_r') = assertAtLeast (m_r,r) in
  let () = prim_send_msg (msg,num) in
  let () = consume (m_r') in
  ret (m') :
(msg:str, num:str, m:mgr) → (m':mgr)

prim_send_msg ::
λ . . . :
(msg:str, num:str) → ()
```

Fig. 1. Bulk messaging application.

In Section 3 we present our second contribution, an effect type system for deriving relational approximations of functions. These approximations are expressed as pairs of constraints in a first-order logic, specifying a pre- and postcondition (or rather, state transforming action) of a given function, similar to Hoare type theory [11]; note that the use of logical formulae as effects is the rationale behind choosing a programming language where functions have named input and output parameters. Typability of functions in the effect type system induces a notion of *static resource safety*. As our second result, we prove a soundness theorem stating that static implies dynamic resource safety. As a corollary, we show that resource managers can always be erased from statically resource safe programs. Proofs have been omitted due to lack of space.

2 A Programming Language for Resource Management

We introduce a simple programming language with built-in constructs for handling resource managers. The language is essentially a simply-typed first-order functional language in ANF [10], with the additional features that functions take and return tuples of values, function types name input and output arguments, scoping avoids shadowing, and the type of resource managers enforces a linearity restriction on its values. The first three of these features are related to giving the language a relational appeal: for the purpose of specifying and reasoning logically, functions ought to be viewed as relations

$$
\begin{array}{rcll}
\langle \text{fundecl} \rangle & ::= & \langle \text{prodtype} \rangle \rightarrow \langle \text{prodtype} \rangle & \textit{(built-in function)} \\
& | & \boldsymbol{\lambda} \langle \text{exp} \rangle \,\mathbf{:}\, \langle \text{prodtype} \rangle \rightarrow \langle \text{prodtype} \rangle & \textit{(λ-abstraction)} \\
\langle \text{exp} \rangle & ::= & \mathbf{if}\ \langle \text{val} \rangle\ \mathbf{then}\ \langle \text{exp} \rangle\ \mathbf{else}\ \langle \text{exp} \rangle & \textit{(conditional)} \\
& | & \mathbf{let}\ (\langle \text{var} \rangle,\ldots,\langle \text{var} \rangle) = \langle \text{fun} \rangle\ (\langle \text{val} \rangle,\ldots,\langle \text{val} \rangle)\ \mathbf{in}\ \langle \text{exp} \rangle & \textit{(function call)} \\
& | & \mathbf{ret}\ (\langle \text{var} \rangle,\ldots,\langle \text{var} \rangle) & \textit{(return)} \\
\langle \text{val} \rangle & ::= & \langle \text{const} \rangle \mid \langle \text{var} \rangle & \\
\langle \text{prodtype} \rangle & ::= & (\langle \text{var} \rangle\mathbf{:}\langle \text{type} \rangle,\ldots,\langle \text{var} \rangle\mathbf{:}\langle \text{type} \rangle) & \\
\langle \text{type} \rangle & ::= & \langle \text{datatype} \rangle \mid \mathbf{mgr} & \\
\langle \text{datatype} \rangle & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{str} \mid \mathbf{res} \mid \mathbf{res\{\}} \mid \langle \text{datatype} \rangle [] &
\end{array}
$$

Fig. 2. BNF grammar.

between input and output parameters. The fourth feature is a means of introducing state into a functional language.

The choice for such a language has been inspired by Grail [2], another first-order functional language in ANF. Moreover, Appel [1] argues that ANF, the intermediate language used by many compilers for functional languages, and SSA, the intermediate representation used by most compilers for imperative languages, are essentially the same thing. Therefore, our language should capture the essence of first-order programming languages, whether functional or imperative.

2.1 Syntax and Static Semantics

Grammar. Figure 2 shows the grammar of the programming language. The nonterminals ⟨fun⟩, ⟨var⟩ and ⟨const⟩ represent *functions*, *variables* and *constants*, respectively. A *program* Π is a partial function from ⟨fun⟩ to ⟨fundecl⟩, i. e., Π maps functions to function declarations, which are either type declarations for built-in functions or λ-abstractions (with type annotations serving as variable binders). We use the notation $\Pi(f) = [\boldsymbol{\lambda}\ldots]\sigma \rightarrow \sigma'$ if we are only interested in the type of f, regardless whether f is built-in or a λ-abstraction. By $dom(\Pi)$, we denote the domain of Π. We denote the restriction of Π to the built-in functions by Π_0, i. e., $\Pi(f)$ is a λ-abstraction if and only if $f \in dom(\Pi) \setminus dom(\Pi_0)$. We assume that Π_0 declares exactly the functions that are shown in Figure 4.

The grammar of *expressions* $e \in$ ⟨exp⟩ and *values* $v \in$ ⟨val⟩ is quite standard for a first-order functional language in ANF. Throughout, functions operate on tuples of values, which is reflected by the syntax for function call and return. The sets of free and bound (by the let-construct) variables of an expression e, denoted by $free(e)$ and $bound(e)$ respectively, are defined in the usual way.

Datatypes $\tau \in$ ⟨datatype⟩ comprise the unit type, integers, strings, resources, multisets of resources, and arrays. A *type* $\tau \in$ ⟨type⟩ is either a datatype or the special type of resource managers, denoted $\mathbf{mgr}$. See Section 2.2 for the interpretations of types. A tuple $(x_1\mathbf{:}\tau_1,\ldots,x_n\mathbf{:}\tau_n) \in$ ⟨prodtype⟩ is a *product type* if the variables $x_1, \ldots, x_n$ are pairwise distinct. Product types appear to associate types to variables,

but they really associate variables *and* types to positions in tuples. A pair of product types of the form $(x_1{:}\tau_1,\ldots,x_m{:}\tau_m) \longrightarrow (x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n)$ forms a *function type* if the variable sets $\{x_1,\ldots,x_m\}$ and $\{x'_1,\ldots,x'_n\}$ are disjoint. We call the product types to the left and right of the arrow *argument type* and *return type*, respectively. As an example consider the type of the function send_msg from Figure 1. It states that send_msg takes two strings and a resource manager and returns a resource manager, while at the same time binding the names of the formal input parameters msg, num and m and announcing that the formal output parameter will be m'.

Static typing. A *type environment* Γ is a functional association list of type declarations of the form $x{:}\tau$, where x is a variable and τ a type. Being functional implies that whenever Γ contains two type declarations $x{:}\tau$ and $x{:}\tau'$ we must have $\tau = \tau'$. Therefore, Γ can be seen as a partial function mapping variables to types. By $dom(\Gamma)$, we denote the domain of this partial function, and for $x \in dom(\Gamma)$, we may write $\Gamma(x)$ for the unique type which Γ associates to x. We write type environments as comma-separated lists, the empty list being denoted by $\emptyset$. The restriction $\Gamma|_X$ of Γ to a set of variables X, is defined in the usual way and induces a partial order $\succeq$ type environments, where $\Gamma' \succeq \Gamma$ iff $\Gamma'|_{dom(\Gamma)} = \Gamma$.

We call a type environment $\Gamma = x_1{:}\tau_1,\ldots,x_n{:}\tau_n$ *linear* if the variables $x_1,\ldots,x_n$ are pairwise distinct. Note that such a linear type environment Γ may be viewed as a product type $\sigma = (x_1{:}\tau_1,\ldots,x_n{:}\tau_n)$, and vice versa. Occasionally, we will write $\Pi(f) = [\boldsymbol{\lambda}\ldots]\Gamma \longrightarrow \Delta$ to emphasise that argument and return types of the function f are to be viewed as linear type environments.

Figure 3 shows the typing rules for the programming language. The judgement $C;\Gamma \vdash v : \tau$ expresses that the value v has type τ in type environment Γ and context C, where a *context* is a set of variables (generally the set of variables occurring in some super-expression of v). Note that (T-const) restricts program constants to the unit value, integers and strings, which are the interpretations of the types $\mathbf{unit}$, $\mathbf{int}$ and $\mathbf{str}$, respectively (see Section 2.2). All other types are abstract in the sense that their values can only be accessed through built-in functions.

The judgement $C;\Gamma \vdash_\Pi e : \sigma$ means that the expression e has product type σ in type environment Γ, context C and program Π. If the program is understood we may write $C;\Gamma \vdash e : \sigma$. There are three things worth noting about expression typing. First, although the type system is linear, weakening and contraction are available to all types but $\mathbf{mgr}$, rendering $\mathbf{mgr}$ the sole linear type of the language. Second, the side condition of (T-let) ensures that let-bound variables do not shadow any variables in the context (which is generally a superset of the set of variables occurring in the let-expression). Third, the rule (T-ret) matches the variables in the return expression to the variables in the product type, thus enforcing that an expression uniformly uses the same variables to return its results (even though these return variables may be let-bound in different branches of the expression). Note that (T-ret) is the only rule to exploit type information about variables. Finally, the judgement $\Gamma \vdash e : \sigma$ (or $\Gamma \vdash_\Pi e : \sigma$ if we want to stress the program Π) means that e has product type σ in a linear type environment Γ.

The judgement $\Pi \vdash f$ states that f is a well-typed λ-abstraction in program Π. Note that the syntax of λ-abstractions does not appear to bind variables, yet it does bind the variables hidden in the argument type. Note also that the restriction on function

Typing of values $C; \Gamma \vdash v : \tau$

(T-var) $\dfrac{}{C; x{:}\tau \vdash x : \tau}$ if $x \in C$ (T-const) $\dfrac{}{C; \emptyset \vdash d : \tau}$ if $\begin{cases} d \in \tau \wedge \\ \tau \in \{\mathbf{unit}, \mathbf{int}, \mathbf{str}\} \end{cases}$

Typing of expressions $C; \Gamma \vdash e : \sigma$

(T-weak) $\dfrac{C; \Gamma \vdash e : \sigma}{C; \Gamma, x{:}\tau \vdash e : \sigma}$ if $\begin{cases} x \in C \wedge \\ \tau \neq \mathbf{mgr} \end{cases}$ (T-contr) $\dfrac{C; \Gamma, x{:}\tau, x{:}\tau \vdash e : \sigma}{C; \Gamma, x{:}\tau \vdash e : \sigma}$ if $\tau \neq \mathbf{mgr}$

(T-if) $\dfrac{C; \Gamma \vdash v : \mathbf{int} \qquad C; \Gamma' \vdash e_1 : \sigma \qquad C; \Gamma' \vdash e_2 : \sigma}{C; \Gamma, \Gamma' \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \sigma}$ (T-xch) $\dfrac{C; \Gamma, \Gamma' \vdash e : \sigma}{C; \Gamma', \Gamma \vdash e : \sigma}$

(T-ret) $\dfrac{C; \Gamma_1 \vdash x_1 : \tau_1 \quad \ldots \quad C; \Gamma_n \vdash x_n : \tau_n}{C; \Gamma_1, \ldots, \Gamma_n \vdash \mathbf{ret}\ (x_1, \ldots, x_n) : (x_1{:}\tau_1, \ldots, x_n{:}\tau_n)}$

(T-let) $\dfrac{\begin{array}{c} \Pi(f) = [\boldsymbol{\lambda} \ldots](z_1{:}\tau_1, \ldots, z_m{:}\tau_m) \to (z'_1{:}\tau'_1, \ldots, z'_n{:}\tau'_n) \\ C; \Gamma_1 \vdash v_1 : \tau_1 \quad \ldots \quad C; \Gamma_n \vdash v_m : \tau_m \\ C \cup \{x'_1, \ldots, x'_n\}; \Gamma', x'_1{:}\tau'_1, \ldots, x'_n{:}\tau'_n \vdash e' : \sigma'' \end{array}}{C; \Gamma_1, \ldots, \Gamma_m, \Gamma' \vdash \mathbf{let}\ (x'_1, \ldots, x'_n) = f\ (v_1, \ldots, v_m)\ \mathbf{in}\ e' : \sigma''}$ if $(*)$

where $(*)$ $\begin{cases} x'_1, \ldots, x'_n \text{ pairwise distinct} \wedge \\ x'_1, \ldots, x'_n \notin C \cup dom(\Gamma') \end{cases}$

Typing of expressions $\Gamma \vdash e : \sigma$

(T-lin) $\dfrac{dom(\Gamma); \Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma}$ if Γ linear

Well-typedness of λ-abstractions $\Pi \vdash f$

(T-lam) $\dfrac{\Pi(f) = \boldsymbol{\lambda} e : (x_1{:}\tau_1, \ldots, x_m{:}\tau_m) \to \sigma' \qquad x_1{:}\tau_1, \ldots, x_m{:}\tau_m \vdash e : \sigma'}{\Pi \vdash f}$

Fig. 3. Typing rules (for a fixed program Π).

types means that the return variables of the body of a λ-abstraction must be disjoint from its argument variables. Finally, we call a program Π *well-typed* if $\Pi \vdash f$ for all $f \in dom(\Pi) \setminus dom(\Pi_0)$.

Lemma 1. *Let e be an expression (referring to an implicit program Π), Γ a type environment and σ a product type.*

1. *If $\Gamma \vdash e : \sigma$ then $free(e) \subseteq dom(\Gamma)$ and $bound(e) \cap dom(\Gamma) = \emptyset$.*
2. *If $\Gamma \vdash e : \sigma$ and $X \supseteq free(e)$ then $\Gamma|_X \vdash e : \sigma$.*

2.2 Interpretation of Types and Effects of Built-in Functions

Constraints. To provide a formal semantics for the built-in functions, we introduce a many-sorted first-order language $\mathcal{L}$ with equality. Sorts of $\mathcal{L}$ are the datatypes of the programming language (note that this excludes the type $\mathbf{mgr}$). Formulae of $\mathcal{L}$ are formed from atomic formulae using the usual Boolean connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ (in decreasing order of precedence), and the quantifiers $\forall x{:}\tau$ and $\exists x{:}\tau$, where $x \in \langle \text{var} \rangle$

is a variable and $\tau \in \langle\text{datatype}\rangle$ a sort. Atomic formulae are the Boolean constants $\top$ and $\bot$, or are constructed from terms using the binary equality predicate $\approx$ (which is available for all sorts), the binary inequality predicate $\leq$ on sort $\mathbf{int}$ or the binary inclusion predicate $\subseteq$ on sort $\mathbf{res\{\}}$. Terms are constructed from variables in $\langle\text{var}\rangle$ and the term constructors, which are introduced below, alongside associating the sorts to specific interpretations.

Sort $\mathbf{unit}$ is interpreted by the one-element set $\{\star\}$. Its only constant is $\star$. There are no function symbols.

Sort $\mathbf{int}$ is interpreted by the integers with infinity. Constants are the integers plus ∞. Function symbols are the usual $- : \mathbf{int} \to \mathbf{int}$ and $+, \cdot, /, \% : \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ (where $/$ and $\%$ denote integer division and remainder, respectively).

Sort $\mathbf{str}$ is interpreted by the set of strings (over some fixed but unspecified alphabet). Constants are all strings. The only function symbol is $++ : \mathbf{str} \times \mathbf{str} \to \mathbf{str}$ (concatenation).

Sort $\mathbf{res}$ is interpreted by an arbitrary infinite set (whose elements are termed *resources*). There are no constants, and $\mathit{fromstr} : \mathbf{str} \to \mathbf{res}$, an embedding of strings into resources, is the only one function symbol.

Sort $\mathbf{res\{\}}$ is interpreted by multisets of resources. It features the constant $\emptyset$ (empty multiset) and the function symbols $\cap, \cup, \uplus : \mathbf{res\{\}} \times \mathbf{res\{\}} \to \mathbf{res\{\}}$ (intersection, union and sum of multisets, respectively), $|\_| : \mathbf{res\{\}} \to \mathbf{int}$ (size of a multiset), $\mathit{count} : \mathbf{res\{\}} \times \mathbf{res} \to \mathbf{int}$ (counting the multiplicity of a resource in a multiset) and $\{\_:\_\} : \mathbf{res} \times \mathbf{int} \to \mathbf{res\{\}}$ (constructing a "singleton" multiset containing a given resource with a given multiplicity and nothing else).

Sort $\tau[]$ is interpreted by integer-indexed arrays of elements of sort τ, where an integer-indexed array is a function from an initial segment of the natural numbers to τ. This sort features the constant null (array of length 0) and the function symbols $\mathit{len} : \tau[] \to \mathbf{int}$ (length of an array), $\_[\_] : \tau[] \times \mathbf{int} \to \tau$ (reading at a given index) and $\_[\_:=\_] : \tau[] \times \mathbf{int} \times \tau \to \tau[]$ (updating a given index with a given value). Note that the values of $a[i]$ and $a[i:=v]$ are generally unspecified if the index i is out of bounds (i. e., $i < 0$ or $i \geq \mathit{len}(a)$). As an exception, for $i = \mathit{len}(a)$, the array $a[i:=v]$ properly extends a, i. e., $\mathit{len}(a[i:=v]) = \mathit{len}(a) + 1$. This models vectors that can grow in size.

Treating the type $\mathbf{mgr}$ as an alias for the sort $\mathbf{res\{\}}$, type environments can be seen as associating sorts to variables. Given a type environment Γ and constraint $\phi \in \mathcal{L}$, we write $\Gamma \vdash \phi$ if ϕ is well-sorted w. r. t. Γ; note that this entails $\mathit{free}(\phi) \subseteq \mathit{dom}(\Gamma)$, where $\mathit{free}(\phi)$ is the set of free variables in ϕ.

Substitutions. A *substitution* μ maps variables $x \in \langle\text{var}\rangle$ to values $\mu(x) \in \langle\text{val}\rangle$ (which are variables again or constants, not arbitrary terms). We denote the domain of a substitution μ by $\mathit{dom}(\mu)$. Given a type environment Γ, we write $\Gamma\mu$ for the type environment that arises from substituting the variables in Γ according to μ. This is defined recursively: $\emptyset\mu = \emptyset$ and $(\Gamma, x{:}\tau)\mu$ equals $\Gamma\mu, x{:}\tau$ if $x \notin \mathit{dom}(\mu)$, or $\Gamma\mu, \mu(x){:}\tau$ if $\mu(x) \in \langle\text{var}\rangle$, or $\Gamma\mu$ if $\mu(x) \in \langle\text{const}\rangle$. Note that $\Gamma\mu$ need not be linear even if Γ is. Given a formula ϕ such that $\Gamma \vdash \phi$, we write $\phi\mu$ for the formula obtained by substituting the free variables of ϕ according to μ, avoiding capture. Note that $\Gamma \vdash \phi$ implies $\Gamma\mu \vdash \phi\mu$.

Valuations. Let Γ be a type environment. A *Γ-valuation* α maps variables $x \in dom(\Gamma)$ to elements $\alpha(x)$ in the interpretation of the sort $\Gamma(x)$; we call α a *valuation* if we do not care about the particular type environment Γ. We denote the domain of α by $dom(\alpha)$. Note that $dom(\alpha) \subseteq dom(\Gamma)$ but not necessarily $dom(\alpha) = dom(\Gamma)$; we call α a *maximal* Γ-valuation if $dom(\alpha) = dom(\Gamma)$. Given a Γ-valuation α and a set of variables X, we denote the restriction of α to X by $\alpha|_X$; note that $dom(\alpha|_X) = dom(\alpha) \cap X$. Restriction induces a partial order $\succeq$ on Γ-valuations, where $\alpha' \succeq \alpha$ iff $\alpha'|_{dom(\alpha)} = \alpha$. Given n pairwise distinct variables $x_i \in dom(\Gamma)$ and corresponding elements d_i in the interpretation of $\Gamma(x_i)$, we write $\alpha\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ for the Γ-valuation α' that maps the x_i to d_i and all other $x \in dom(\alpha)$ to $\alpha(x)$. In the special case $dom(\alpha) = \emptyset$, we may drop α and simply write $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$.

Entailment. Let $\phi, \psi \in \mathcal{L}$ be constraints such that $\Gamma \vdash \phi$ and $\Gamma \vdash \psi$. Given a Γ-valuation α with $\mathit{free}(\phi) \subseteq dom(\alpha)$, we write $\alpha \models \phi$ if α satisfies ϕ. We write $\models \phi$ if $\alpha \models \phi$ for all Γ-valuations α with $\mathit{free}(\phi) \subseteq dom(\alpha)$, and we write $\phi \models \psi$ if $\alpha \models \phi$ implies $\alpha \models \psi$ for all Γ-valuations α with $\mathit{free}(\phi) \cup \mathit{free}(\psi) \subseteq dom(\alpha)$. Entailment induces a theory $\mathcal{T} = \{\phi \mid \mathit{free}(\phi) = \emptyset \wedge \top \models \phi\}$, with respect to which entailment can be reduced to unsatisfiability. Note that unsatisfiability w. r. t. $\mathcal{T}$ is not even semi-decidable as $\mathcal{T}$ contains Peano arithmetic. Thus for reasoning purposes, we will generally approximate $\mathcal{T}$ by weaker theories.

Effects. Let f be a built-in function with $\Pi(f) = \Gamma \longrightarrow \Delta$ (viewing argument and return types of f as type environments Γ and Δ, respectively.) An *effect* for f is a pair of constraints ϕ and ψ such that $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$. (Note that $\Gamma \longrightarrow \Delta$ being a function type implies $dom(\Gamma) \cap dom(\Delta) = \emptyset$, hence Γ, Δ is a type environment.) We write $\phi \rightarrow \psi$ to denote such an effect, and we call ϕ its *precondition* and ψ its *action*.

An *effect environment* maps the built-in functions $f \in dom(\Pi_0)$ to effects for f. Figure 4 displays the effect environment Θ_0, providing an axiomatic, relational semantics for all $f \in dom(\Pi_0)$. This semantics ties most built-in functions to corresponding logical operators in a straightforward way; note the non-trivial preconditions for division, reading and writing arrays, and constructing singleton multisets. The effects of functions operating on resource managers warrant some explanation.

init returns an empty manager m'.

enable non-deterministically adds some sub-multiset of r to manager m, returning the result in manager m'; the complement of the added multiset is returned in r'. In an implementation [3] the multiset to be added to m would be chosen by some *policy*, perhaps involving security profiles or user input; we use non-determinism to abstractly model such policy mechanisms.

split splits the multiset held by manager m and distributes it to the managers m'_1 and m'_2 such that m'_2 gets the largest possible sub-multiset of r.

join adds the multisets held by managers m_1 and m_2, returning their sum in m'.

consume is an explicit destructor for manager m and all its resources; the linear type system means that calls to **consume** are necessary even if m is known to be empty.

assertEmpty acts as identity on managers, but subject to the precondition that m is empty; it will be treated specially by the programming language semantics.

f	$\Pi_0(f)$	$\Theta_0(f)$
$\mathbf{id}_\tau$	$(x{:}\tau) \longrightarrow (x'{:}\tau)$	$\top \rightarrow x' \approx x$
$\mathbf{eq}_\tau$	$(x_1{:}\tau, x_2{:}\tau) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx 1 \wedge x_1 \approx x_2 \vee i' \approx 0 \wedge x_1 \not\approx x_2$
add	$(i_1{:}\mathbf{int}, i_2{:}\mathbf{int}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx i_1 + i_2$
sub		$\top \rightarrow i' \approx i_1 + (-i_2)$
mul		$\top \rightarrow i' \approx i_1 \cdot i_2$
div		$i_2 \not\approx 0 \rightarrow i' \approx i_1 \,/\, i_2$
mod		$i_2 \not\approx 0 \rightarrow i' \approx i_1 \,\%\, i_2$
leq		$\top \rightarrow i' \approx 1 \wedge i_1 \leq i_2 \vee i' \approx 0 \wedge i_1 \not\leq i_2$
conc	$(w_1{:}\mathbf{str}, w_2{:}\mathbf{str}) \longrightarrow (w'{:}\mathbf{str})$	$\top \rightarrow w' \approx w_1 +\!\!+ w_2$
fromstr	$(w{:}\mathbf{str}) \longrightarrow (c'{:}\mathbf{res})$	$\top \rightarrow c' \approx \mathit{fromstr}(w)$
$\mathbf{null}_\tau$	$() \longrightarrow (a'{:}\tau[])$	$\top \rightarrow a' \approx \mathit{null}$
$\mathbf{length}_\tau$	$(a{:}\tau[]) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \mathit{len}(a)$
$\mathbf{read}_\tau$	$(a{:}\tau[], i{:}\mathbf{int}) \longrightarrow (x'{:}\tau)$	$0 \leq i \wedge i < \mathit{len}(a) \rightarrow x' \approx a[i]$
$\mathbf{write}_\tau$	$(a{:}\tau[], i{:}\mathbf{int}, x{:}\tau) \longrightarrow (a'{:}\tau[])$	$0 \leq i \wedge i \leq \mathit{len}(a) \rightarrow a' \approx a[i{:=}x]$
empty	$() \longrightarrow (r'{:}\mathbf{res}\{\})$	$\top \rightarrow r' \approx \emptyset$
single	$(c{:}\mathbf{res}, i{:}\mathbf{int}) \longrightarrow (r'{:}\mathbf{res}\{\})$	$i \geq 0 \rightarrow r' \approx \{c{:}i\}$
inter	$(r_1{:}\mathbf{res}\{\}, r_2{:}\mathbf{res}\{\}) \longrightarrow (r'{:}\mathbf{res}\{\})$	$\top \rightarrow r' \approx r_1 \cap r_2$
union		$\top \rightarrow r' \approx r_1 \cup r_2$
sum		$\top \rightarrow r' \approx r_1 \uplus r_2$
size	$(r{:}\mathbf{res}\{\}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \lvert r \rvert$
count	$(r{:}\mathbf{res}\{\}, c{:}\mathbf{res}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \mathit{count}(r, c)$
include	$(r_1{:}\mathbf{res}\{\}, r_2{:}\mathbf{res}\{\}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx 1 \wedge r_1 \subseteq r_2 \vee i' \approx 0 \wedge r_1 \not\subseteq r_2$
init	$() \longrightarrow (m'{:}\mathbf{mgr})$	$\top \rightarrow m' \approx \emptyset$
enable	$(m{:}\mathbf{mgr}, r{:}\mathbf{res}\{\}) \longrightarrow (m'{:}\mathbf{mgr}, r'{:}\mathbf{res}\{\})$	$\top \rightarrow r' \subseteq r \wedge m \uplus r \approx m' \uplus r'$
split	$(m{:}\mathbf{mgr}, r{:}\mathbf{res}\{\}) \longrightarrow (m'_1{:}\mathbf{mgr}, m'_2{:}\mathbf{mgr})$	$\top \rightarrow m'_2 \approx m \cap r \wedge m \approx m'_1 \uplus m'_2$
join	$(m_1{:}\mathbf{mgr}, m_2{:}\mathbf{mgr}) \longrightarrow (m'{:}\mathbf{mgr})$	$\top \rightarrow m' \approx m_1 \uplus m_2$
consume	$(m{:}\mathbf{mgr}) \longrightarrow ()$	$\top \rightarrow \top$
assertEmpty	$(m{:}\mathbf{mgr}) \longrightarrow (m'{:}\mathbf{mgr})$	$m \approx \emptyset \rightarrow m' \approx m$
assertAtLeast	$(m{:}\mathbf{mgr}, r{:}\mathbf{res}\{\}) \longrightarrow (m'{:}\mathbf{mgr})$	$r \subseteq m \rightarrow m' \approx m$

Fig. 4. Types and effects of built-in functions. The subscripts τ indicate families of functions indexed by $\tau \in \langle\text{datatype}\rangle$, except for $\mathbf{id}_\tau$, which is indexed by $\tau \in \langle\text{type}\rangle$.

assertAtLeast acts as identity on managers, but subject to the precondition that the manager m contains the multiset r; will be treated specially by the programming language semantics.

To facilitate the presentation of programming language semantics, we capture the logical semantics of effects directly in terms of valuations. Given a built-in function f with $\Pi_0(f) = \Gamma \longrightarrow \Delta$ and $\Theta_0(f) = \phi \rightarrow \psi$, we define $\mathit{Eff}^{\Pi_0}_{\Theta_0}(f)$ to be the set of maximal (Γ, Δ)-valuations such that $\alpha \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f)$ if and only if $\alpha \models \phi \wedge \psi$.

2.3 Small-step Reduction Semantics

We present a stack-based reduction semantics (which is essentially a continuation semantics) for our programming language. We will show that reduction preserves the

resources stored in resource managers, thanks to linearity. Throughout this section, let Π be a fixed well-typed program.

Stacks. We call a tuple $\langle x_1, \ldots, x_n | \alpha, e \rangle$ a *frame* if $x_1, \ldots, x_n$ is a list of pairwise distinct variables, α is a valuation and e is an expression such that

- $dom(\alpha) \cap \{x_1, \ldots, x_n\} = \emptyset$ and
- $dom(\alpha) \subseteq free(e) \subseteq dom(\alpha) \cup \{x_1, \ldots, x_n\}$.

The roles of e (redex) and α (providing values for the free variables of e) should be clear. The x_i are only present if the frame is suspended waiting for a function to return in which case the x_i act as slots for the return values. A *pre-stack* is either $\lightning$ or ϵ or $F :: S$, where F is a frame and S is a pre-stack. (Pre-stacks essentially correspond to continuations in an abstract machine interpreting λ-terms in ANF [10].) A *stack* (or Π-stack if we want to emphasise the program Π) is a pre-stack of the form $\lightning$ or $\langle |\alpha, e\rangle :: S$. We call $\lightning$ the *error stack*. A stack of the form $\langle |\alpha, \mathbf{ret}\ (x_1,\ldots,x_n)\rangle :: \epsilon$ is called *terminal*. If $F :: S$ is a stack then F is its *top frame*.

Reduction. Figure 5 presents the rules generating the reduction relation $\rightsquigarrow_\Pi$ on stacks. We denote the reflexive-transitive closure of $\rightsquigarrow_\Pi$ by $\rightsquigarrow_\Pi^*$. As usual Π may be omitted if it is understood. Note that reduction performs an eager garbage collection in that it deallocates unused variables immediately by restricting the valuation α in the post stack to the free variables of the expression e.

Reduction is deterministic, except for calls to the built-in function $\mathbf{enable}$.

Proposition 2. *For all stacks S_0 there is at most one stack S_1 such that $S_0 \rightsquigarrow S_1$, unless S_0 is of the form $\langle |\alpha, \mathbf{let}\ (m',r') = \mathbf{enable}\ (m,r)\ \mathbf{in}\ e\rangle :: S_0'$.*

Typed stacks. Reduction is untyped since type information is not needed at runtime. However, various properties of reduction are best stated if the type of variables is known. Therefore, we annotate stacks with type environments and conservatively extend reduction to typed stacks.

Given a frame $\langle x_1, \ldots, x_n | \alpha, e\rangle$, we call $\langle x_1, \ldots, x_n | \alpha, e\rangle^\Gamma$ a *typed frame* if Γ is a linear type environment such that

- $dom(\Gamma) = dom(\alpha) \cup \{x_1, \ldots, x_n\}$,
- α is a Γ-valuation, and
- $\Gamma \vdash e : \sigma$ for some product type σ.

A *typed pre-stack* is $\lightning$, or ϵ, or $F :: \epsilon$ where F is a typed frame, or $F :: F' :: S'$ where S' is a typed pre-stack and $F = \langle x_1, \ldots, x_m | \alpha, e\rangle^\Gamma$ and $F' = \langle x_1', \ldots, x_n' | \alpha', e'\rangle^{\Gamma'}$ are typed frames such that $\Gamma \vdash e : (z_1' : \Gamma'(x_1'), \ldots, z_n' : \Gamma'(x_n'))$ for some variables $z_1', \ldots, z_n'$. A *typed stack* is typed pre-stack of the form $\lightning$ or $\langle |\alpha, e\rangle^\Gamma :: S$. Given a typed frame $F = \langle x_1, \ldots, x_n | \alpha, e\rangle^\Gamma$, we denote its underlying frame $\langle x_1, \ldots, x_n | \alpha, e\rangle$ by $F^\natural$. We extend this notation to typed (pre-)stacks, writing $S^\natural$ for the (pre-)stack underlying the typed (pre-)stack S.

The following proposition shows that reduction does not break the invariants maintained by typed stacks.

$$\text{(R-ret)}\ \frac{\alpha'' = \alpha'\{x'_1 \mapsto \alpha(x_1), \ldots, x'_n \mapsto \alpha(x_n)\}}{\langle|\alpha, \mathbf{ret}\ (x_1,\ldots,x_n)\rangle :: \langle x'_1, \ldots, x'_n|\alpha', e'\rangle :: S \ \rightsquigarrow\ \langle|\alpha''|_{free(e')}, e'\rangle :: S}$$

$$\text{(R-let}_1^{\natural})\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \\ \alpha' = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\}\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in\ ret}\ (x'_1,\ldots,x'_n)\rangle :: S \rightsquigarrow \langle|\alpha'|_{free(e)}, e\rangle :: S}$$

$$\text{(R-let}_1)\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad e' \neq \mathbf{ret}\ (x'_1,\ldots,x'_n) \\ \alpha' = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\}\end{array}}{\begin{array}{l}\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \\ \rightsquigarrow\ \langle|\alpha'|_{free(e)}, e\rangle :: \langle x'_1, \ldots, x'_n|\alpha|_{free(e')}, e'\rangle :: S\end{array}}$$

$$\text{(R-let}_2)\ \frac{\begin{array}{c}\Pi_0(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f) \qquad \alpha'_f \succeq \alpha_f \\ \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1), \ldots, x'_n \mapsto \alpha'_f(z'_n)\}\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \ \rightsquigarrow\ \langle|\alpha'|_{free(e')}, e'\rangle :: S}$$

$$\text{(R-let}_2^{\lightning})\ \frac{\begin{array}{c}\Pi_0(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad f \in \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \forall \alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f) : \alpha'_f \not\succeq \alpha_f\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \ \rightsquigarrow\ \lightning}$$

$$\text{(R-if}_1)\ \frac{\alpha(v) \neq 0}{\langle|\alpha, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle :: S \ \rightsquigarrow\ \langle|\alpha|_{free(e_1)}, e_1\rangle :: S}$$

$$\text{(R-if}_2)\ \frac{\alpha(v) = 0}{\langle|\alpha, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle :: S \ \rightsquigarrow\ \langle|\alpha|_{free(e_2)}, e_2\rangle :: S}$$

Fig. 5. Small-step reduction relation $\rightsquigarrow$ (for a fixed program Π). Application of valuations α extends to values $v \in \langle \text{val} \rangle$ in the natural way, i. e., $\alpha(v) = v$ if v is a constant.

Proposition 3. *Let $\hat{S}_0$ be a typed stack and S_1 a stack. If $\hat{S}_0^{\natural} \rightsquigarrow S_1$ then there is a typed stack $\hat{S}_1$ such that $\hat{S}_1^{\natural} = S_1$.*

The proposition justifies the view of reduction on typed stacks as a conservative extension of the reduction relation defined in Figure 5, where reduction on typed stacks is defined by $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ if and only if $\hat{S}_0^{\natural} \rightsquigarrow_\Pi \hat{S}_1^{\natural}$; as usual Π may be omitted if it is understood.

We call a stack S_0 *stuck* if there is no stack S_1 such that $S_0 \rightsquigarrow S_1$, and S_0 is neither terminal nor the error stack. Our next result shows that reduction on typed stacks will get stuck only at calls to built-in functions (other than $\mathbf{assertEmpty}$ and $\mathbf{assertAtLeast}$), and only if the preconditions of these calls fail. As the effects listed in Figure 4 reveal, reduction will get stuck only upon attempts to divide by 0, access arrays out of bounds or construct singleton multisets with negative multiplicity.

Proposition 4. *Let $\hat{S}$ be a typed stack. If $\hat{S}^{\natural}$ is stuck then it is of the form*

$$\langle |\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e' \rangle :: S'\,,$$

$f \in dom(\Pi_0) \setminus \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\}$, and there is no $\alpha'_f \in Eff_{\Theta_0}^{\Pi_0}(f)$ such that $\alpha'_f \succeq \alpha_f$, where α_f is defined as in rule (R-let$_2$).

Preservation of resources. Given a typed frame $F = \langle x_1, \ldots, x_n | \alpha, e \rangle^{\Gamma}$, we define the multiset *res*(F) of *resources* in F by *res*$(F) = \biguplus\{\alpha(x) \mid x \in dom(\alpha), \Gamma(x) = \mathbf{mgr}\}$. We extend *res* to typed non-error stacks by defining *res*$(\epsilon) = \emptyset$ and *res*$(F :: S) =$ *res*$(F) \uplus$ *res*(S). Proposition 5 states *resource preservation*: The sum of all resources in the system remains unchanged by reduction, unless the built-in functions $\mathbf{enable}$ and $\mathbf{consume}$ are called. The former admits increasing (but not decreasing) the resources, whereas the latter behaves the other way round. Obviously, resource preservation depends on the linearity restriction on type $\mathbf{mgr}$, otherwise resources could be duplicated by re-using managers.

Proposition 5. *Let S_0 and S_1 be typed stacks such that $S_0 \rightsquigarrow S_1 \neq \lightning$.*

1. *If S_0 is of the form $\langle |\alpha, \mathbf{let}\ (m',r') = \mathbf{enable}\ (m,r)\ \mathbf{in}\ e \rangle^{\Gamma} :: S'_0$ then* res$(S_0) \subseteq$ *res*(S_1).
2. *If S_0 is of the form $\langle |\alpha, \mathbf{let}\ () = \mathbf{consume}\ (m)\ \mathbf{in}\ e \rangle^{\Gamma} :: S'_0$ then* res$(S_0) \supseteq$ *res*(S_1).
3. *In all other cases,* res$(S_0) =$ *res*(S_1).

2.4 Erasing Resource Managers

According to the reduction semantics, a call to $\mathbf{assertEmpty}$ or $\mathbf{assertAtLeast}$ either does nothing[1] or goes wrong, and calling one of these two tests is the only way to go wrong. Hence, if we know that a program cannot go wrong (and Section 3 will present a type system for proving just that) then we can erase all calls to these built-ins (or rather, replace them by true no-ops) and obtain an equivalent program.

In fact, we can do more than that. Once the assertion built-ins are gone, it is even possible to remove the resource managers themselves. By the design of the programming language (in particular, the choice of built-in operations on resource managers) the contents of resource managers cannot influence the values of variables of any other type. Informally, this justifies replacing the resource managers themselves by variables of type $\mathbf{unit}$ whenever we know that a program cannot go wrong. Erasing resource managers also means that the built-in functions acting on managers can be replaced by simpler ones on $\mathbf{unit}$: all of which are no-ops, except for $\mathbf{enable}$ itself.[2] The remainder of the section formalises this intuition.

Figure 6 shows the necessary program transformations to erase resource managers. Most fundamentally, erasure maps the manager type $\mathbf{mgr}$ to the unit type $\mathbf{unit}$.

[1] Due to the linearity restriction on resource managers these functions must copy the input manager to an output manager; a true no-op would violate resource preservation.

[2] We do keep the calls in place, so that erasure preserves the structure of programs; this simplifies reasoning, and does not preclude optimising away no-op calls at a later stage.

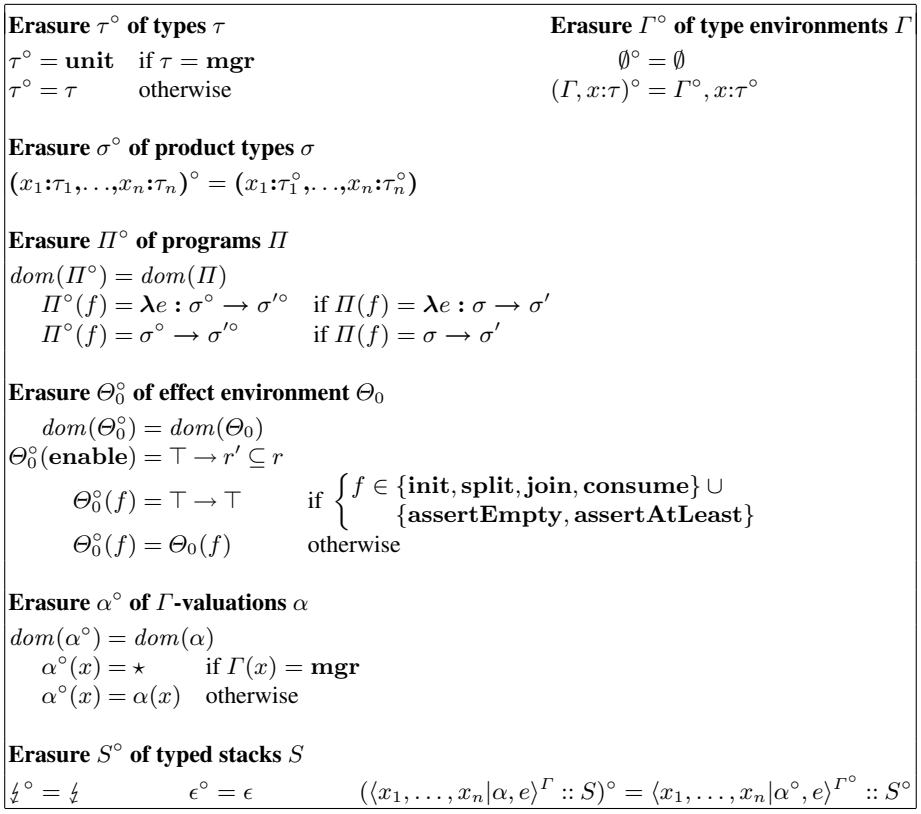


Fig. 6. Erasure of resource managers.

Erasure on types determines erasure on product types, type environments, programs and valuations (where erasure uniformly maps the values of $\mathbf{mgr}$-variables to $\star$, the only value of type $\mathbf{unit}$), which in turn determines erasure on typed stacks. As outlined above, erasure on effect environments trivialises the effect of resource manager built-ins, except $\mathbf{enable}$, and preserves the effects of all built-ins not operating on managers. The effect of $\mathbf{enable}$ after erasure is to non-deterministically choose a sub-multiset of r and return its complement in r'. This reflects the fact that calls to $\mathbf{enable}$ provide points of interaction for the policy (e. g., the user) to decide how many resources the system is granted. Erasing resource managers does not mean that policy decisions are fixed, it just removes the managers' book keeping about those decisions.

Lemma 6. *Let Π be a well-typed program and S a typed Π-stack. Then Π° is a well-typed program and S° a typed Π°-stack.*

Erasure makes trivial the effects of $\mathbf{assertEmpty}$ and $\mathbf{assertAtLeast}$, and in particular, replaces their precondition by $\top$. Thus a program cannot go wrong after erasure, as rule (R-let$^{\lightning}_2$) will never apply.

Proposition 7. *Let Π be a well-typed program and S a Π°-stack S. Then $S \not\rightsquigarrow^*_{\Pi^\circ} \lightning$.*

The next result states that the small-step reduction relation $\rightsquigarrow_\Pi$ of a program Π is almost bisimulation equivalent to the reduction relation $\rightsquigarrow_{\Pi^\circ}$ of its erasure. In fact, it shows that the relation $R = \{\langle S, S^\circ\rangle \mid S \text{ is a } \Pi\text{-stack}\}$ would be a bisimulation if $\rightsquigarrow_\Pi$ could not reduce stacks to the error stack $\lightning$. Put differently, if Π cannot go wrong then $\rightsquigarrow_\Pi$ and $\rightsquigarrow_{\Pi^\circ}$ are bisimulation equivalent. The proof of this theorem is by case analysis on the reduction relation $\rightsquigarrow_\Pi$ of the unerased program. As a corollary, we get that reachability in the erased program is essentially the same as reachability in the unerased one, provided that the unerased program cannot go wrong.

Theorem 8. *Let Π be a well-typed program and $\hat{S}_0$ a typed Π-stack with $\hat{S}_0 \not\rightsquigarrow_\Pi \lightning$.*

1. *For all typed Π-stacks $\hat{S}_1$, if $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ then $\hat{S}_0^\circ \rightsquigarrow_{\Pi^\circ} \hat{S}_1^\circ$.*
2. *For all typed Π°-stacks S_1, if $\hat{S}_0^\circ \rightsquigarrow_{\Pi^\circ} S_1$ then there is a typed Π-stack $\hat{S}_1$ such that $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ and $\hat{S}_1^\circ = S_1$.*

Corollary 9. *Let Π be a well-typed program and S_0 a typed Π-stack. If $S_0 \not\rightsquigarrow^*_\Pi \lightning$ then $\{S^\circ \mid S_0 \rightsquigarrow^*_\Pi S\} = \{S \mid S_0^\circ \rightsquigarrow^*_{\Pi^\circ} S\}$.*

What distinguishes erasure of resource managers from other erasure results (e. g., type erasure during compilation, Java generics erasure) is that here, erasure does not completely remove a language construct. Instead, it removes the book keeping but retains the semantically important bit that deals with dynamic policy decisions.

2.5 Big-step Relational Semantics

The reduction semantics presented in Section 2.3 is good for showing preservation properties, like the preservation of resources. However, it does not easily yield a relational view on functions, relating input and output parameters. This is achieved by a relational semantics, which we will prove equivalent to the reduction semantics. Contrary to the reduction semantics, which was originally untyped and had type environments added conservatively, the relational semantics will be typed from the start. (Types do not hurt here, as the relational semantics is not geared towards execution.)

Throughout this section, we assume that Π is a well-typed program. A *state* β is either the error state $\lightning$ or a normal state $\langle \Gamma; \alpha\rangle$, where Γ is a linear type environment and α a maximal Γ-valuation. Given an expression e, a normal state $\langle \Gamma; \alpha\rangle$ and a state β', we define the judgement $e, \langle \Gamma; \alpha\rangle \Downarrow_\Pi \beta'$ (or $e, \langle \Gamma; \alpha\rangle \Downarrow \beta'$ if Π is understood) by the rules in Figure 7 if $dom(\Gamma) \cap bound(e) = \emptyset$ and there are Γ_e and σ such that $\Gamma \succeq \Gamma_e$ and $\Gamma_e \vdash e : \sigma$. The intended meaning of $e, \langle \Gamma; \alpha\rangle \Downarrow \beta'$ is that evaluating expression e in state $\langle \Gamma; \alpha\rangle$ may terminate and result in state β'.

The reduction semantics deallocates variables once they become unused (an eager garbage collection, so to say), which is essential for the linear variables as otherwise resource preservation would not hold. However, the intermediate values of variables are thus lost. In contrast, the relational semantics names and records all intermediate values, even the linear ones, as $e, \langle \Gamma; \alpha\rangle \Downarrow \langle \Gamma'; \alpha'\rangle$ implies $\Gamma' \succeq \Gamma$ and $\alpha' \succeq \alpha$.

By definition, violations of resource safety manifest themselves in reductions ending in the error stack, and hence reductions which diverge or get stuck cannot

Evaluation of expressions $e, \langle \Gamma; \alpha \rangle \Downarrow \beta'$

$$\text{(E-ret)}\ \frac{}{\mathbf{ret}\ (x_1,\ldots,x_n), \langle \Gamma; \alpha \rangle \Downarrow \langle \Gamma; \alpha \rangle}$$

$$\text{(E-let}_1\text{)}\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \qquad \Gamma_f = z_1{:}\tau_1,\ldots,z_m{:}\tau_m \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad e, \langle \Gamma_f; \alpha_f \rangle \Downarrow \langle \Gamma'_f; \alpha'_f \rangle \\ \Gamma' = \Gamma, x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n \qquad \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1),\ldots,x'_n \mapsto \alpha'_f(z'_n)\} \\ e', \langle \Gamma'; \alpha' \rangle \Downarrow \beta''\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \beta''}$$

$$\text{(E-let}_1^{\lightning}\text{)}\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad \Gamma_f = z_1{:}\tau_1,\ldots,z_m{:}\tau_m \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad e, \langle \Gamma_f; \alpha_f \rangle \Downarrow \lightning\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \lightning}$$

$$\text{(E-let}_2\text{)}\ \frac{\begin{array}{c}\Pi(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad \alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f) \qquad \alpha'_f \succeq \alpha_f \\ \Gamma' = \Gamma, x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n \qquad \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1),\ldots,x'_n \mapsto \alpha'_f(z'_n)\} \\ e', \langle \Gamma'; \alpha' \rangle \Downarrow \beta''\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \beta''}$$

$$\text{(E-let}_2^{\lightning}\text{)}\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad f \in \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad \forall \alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f) : \alpha'_f \not\succeq \alpha_f\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \lightning}$$

$$\text{(E-if}_1\text{)}\ \frac{e_1, \langle \Gamma; \alpha \rangle \Downarrow \beta'}{\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \langle \Gamma; \alpha \rangle \Downarrow \beta'}\ \text{if } \alpha(v) \neq 0$$

$$\text{(E-if}_2\text{)}\ \frac{e_2, \langle \Gamma; \alpha \rangle \Downarrow \beta'}{\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \langle \Gamma; \alpha \rangle \Downarrow \beta'}\ \text{if } \alpha(v) = 0$$

Fig. 7. Big-step evaluation relation (for a fixed program Π).

violate resource safety. Therefore, resource safety is not affected by the fact that the relational semantics ignores such reductions. Under this proviso, Proposition 10 shows the equivalence of reduction and relational semantics.

Proposition 10. *Let $\langle \Gamma; \alpha \rangle$ and $\langle \Gamma'; \alpha' \rangle$ be states. Let e be an expression such that $dom(\Gamma) = free(e)$ and $\Gamma \vdash e : \sigma$ for some product type σ. Then*

1. *$e, \langle \Gamma; \alpha \rangle \Downarrow \lightning$ if and only if $\langle |\alpha, e\rangle^{\Gamma} :: \epsilon \rightsquigarrow^* \lightning$, and*
2. *$e, \langle \Gamma; \alpha \rangle \Downarrow \langle \Gamma'; \alpha' \rangle$ if and only if there is a typed stack $\langle |\alpha'', \mathbf{ret}\ (x_1,\ldots,x_n)\rangle^{\Gamma''} :: \epsilon$ such that $\langle |\alpha, e\rangle^{\Gamma} :: \epsilon \rightsquigarrow^* \langle |\alpha'', \mathbf{ret}\ (x_1,\ldots,x_n)\rangle^{\Gamma''} :: \epsilon$ and $\Gamma' \succeq \Gamma''$ and $\alpha' \succeq \alpha''$.*

3 Effect Type System

In this section, we will develop a type system to statically guarantee dynamic resource safety, i. e., the absence of reductions to the error stack $\lightning$. We will do so by annotating

functions with effects and then extending the notion of effect to a judgement on expressions, which we will define by a simple set of typing rules.

3.1 Effect Type System

We extend the notion of effect $\phi \rightarrow \psi$ from built-in functions to λ-abstractions. To be precise, $\phi \rightarrow \psi$ is an *effect* for f if $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$, where $\Pi(f) = [\boldsymbol{\lambda} \ldots] \Gamma \longrightarrow \Delta$, regardless of whether f is built-in or a λ-abstraction. In line with this extension, an *effect environment* Θ maps all functions $f \in dom(\Pi)$ to effects $\Theta(f)$ for f.

In order to derive the effects of λ-abstractions, we generalise effects to effect types for expressions and develop a type system for inductively constructing such effect types. Effects relate input and output parameters of functions by logical formulae. Likewise, effect types shall relate input and output parameters of expressions. Here, the input parameters of an expression are its free variables; the output parameters are those variables that are not free yet but will become free during reduction, i. e., the (let-)bound variables. Formally, an *effect type* $\Gamma; \phi \rightarrow \Delta; \psi$ is a pair of constraints ϕ and ψ together with a pair of type environments Γ and Δ such that $dom(\Gamma) \cap dom(\Delta) = \emptyset$ and $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$. We call ϕ and ψ *precondition* and *action*, and Γ and Δ *input* and *output (parameters)*, respectively. Given an expression e, we say that an effect type $\Gamma; \phi \rightarrow \Delta; \psi$ is an *effect type for* e if $dom(\Gamma) \cap bound(e) = \emptyset$.

We say that an effect type $\Gamma; \phi \rightarrow \Delta; \psi$ is *stronger than* an effect type $\Gamma'; \phi' \rightarrow \Delta'; \psi'$, denoted by $\Gamma; \phi \rightarrow \Delta; \psi \supseteq \Gamma'; \phi' \rightarrow \Delta'; \psi'$, if $\phi' \models \phi$ and $(\phi' \wedge \psi) \models \psi'$, i. e., the stronger effect type $\Gamma; \phi \rightarrow \Delta; \psi$ has a weaker precondition but stronger action. The stronger-than relation $\supseteq$ is a quasi-order, i. e., reflexive and transitive, and induces an equivalence relation on effect types, the *as-strong-as* relation, which we denote by $\equiv$. Note that for every effect type $\Gamma; \phi \rightarrow \Delta; \psi$ is as strong as an effect type $\Gamma'; \phi \rightarrow \Delta'; \psi$ with linear type environments Γ' and Δ'.

Figure 8 presents the typing rules for deriving effect types. There, the judgement $\Theta \vdash_\Pi e : \Gamma; \phi \rightarrow \Delta; \psi$ states that expression e has effect type $\Gamma; \phi \rightarrow \Delta; \psi$ in the context of program Π and effect environment Θ. If Π is understood, we may omit it and write $\Theta \vdash e : \Gamma; \phi \rightarrow \Delta; \psi$ instead. The judgement $\Pi, \Theta \vdash f$ means that the effect type ascribed to a λ-abstraction f by Θ and Π is consistent with the effect type derived for the body of f. We say that Θ is an *admissible* effect environment for a program Π if $\Pi, \Theta \vdash f$ for all λ-abstractions $f \in dom(\Pi) \setminus dom(\Pi_0)$.

Lemma 11. *Let e be an expression, Θ an effect environment (referring to an implicit program Π) and $\Gamma; \phi \rightarrow \Delta; \psi$ an effect type. If $\Theta \vdash e : \Gamma; \phi \rightarrow \Delta; \psi$ then $\Gamma; \phi \rightarrow \Delta; \psi$ is an effect type for e.*

Theorem 12 states soundness of effect typing w. r. t. the big-step relational semantics. The proof is by double induction on the derivation of relational semantics judgements over the derivation of effect type judgements. As a corollary, we get that reduction starting from a state that satisfies the precondition can't go wrong, hence resource managers can be erased. In fact, the untyped reductions in the erased program match exactly the typed reductions in the original program.

Typing of expression effects $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$

$$\text{(ET-weak)}\ \frac{\Theta \vdash e : \Gamma; \phi \to \Delta; \psi}{\Theta \vdash e : \Gamma'; \phi' \to \Delta'; \psi'}\ \text{if}\ \begin{cases} dom(\Gamma') \cap bound(e) = \emptyset \wedge \\ \Gamma; \phi \to \Delta; \psi \supseteq \Gamma'; \phi' \to \Delta'; \psi' \end{cases}$$

$$\text{(ET-ret)}\ \frac{}{\Theta \vdash \mathbf{ret}\ (x_1,\ldots,x_n) : \emptyset; \top \to \emptyset; \top}$$

$$\text{(ET-if)}\ \frac{\Theta \vdash e_1 : \Gamma; v \not\approx 0 \wedge \phi \to \Delta; \psi \qquad \Theta \vdash e_2 : \Gamma; v \approx 0 \wedge \phi \to \Delta; \psi}{\Theta \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \Gamma; \phi \to \Delta; \psi}$$

$$\text{(ET-let)}\ \frac{\begin{array}{c} \Pi(f) = [\boldsymbol{\lambda}\ldots]\Gamma \to \Delta \qquad \Gamma = z_1{:}\tau_1, \ldots, z_m{:}\tau_m \qquad \Delta = z'_1{:}\tau'_1, \ldots, z'_n{:}\tau'_n \\ \Theta(f) = \phi \to \psi \qquad \mu = \{z_1 \mapsto v_1, \ldots, z_m \mapsto v_m, z'_1 \mapsto x'_1, \ldots, z'_n \mapsto x'_n\} \\ \Theta \vdash e' : \Gamma', \Delta'; \phi' \wedge \psi' \to \Delta''; \psi'' \end{array}}{\Theta \vdash \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e' : \Gamma'; \phi' \to \Delta', \Delta''; \psi' \wedge \psi''}\ \text{if}\ (*)$$

$$\text{where}\ (*)\ \begin{cases} dom(\Gamma') \cap \{x'_1, \ldots, x'_n\} = \emptyset \wedge \\ \Gamma\mu; \phi\mu \to \Delta\mu; \psi\mu \supseteq \Gamma'; \phi' \to \Delta'; \psi' \end{cases}$$

Well-typedness of λ-abstraction effects $\Pi, \Theta \vdash f$

$$\text{(ET-lam)}\ \frac{\Pi(f) = \boldsymbol{\lambda} e : \Gamma \to \Delta \qquad \Theta(f) = \phi \to \psi \qquad \Theta \vdash e : \Gamma; \phi \to \Delta; \psi}{\Pi, \Theta \vdash f}$$

Fig. 8. Typing rules for effect types (for a fixed program Π).

Theorem 12. *Let Θ be an admissible effect environment for a well-typed program Π. Let e be an expression and $\Gamma; \phi \to \Delta; \psi$ an effect type such that $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$. Let $\langle \Gamma; \alpha \rangle$ and β' be states such that $e, \langle \Gamma; \alpha \rangle \Downarrow \beta'$ (which implies $\Gamma_e \vdash e : \sigma$ for some Γ_e, σ). If $\alpha \models \phi$ then $\beta' = \langle \Gamma'; \alpha' \rangle$ for some Γ' and α' such that $\alpha' \models \phi \wedge \psi$. (In particular, if $\alpha \models \phi$ then $\beta' \neq \lightning$.)*

Corollary 13. *Let Θ be an admissible effect environment for a well-typed program Π. Let e be an expression and $\Gamma; \phi \to \Delta; \psi$ an effect type such that $\Theta \vdash_\Pi e : \Gamma; \phi \to \Delta; \psi$. Let α be a maximal Γ-valuation, and let $\hat{S}_0 = \langle |\alpha|_{free(e)}, e \rangle^{\Gamma|_{free(e)}} :: \epsilon$ be a typed Π-stack (which implies $\Gamma|_{free(e)} \vdash_\Pi e : \sigma$ for some σ). If $\alpha \models \phi$ then*

1. *$\hat{S}_0 \not\rightsquigarrow^*_\Pi \lightning$ and*
2. *for all (untyped) Π°-stacks S, $\hat{S}_0^{\circ\natural} \rightsquigarrow^*_{\Pi^\circ} S$ if and only if there is a typed Π-stack $\hat{S}$ such that $\hat{S}_0 \rightsquigarrow^*_\Pi \hat{S}$ and $\hat{S}^{\circ\natural} = S$. (In particular, $\hat{S}_0^{\circ\natural} \not\rightsquigarrow^*_{\Pi^\circ} \lightning$.)*

3.2 Example: Bulk Messaging Application

To illustrate the use of the effect type system, we revisit the example from Figure 1. The interesting bits of code are in the functions send_bulk and send_msg.

The function send_bulk first builds up a multiset of resources r by converting the strings representing phone numbers in nums into resources. Next it attempts to authorise the use of all resources by having enable add r to an empty resource manager m. If this

f	$\Theta(f)$
send_bulk	$\top \to \top$
res_from_nums	$\top \to \mathsf{r} \approx bagof(map_{fromstr}(\mathsf{nums}))$
res_from_nums'	$0 \leq \mathsf{i} \leq len(\mathsf{nums}) \wedge \mathsf{r'} \approx bagof(map_{fromstr}(subarray(\mathsf{nums}, \mathsf{i}, len(\mathsf{nums})))) \to \mathsf{r} \approx bagof(map_{fromstr}(\mathsf{nums}))$
send_msgs	$bagof(map_{fromstr}(\mathsf{nums})) \subseteq \mathsf{m} \to \mathsf{m} \approx \mathsf{m'} \uplus bagof(map_{fromstr}(\mathsf{nums}))$
send_msgs'	$0 \leq \mathsf{i} \leq len(\mathsf{nums}) \wedge bagof(map_{fromstr}(subarray(\mathsf{nums}, 0, \mathsf{i}))) \subseteq \mathsf{m} \to \mathsf{m} \approx \mathsf{m'} \uplus bagof(map_{fromstr}(subarray(\mathsf{nums}, 0, \mathsf{i})))$
send_msg	$count(\mathsf{m}, fromstr(\mathsf{num})) \geq 1 \to \mathsf{m} \approx \mathsf{m'} \uplus \{\!\{fromstr(\mathsf{num}){:}1\}\!\}$
prim_send_msg	$\top \to \top$

$\forall a : len(map_{fromstr}(a)) \approx len(a)$
$\forall a \forall i : 0 \leq i < len(a) \Rightarrow map_{fromstr}(a)[i] \approx fromstr(a[i])$

$\forall a \forall j \forall k : 0 \leq j \leq k \leq len(a) \Rightarrow len(subarray(a, j, k)) = k + (-j)$
$\forall a \forall j \forall k \forall i : 0 \leq j \leq k \leq len(a) \wedge 0 \leq i < len(subarray(a, j, k)) \Rightarrow subarray(a, j, k)[i] = a[j + i]$

$\forall a : |bagof(a)| \approx len(a)$
$\forall a : len(a) \approx 1 \Rightarrow bagof(a) \approx \{a[0]{:}1\}$
$\forall a \forall k : 0 \leq k \leq len(a) \Rightarrow bagof(a) \approx bagof(subarray(a, 0, k)) \uplus bagof(subarray(a, k, len(a)))$

Fig. 9. Bulk messaging application: admissible effect environment Θ and axiomatisation of theory extension; for the sake of readability sort information is suppressed in the axioms.

fails, i. e., the multiset r' returned by enable is of non-zero size, send_bulk terminates (after destroying m' and whatever resources it holds).[3] If authorising all resources succeeds, send_bulk calls send_msgs to actually send the messages while checking that the manager m' contains the required resources. After that, send_bulk checks that send_msgs has used up all resources by asserting that the returned manager m'' is empty; failing this assertion will trigger a runtime error. Finally, send_bulk explicitly destroys the empty manager m''' and terminates.

The function send_msg sends one message, checking whether the resource manager m holds the resource required. It does so by converting the string num into a singleton multiset of resources r. Then it splits the manager m into m' and m_r, so that m_r contains at most the resources in r. Next, send_msg asserts that m_r contains at least r; failing this assertion will trigger a runtime error. Succeeding the assertion, send_msg calls the primitive send function, destroys the now used resource by consuming m_r', and returns the remaining resources in the manager m'.

The bulk messaging example is statically resource safe, as witnessed by the admissible effect environment displayed in Figure 9. Of particular interest is the effect $\top \to \top$ ascribed to the main function send_bulk. This least informative effect expresses nothing about the function itself but implies the absence of runtime errors via Corollary 13.

The effects require an extension of the theory $\mathcal{T}$ (see Section 2.2) by three new functions, axiomatised in Figure 9. The function map maps an array of strings to an

[3] A more sophisticated version of the application could deal more gracefully with enable granting only part of the requested resources. This would require more complex code to inspect the multisets r and r' (but not the resource manager m').

array of resources, $subarray$ takes an array and cuts out the sub-array between two given indices, and $bagof$ converts an array of resources to a multiset (containing the same elements with the same multiplicity). Note that the axiomatisation of $bagof$ is not complete[4] but sufficient for our purposes.

Effect type checking, e. g., for checking admissibility of the effect environment Θ from Figure 9, requires checking the side condition of the weakening rule (ET-weak), which involves checking logical entailment w. r. t. to an extension of the theory $\mathcal{T}$. Due to the high undecidability of $\mathcal{T}$, we actually check entailment w. r. t. (an extension of) an approximation of $\mathcal{T}$; in particular, we approximate multiplication and division by uninterpreted functions. For the bulk messaging example, we used an SMT solver [4] that can handle linear integer arithmetic and arrays. We added axioms for multisets and the axioms in Figure 9. Due to an incomplete quantifier instantiation heuristic, we had to instantiate a number of these axioms by hand, yet eventually, the solver was able to prove all the entailments required by the weakening rules.

Even though arising from a single example, we believe that the extension of the theories of multisets and arrays with the functions $subarray$ and $bagof$ is quite generic and could prove useful in many cases.

4 Conclusion

We have presented a programming language with support for complex resource management, close to the standard SSA/ANF forms of compiler intermediate languages [1]. By construction, programs are *dynamically resource safe* in that any attempts to abuse resources are trapped. We have extended the language with an effect type system which guarantees the for well-typed programs no such attempts occur: we have *static resource safety*. In addition, for such programs the bookkeeping required by dynamic resource management can be erased.

Related Work. Many tools and methods have been proposed to assist with resource management at runtime, e.g., in Java, the JRes [9] and J-Seal [8] frameworks. Generally, these aim to enable programs to react to fluctuations of resources caused by an unpredictable environment. Our aim, however is to track the flow of resources through the program, where the environment can influence the availability of resources only at well-understood points of interaction with the program and with clear availability policies. This offers the chance for more precise resource control whose behaviour can be predicted statically.

This paper builds on previous work [3] with a Java library implementing resource managers and focusing on the dynamic aspects of resource management policies. This Java library supports essentially the same operations on resource managers as our functional language, except that state is realised by destructive updates instead of linear types. While [3] does not provide a static analysis to prove static resource safety, it does outline how dynamic accounting could be erased if static resource safety were provable. Our work here shows one way to do just that.

[4] A complete axiomatisation of $bagof$ is possible in the full first-order theory of multisets and arrays but it is much more complicated and unusable in practise.

Our approach is in line with a general trend of providing the programmer with language-based mechanisms for security and additional static analyses (often using type systems) which use these mechanisms. This combination provides a desirable graceful degradation: if static analysis succeeds in proving certain properties, then the program may be optimised without affecting security. Yet, even if the analyses fail the language based mechanisms will enforce the security properties at runtime.

The context of our work is the MOBIUS project [5] on proof-carrying code (PCC) for mobile devices. Our effect type system is very simple and in principle well-suited for a PCC setting where checkers themselves are resource bounded. However, the weakening rule relies on checking logical entailment in a first-order theory, which is undecidable in general. Therefore, a certificate for PCC need not only provide a type derivation tree but also proofs (in some proof system) for the entailment checks in the weakening rule. The development of a suitable such proof system is a topic for further research, as is the investigation of decidable fragments of relevant first-order theories.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein. Ian Stark was also supported by an Advanced Research Fellowship from the UK Engineering and Physical Sciences Research Council, EPSRC project GR/R76950/01.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.
[2] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoret. Comput. Sci.*, 389(3):411–445, 2007.
[3] D. Aspinall, P. Maier, and I. Stark. Monitoring external resources in Java MIDP. *Electron. Notes Theor. Comput. Sci.*, 197:17–30, 2008.
[4] C. Barrett, L. de Moura, and A. Stump. Design and results of the 2nd annual satisfiability modulo theories competition. *Form. Meth. Syst. Des.*, 31(3):221–239, 2007.
[5] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. MOBIUS: Mobility, ubiquity, security. Objectives and progress report. In *Proc. TGC 2006*, LNCS 4661, pp.10–29. Springer, 2007.
[6] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Proc. LPAR 2004*, LNCS 3452, pp.347–362. Springer, 2005.
[7] F. Besson, G. Dufay, and T. P. Jensen. A formal model of access control for mobile interactive devices. In *Proc. ESORICS 2006*, LNCS 4189, pp.110–126. Springer, 2006.
[8] W. Binder, J. Hulaas, and A. Villazón. Portable resource control in Java. In *Proc. OOPSLA 2001*, pp.139–155. ACM, 2001.
[9] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. OOPSLA '98*, pp.21–35. ACM, 1998.
[10] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PLDI '93*, pp.237–247. ACM, 1993.
[11] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. ESOP 2007*, LNCS 4421, pp.189–204. Springer, 2007.
[12] Unknown. Redbrowser.A, Feb. 2006. J2ME trojan, variously identified in the wild as *Redbrowser.A* (F-Secure), *J2ME/Redbrowser.a* (McAfee), *Trojan.Redbrowser.A* (Symantec), *Trojan-SMS.J2ME.Redbrowser.a* (Kaspersky Lab).

Informatics Report EDI-INF-RR-1341

Resource Analysis for Iterative Java Programs via Lattice-point Enumeration in Polytopes

Kenneth MacKenzie
School of Informatics, University of Edinburgh

August 2009

1 Introduction

We give an overview of some aspects of the theory of lattice point enumeration in polyhedra, and give brief description of a compiler which uses these methods to calculate resource bounds for iterative Java programs. We are particularly interested in the question of producing *certified* resource bounds for *mobile* programs, and so our discussion will draw attention to some of the issues that are especially pertinent in this context. For example, it is desirable that it should be possible to express the results of computations compactly, and also that it should be possible to *check* these results quickly, even if the computations performed to obtain them may have been time-consuming.

2 Loops and Geometry

Consider the following loop L which might occur in a C or Java program:

```
for (i=1; i<=9; i++)
    for (j=1; j<=i && j<=7; j++) B
```

If one is interested in (say) the memory usage of a program including this code, then if the block B allocates some memory, an obvious problem is to determine exactly how many times B is executed.

If B does not alter the values of i and j then within B we have the invariants

$$1 \leq i \leq 9$$
$$1 \leq j \leq i$$
$$1 \leq j \leq 7.$$

Considered as inequalities over the real numbers, these define a trapezoidal region P in the (i, j)-plane, and it is easy to see that number of times the block B is executed is equal to $|P \cap \mathbb{Z}^2|$, the number of lattice points[1] within the polygon P (see Figure 1).

[1] i.e. points with integral coordinates.

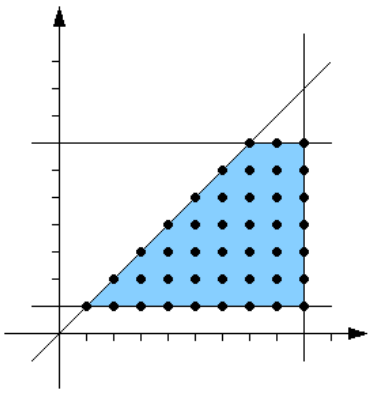


Figure 1: Polygon for loop L

There is a rich mathematical theory of the enumeration of lattice points in polytopes (the generalisation of polygons to higher dimensions) and we will describe some aspects of this theory and its relations to program analysis.

2.1 Linear inequalities and halfspaces

Fix an integer $d \geq 0$ and $a_1, \ldots, a_d \in \mathbb{R}$. We will be interested in solutions $(x_1, \ldots, x_n) \in \mathbb{R}^d$ of inequalities of the form

$$a_1 x_1 + \cdots + a_d x_d \leq b$$

In our applications, such inequalities will arise in the form of linear constraints on program variables.
Consider firstly the equality

$$a_1 x_1 + \cdots + a_d x_d = 0. \tag{1}$$

Writing $\mathbf{a} = (a_1, \ldots, a_d)$ and $\mathbf{x} = (x_1, \ldots, x_d)$ we can rewrite this as $\mathbf{a} \cdot \mathbf{x} = 0$, and this allows us to write the set H of solutions of (1) as

$$H = \{\mathbf{x} \in \mathbb{R}^d : \mathbf{a} \cdot \mathbf{x} = 0\},$$

If $\mathbf{a} \neq \mathbf{0}$ then H is the set of all vectors orthogonal to the vector $\mathbf{a}$, which form a *hyperplane* in $\mathbb{R}^d$, that is a $(d-1)$-dimensional subspace of $\mathbb{R}^d$. In $\mathbb{R}^2$ a hyperplane is a line passing through the origin (Figure 2), and in $\mathbb{R}^3$, a hyperplane is a plane passing through the origin.
Similarly, if $b \in \mathbb{R}$ then the set of solutions to

$$a_1 x_1 + \cdots + a_d x_d = b. \tag{2}$$

is an *affine hyperplane* H', the set of all points at a distance of $b/\|\mathbf{a}\|$ from the hyperplane H (Figure 3).

Finally, the set of solutions to the inequality

$$a_1 x_1 + \cdots + a_d x_d \leq b. \tag{3}$$

describes a *halfspace*, the set of all points on one side of the affine hyperplane H' (Figure 4).

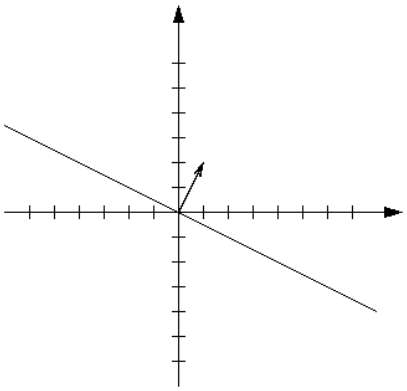


Figure 2: $\{\binom{x}{y} \in \mathbb{R}^2 : \binom{x}{y} \cdot \binom{1}{2} = 0\}$

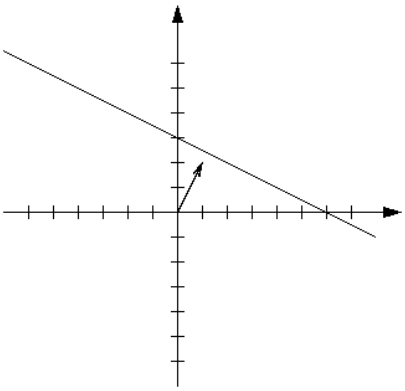


Figure 3: $\{\binom{x}{y} \in \mathbb{R}^2 : \binom{x}{y} \cdot \binom{1}{2} = 6\}$

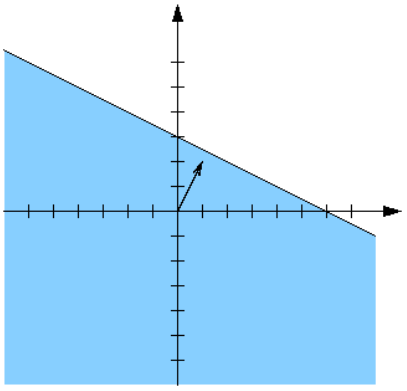


Figure 4: $\{\binom{x}{y} \in \mathbb{R}^2 : \binom{x}{y} \cdot \binom{1}{2} \leq 6\}$

2.2 Polyhedra and Polytopes

In the previous section we have considered the set of solutions to a single linear inequality. We now consider the set of simultaneous solutions to a finite set of such inequalities: geometrically, this is is equal to the intersection of a finite number of halfspaces.

Definition 1. A *convex polyhedron* in $\mathbb{R}^d$ is the intersection of a finite number of halfspaces.

A convex polyhedron is thus the set of solutions to a system of n inequalities

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1d}x_d &\leq b_1 \\ a_{21}x_1 + \cdots + a_{2d}x_d &\leq b_2 \\ &\vdots \\ a_{n1}x_1 + \cdots + a_{nd}x_d &\leq b_n \end{aligned}$$

which may be written more compactly as

$$\mathbf{Ax} \leq \mathbf{b}. \tag{4}$$

where $\mathbf{A}$ lies in $\mathbb{R}^{n\times d}$, the set of $n \times d$ matrices over $\mathbb{R}$. Note that we always use $\leq$; this is not as restrictive as it might at first appear, since the inequality $\mathbf{a} \cdot \mathbf{x} \geq \mathbf{b}$ can be rewritten as $(-\mathbf{a}).\mathbf{x} \leq -\mathbf{b}$, and an equality $\mathbf{a} \cdot \mathbf{x} = \mathbf{b}$ can be written as a conjunction of two inequalities. $\mathbf{a} \cdot \mathbf{x} \geq \mathbf{b}$ and $(-\mathbf{a}) \cdot \mathbf{x} \leq -\mathbf{b}$,

The *dimension* $\dim P$ of a polyhedron P in $\mathbb{R}^d$ is the dimension of the smallest affine subspace containing P. We always have $\dim P \leq d$, and if $\dim P = d$ then P is said to be *full-dimensional*. It should be clear that not all polyhedra are full-dimensional; for example, if we let

$$P = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2 : \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

then we see that P is the 1-dimensional line $x = y$ in $\mathbb{R}^2$.

2.3 Faces of polyhedra

For a 3-dimensional polyhedron, we are familiar with the notion of *vertex*, *edge*, and *face*. For general d, this generalises to the notion of k-*face*, a k-face being a k-dimensional subpolyhedron which is extremal in a suitable sense. Faces of certain dimensions have special names. Given a d-polyhedron P,

- a 0-face is called a *vertex*
- a 1-face is called an *edge*
- a $(d-2)$-face is called a *ridge*
- a $(d-1)$-face is called a *facet*
- P has a single d-face, P itself

The set of faces of P of all dimensions is partially ordered by inclusion, and in fact forms a lattice, called the *face lattice* of P. Two polyhedra are said to be *combinatorially equivalent* if they have isomorphic face lattices. This is the natural definition of what it means for two polyhedra to be of the same general "shape".

Note that this definition of polyhedron is broader than the usual 3-dimensional notion in that there is no requirement that a polyhedron be *bounded* (*ie*, of finite extent; contained in some sphere): for example a halfspace is a polyhedron, as is $\mathbb{R}^d$ itself, the polyhedron determined by an empty set of constraints. The next section will define the notion of a *polytope*, which is perhaps a more natural generalisation of the usual concept of polyhedron.

2.4 Polytopes

Recall that if $\mathbf{p}, \mathbf{q} \in R^d$ then the line segment from $\mathbf{p}$ to $\mathbf{q}$ is

$$l(\mathbf{p}, \mathbf{q}) = \{t\mathbf{p} + (1-t)\mathbf{q} : 0 \leq t \leq 1\},$$

and that a subset X of $\mathbb{R}^d$ is said to be *convex* if whenever $\mathbf{p}, \mathbf{q} \in X$, $l(\mathbf{p}, \mathbf{q}) \subseteq X$. Every set $Y \subseteq \mathbb{R}^d$ is contained in a unique minimal convex set called the *convex hull* of Y.

Definition 2. A *convex polytope* (or just *polytope*) in $\mathbb{R}^d$ is the convex hull of a finite subset $Y \subset \mathbb{R}^d$. The convex hull of a set $Y = \{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ is

$$\operatorname{conv} Y = \left\{ \sum_{i=1}^{m} t_i \mathbf{y}_i : t_i \in \mathbb{R}, t_i \geq 0, \sum_{i=1}^{m} t_i = 1 \right\}.$$

We will sometimes also denote $\operatorname{conv}\{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ by $\operatorname{conv} \mathbf{Y}$, where $\mathbf{Y} \in \mathbb{R}^{d \times m}$ is the $d \times m$ matrix whose columns are the vectors $\mathbf{y}_1, \ldots, \mathbf{y}_m$.

The following theorem [37, Theorem 1.1] may appear obvious, but turns out to be surprisingly difficult to prove.

Theorem 3. *Every bounded convex polyhedron in $\mathbb{R}^d$ is a convex polytope, and vice versa.*

It follows that every convex polytope P has two different descriptions: the *facet representation* (or *halfspace representation*)

$$P = \{\mathbf{x} \in \mathbb{R}^d : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$$

and the *vertex representation*[2]

$$P = \operatorname{conv} \mathbf{Y}$$

Note that if we have a polytope P described by m halfspaces, the number of facets is at most m; however, some halfspaces may be redundant, meaning that the actual number of facets may be strictly less than m. For example, consider the 1-polytope $P = \{x \in \mathbb{R} : x \geq 0, x \leq 1, -2 \leq x \leq 2\}$. This is equal to the closed interval $[-1, 1]$ and the constraint $-2 \leq x \leq 2$ is redundant.
The problem of obtaining a vertex representation from a facet representation is known as the *Vertex Enumeration problem*, and that of obtaining a facet representation from a vertex representation is known as the *Facet Enumeration problem*. We will consider these problems in more detail later.
The general theory of polyhedra has many applications in mathematics and in computer science. See [6] for a survey of CS applications.

3 Lattice-point enumeration

As indicated earlier, we are interested in $|P \cap \mathbb{Z}^d|$, the number of lattice points in a polytope P. This problem becomes very difficult when polytopes whose vertices have irrational coordinates are involved, so we will only consider the following types of polytopes.

Definition 4. A polytope $P = \operatorname{conv}\{\mathbf{y}_1, \ldots \mathbf{y}_n\}$ is said to be *integral* if every coordinate y_{ij} is an integer, and *rational* if every coordinate is a rational number.

Note that even if a polytope P is given in the form $P = \{\mathbf{x} \in \mathbb{R}^d : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ with all a_{ij} and b_i in $\mathbb{Z}$, P will *not* in general be an integral polytope; it will however always be rational.

[2] In fact, there is even an extended version of the vertex description which applies to unbounded polyhedra. This is known as the *Minkowski description*: see [37] for details.

Many applications of polytope methods have been based on the work of Eugène Ehrhart, who studied the problem of how the number of lattice points inside a polytope grows as the size of the polytope increases. More precisely, let

$$P = \text{conv}\{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$$

be a polytope and for $n \in \mathbb{N}$, let

$$nP = \text{conv}\{n\mathbf{y}_1, \ldots, n\mathbf{y}_m\}$$

be the *n-fold dilate* of P. Ehrhart showed that $|nP \cap \mathbb{Z}^d|$ is a *quasipolynomial* in n, which we will now define.

Definition 5. Let $k \in \mathbb{N}$. A *periodic number with period k* is a function $f : \mathbb{Z} \to \mathbb{Z}$ with the property that $f(n + k) = f(n)$ for all $n \in \mathbb{Z}$.

A periodic number with period k is uniquely determined by its value at the points $0, \ldots, k-1$, and this allows us to define a compact notation for periodic numbers.

Definition 6. Let $k \in \mathbb{N}$ and $a_0, \ldots, a_{k-1} \in \mathbb{Z}$. We write

$$[a_0, \ldots, a_{k-1}]_n = \begin{cases} a_0 & \text{if } n \equiv 0 \pmod{k} \\ a_1 & \text{if } n \equiv 1 \pmod{k} \\ \vdots & \\ a_{k-1} & \text{if } n \equiv k-1 \pmod{k}. \end{cases}$$

The function $n \mapsto [a_0, \ldots, a_{k-1}]_n$ is a periodic number with period k, and every periodic number can be written in a similar way.

Definition 7. A *quasipolynomial* (or *pseudopolynomial*) of degree d and quasiperiod k is a function $f : \mathbb{Z} \to \mathbb{Z}$ of the form

$$f(n) = \sum_{i=0}^{d} a_i(n) n^i$$

where each a_i is a periodic number of period k and a_d is not identically zero.

Alternatively, we may write such a quasipolynomial in the form

$$f(n) = \begin{cases} f_0(n) & \text{if } n \equiv 0 \pmod{k} \\ f_1(n) & \text{if } n \equiv 1 \pmod{k} \\ \vdots & \\ f_{k-1}(n) & \text{if } n \equiv k-1 \pmod{k}. \end{cases}$$

where each f_j is a polynomial of the usual kind and $\max\{\deg f_0, \ldots, \deg f_{k-1}\} = d$.

We can now state Ehrhart's Theorem.

Theorem 8 (Ehrhart[17, 18])**.** *Let $P = \text{conv}\{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ be a convex polytope in $\mathbb{Z}^d$ and let*

$$\mathcal{E}_P(n) = |nP \cap \mathbb{Z}^d|$$

- *If P is integral then $\mathcal{E}_P(n)$ is a polynomial of degree* $\dim P$
- *If P is rational then $\mathcal{E}_P(n)$ is a quasipolynomial of degree* $\dim P$ *and quasiperiod equal to the greatest common denominator of the coordinates of the vertices of P.*

The function $\mathcal{E}_P(n)$ is referred to as the *enumerator* of P, or the *Ehrhart (quasi-)polynomial* of P.
As an exercise, we invite the reader to show that the Ehrhart quasipolynomial of the rectangular polytope

$$P = \operatorname{conv}\begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} \end{pmatrix} \subset \mathbb{R}^2$$

is given by

$$\mathcal{E}_P(n) = [1, \tfrac{1}{3}, \tfrac{1}{3}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{1}{6}]_n + [\tfrac{5}{6}, \tfrac{1}{2}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{2}{3}, \tfrac{1}{3}]_n \cdot n + [\tfrac{1}{6}, \tfrac{1}{6}, \tfrac{1}{6}, \tfrac{1}{6}, \tfrac{1}{6}, \tfrac{1}{6}]_n \cdot n^2$$

3.1 Ehrhart polynomials and program analysis

There is a considerable amount of research applying Ehrhart polynomials to program analysis and optimisation, especially in the field of high-performance computing involving array calculations. One of the first papers in this area is due to Clauss [14], where it is stated that the problem of counting the number of solutions to equations derived from affine loop bounds is applicable to problems such as counting the flops executed by a loop, the number of memory locations touched by a loop, the array elements that must be transmitted from one processor to another during parallel array computations, the maximum parallelism induced by a loop from a given time-schedule, and several other problems.

Clauss applies the theory of Ehrhart polynomials to the analysis of loops where the upper bound is parametric. This allows him to deal with situations such as

```
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++) B
```

where the total number of executions is controlled by a single parameter `n`. In this case, increasing the parameter corresponds to dilating the entire associated polytope, which is the situation which Ehrhart polynomials describe.

The case when several independent parameters are involved is more difficult. In this case varying a single parameter corresponds to moving a single facet of the associated polytope while leaving the others in place. For example,

```
for (i=1; i<=n; i++)
    for (j=1; j<=m; j++) B
```

has two parameters `m` and `n`, and the associated polytope is a rectangle. Varying one of the parameters corresponds to changing the width of the rectangle, but not the height (or vice versa). In more complex situations, variation of parameters may lead to a change in the combinatorial type of the associated polytope (see Appendix A).

To deal with this, Clauss uses the work of Loechener and Wilde [25], where it is shown that given a polytope P whose shape and size is controlled by several

parameters, the parameter space can be decomposed into a finite number of regions (called *validity domains*) such that each domain D has an associated quasipolynomial which counts the number of lattice points within the polytope as long as the parameters lie in D; if the parameters move into a different validity domain then you must switch to a different quasipolynomial. Is a later paper Clauss and Loechner [15] extend this work further to produce a method for dealing with the multi-parameter case. This method was purported to be entirely automatic, but this claim was subject to some criticism in [36, §2].

The methods of Clauss seem to have remained largely within the high-performance/parallel computing community (see [24, 33] for example) until 2006, when Braberman et al [11] (and see also [10]) showed how to adapt these techniques to predict the memory usage of (iterative) Java programs; at present this appears to be the only application of polytope methods within the programming language community.

Note that if we restrict to natural numbers, then linear inequalities of the type considered above are exactly the type of inequalities that occur in *Presburger arithmetic*. It follows that the lattice point enumeration problem subsumes the problem of counting solutions to systems of Presburger inequalities. This point of view is examined in greater depth in Pugh[34].

3.2 Drawbacks of Ehrhart polynomials

The standard method used to compute Ehrhart polynomials is *interpolation*, where the coefficients of a polynomial f of degree d are derived from the values of the polynomial at $d+1$ distinct points: this data gives a $(d+1) \times (d+1)$ system of linear equations in the coefficients of f which can then be solved by Gaussian elimination or some other technique. In the case of a quasipolynomial of period k and degree d, this requires us to solve k systems of $(d+1) \times (d+1)$ equations. Recalling that the period k of the Ehrhart polynomial associated with a rational polytope P is the greatest common denominator of the coefficients of the vertices of P, it becomes clear that a considerable amount of computation can be required to calculate $\mathcal{E}_P(n)$. In addition to this, the initial $d+1$ values of the k polynomial components of the quasipolynomial have to be computed by explicitly counting the number of lattice points in the dilates $0P, P, 2P, \ldots, (d+1)P$. The number k can be very large, even for relatively simple polytopes. For example, for the triangular polytope

$$P = \operatorname{conv}\begin{pmatrix} 1/4 & 5/7 & 8/9 \\ 2/5 & 2/11 & 1/2 \end{pmatrix}$$

the quasiperiod of $\mathcal{E}_P(n)$ is 13,680. Calculating the Ehrhart polynomial of P thus requires the solution of 13,680 3×3 systems of linear equations, which would be reasonably time-consuming. In fact, even if the dimension d is fixed, the time taken to compute (via interpolation) the Ehrhart polynomial of a polytope with n vertices can grow exponentially with n (see [36, §2.3]), whereas the methods presented in the next section are polynomial in fixed dimension.

The sheer amount of data required to specify an Ehrhart function is also something of a barrier in the context of certified resource analysis, where such functions would have to be recorded in certificates accompanying mobile programs. This may not in fact be an insurmountable problem. One could possibly

find simpler functions which are upper bounds for the exact Ehrhart function (see [32]); this would save space at the expense of a (hopefully small) loss of precision. Another issue is that Ehrhart functions are not arbitrary quasipolynomials: for example it is clear that they are increasing functions, whereas a general quasipolynomial can have polynomial components which are completely unrelated, leading to a function whose value oscillates drastically. It is conceivable that the quasipolynomials arising as Ehrhart functions have special properties which would enable them to be specified by a relatively small amount of data. Unfortunately, it seems that very little is known about exactly which quasipolynomials can occur as Ehrhart polynomials (see [28, 9] for some partial results) so at present it is difficult to be precise about the minimum of data required to explicitly specify an Ehrhart function. However, the results discussed in the next section may enable us to bypass this problem.

4 Generating functions

The difficulty of computing Ehrhart polynomials suggests that they would be unsuitable for polytope-based analyses in a certifying framework, but fortunately some more recent results provide a much more efficient means of enumerating lattice points. The basic tool in this theory is the *generating function* of a polytope, which is a multivariate polynomial with a term for every lattice point in the polytope. More concretely, suppose we have a polytope P in R^d. We will consider polynomials in the variables $x_1, \ldots, x_d$. Given $\mathbf{v} = (v_1, ..., v_d) \in \mathbb{Z}^d$ we define

$$\mathbf{x}^{\mathbf{v}} = x_1^{v_1} x_2^{v_2} \cdots x_d^{v_d}$$

and the generating function of P is then defined by

$$\mathcal{G}_P(\mathbf{x}) = \sum \{\mathbf{x}^{\mathbf{v}} : \mathbf{v} \in P \cap \mathbb{Z}^d\}$$

It is easy to see that the number of lattice points in P is given by $\mathcal{G}_P(1, \ldots, 1)$. The obvious difficulty here is that the polynomial $\mathcal{G}_P(\mathbf{x})$ will in general be enormous and costly to compute. Recall our earlier example, which gave rise to a trapezoidal region in $\mathbb{R}^2$:

```
for (i=1; i<=9; i++)
    for (j=1; j<=i && j<=7; j++) B
```

For this relatively small example, the full generating function is equal to

$$\begin{aligned} G_P(x,y) = {} & xy + x^2y + x^3y + x^4y + x^5y + x^6y + x^7y + x^8y + x^9y \\ & + x^2y^2 + x^3y^2 + x^4y^2 + x^5y^2 + x^6y^2 + x^7y^2 + x^8y^2 + x^9y^2 \\ & + x^3y^3 + x^4y^3 + x^5y^3 + x^6y^3 + x^7y^3 + x^8y^3 + x^9y^3 \\ & + x^4y^4 + x^5y^4 + x^6y^4 + x^7y^4 + x^8y^4 + x^9y^4 \\ & + x^5y^5 + x^6y^5 + x^7y^5 + x^8y^5 + x^9y^5 \\ & + x^6y^6 + x^7y^6 + x^8y^6 + x^9y^6 \\ & + x^7y^7 + x^8y^7 + x^9y^7 \end{aligned}$$

which is already quite unwieldy.

Fortunately, Alexander Barvinok [7] has recently shown how to express the generating function as a sum of short rational functions which are easily determined from local information at the vertices of P. In the case above, we have

$$\mathcal{G}_P(x,y) = \frac{xy}{(1-x)(1-xy)} + \frac{x^9 y}{(1-x^{-1})(1-y)} + \frac{x^9 y^9}{(1-y^{-1})(1-x^{-1}y^{-1})}$$

Where does this formula come from? Barvinok exploits a theorem of Brion[13] which states that the generating function of a polytope is (up to a certain equivalence relation) the sum of the generating functions of the *supporting cones* at the vertices of the polytope. Informally, to obtain the supporting cone $\mathcal{K}_{\mathbf{v}}$ at a vertex $\mathbf{v}$, take all edges emanating from $\mathbf{v}$, extend them to infinity, and then take the subset of $\mathbb{R}^d$ which they enclose (Figure 5).

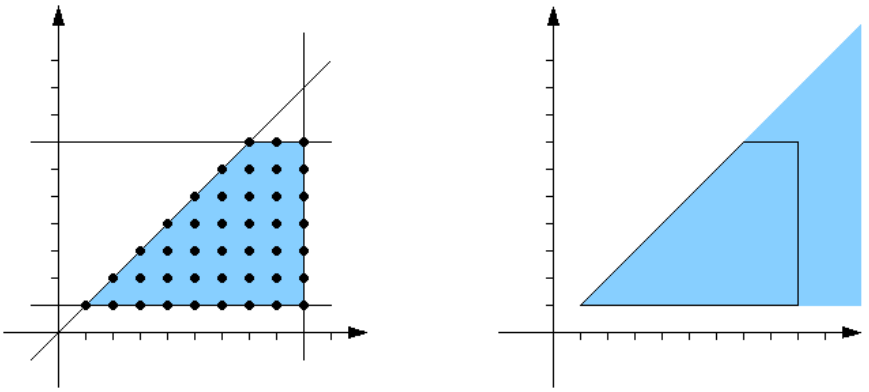


Figure 5: Supporting cone at vertex $\binom{1}{1}$

Note that a supporting cone is always unbounded and hence its generating function has infinitely many terms; however, it can be shown that the generating function is still well-behaved, in the sense that there is some nonempty region D in $\mathbb{R}^d$ such that $\mathcal{G}_{\mathcal{K}_{\mathbf{v}}}(\mathbf{x})$ converges to a finite limit whenever $\mathbf{x} \in D$. Moreover, Barvinok shows that any cone can be decomposed into a signed sum of *unimodular cones*[3] whose generating functions are easy to compute. Once these functions been found they can be combined to give the generating function of the original cone, and Brion's Theorem can then be used to calculate $\mathcal{G}_P(\mathbf{x})$ as a sum of the various $\mathcal{G}_{\mathcal{K}_{\mathbf{v}}}(\mathbf{x})$

It so happens that in our example all of the supporting cones are unimodular, so it's easy to find their generating functions. Consider, for example the vertex $v = (1,1)$ (Figure 5). Here the generating function of the supporting cone is

$$\frac{xy}{(1-x)(1-xy)}.$$

The term xy in the numerator is obtained from the fact that the vertex lies at the point $x = 1, y = 1$, and the numerator comes from the fact that the

[3]See [7] for the definition

supporting cone is generated by the vectors $\binom{1}{0}$ (giving the term $(1-x)$ and $\binom{1}{1}$ (giving the term $(1-xy)$).

Similarly, at the vertex $(9,7)$ the generating function of the supporting cone is Here the generating function of the supporting cone is

$$\frac{x^9y^7}{(1-x^{-1})(1-y^{-1})}.$$

We now obtain the generating function of the entire polytope as the sum

$$\begin{aligned}
\mathcal{G}_P(x,y) &= \frac{xy}{(1-x)(1-xy)} + \frac{x^9y}{(1-x^{-1})(1-y)} \\
&\quad + \frac{x^9y^7}{(1-y^{-1})(1-x^{-1})} + \frac{x^7y^7}{(1-x)(1-x^{-1}y^{-1})} \\
&= \frac{xy}{(1-x)(1-xy)} + \frac{x.x^9y}{x(1-x^{-1})(1-y)} \\
&\quad + \frac{x.y.x^9y^7}{y(1-y^{-1})x(1-x^{-1})} + \frac{xy.x^7y^7}{(1-x)xy(1-x^{-1}y^{-1})} \\
&= \frac{xy}{(1-x)(1-xy)} + \frac{x^{10}y}{(x-1)(1-y)} + \frac{x^{10}y^8}{(y-1)(x-1)} + \frac{x^8y^8}{(1-x)(xy-1)} \\
&= \frac{xy(1-y)}{(1-x)(1-xy)(1-y)} + \frac{x^{10}y(1-xy)}{(x-1)(1-y)(1-xy)} \\
&\quad + \frac{x^{10}y^8(1-xy)}{(y-1)(x-1)(1-xy)} + \frac{x^8y^8(1-y)}{(1-x)(xy-1)(1-y)} \\
&= \frac{xy(1-y) - x^{10}y(1-xy) + x^{10}y^8(1-xy) - x^8y^8(1-y)}{(1-x)(1-y)(1-xy)} \\
&= \frac{xy - xy^2 - x^{10}y + x^{11}y^2 + x^{10}y^8 - x^{11}y^9 - x^8y^8 + x^8y^9}{(1-x)(1-y)(1-xy)}
\end{aligned}$$

To determine the number of lattice points in P we still have to evaluate $\mathcal{G}_P(x,y)$ at the point $(1,1)$, which is complicated by the fact that the denominator of the rational function above vanishes at this point. However, the point (1,1) is in fact a removable singularity of the function, and $\mathcal{G}_P(1,1)$ can easily be determined by calculating the residue at $(1,1)$, which can be done quickly by various methods.

For example, in this case we can remove the singularity by repeatedly applying L'Hôpital's rule[4] by differentiating the numerator and denominator until the denominator is non-zero:

[4]If f and g are continuous at a and $\lim_{x\to a} f(x) = \lim_{x\to a} g(x) = 0$ then $\lim_{x\to a} f(x)/g(x) = \lim_{x\to a} f'(x)/g'(x)$

$$
\begin{aligned}
\left|P \cap \mathbb{Z}^2\right| &= \mathcal{G}_P(1,1) \\
&= \lim_{(x,y)\to(1,1)} \frac{xy - xy^2 - x^{10}y + x^{11}y^2 + x^{10}y^8 - x^{11}y^9 - x^8y^8 + x^8y^9}{(1-x)(1-y)(1-xy)} \\
&= \lim_{(x,y)\to(1,1)} \frac{xy - xy^2 - x^{10}y + x^{11}y^2 + x^{10}y^8 - x^{11}y^9 - x^8y^8 + x^8y^9}{1 - x - y + x^2y + xy^2 - x^2y^2} \\
&= \lim_{(x,y)\to(1,1)} \frac{\frac{\partial}{\partial y}(xy - xy^2 - x^{10}y + x^{11}y^2 + x^{10}y^8 - x^{11}y^9 - x^8y^8 + x^8y^9)}{\frac{\partial}{\partial y}(1 - x - y + x^2y + xy^2 - x^2y^2)} \\
&= \lim_{(x,y)\to(1,1)} \frac{\frac{\partial}{\partial y}(x - 2xy - x^{10} + 2x^{11}y + 8x^{10}y^7 - 9x^{11}y^8 - 8x^8y7 + 9x^8y^8)}{\frac{\partial}{\partial y}(-1 + x^2 + 2xy - 2x^2y)} \\
&= \lim_{(x,y)\to(1,1)} \frac{\frac{\partial}{\partial x}(-2x + 2x^{11} + 56x^{10}y^6 - 72x^{11}y^7 - 56x^8y^6 + 72x^8y^7)}{\frac{\partial}{\partial x}(2x - 2x^2)} \\
&= \lim_{(x,y)\to(1,1)} \frac{(-2 + 22x^{10} + 560^9y^6 - 448x^7y^6 + 576x^7y^7)}{(2 - 4x)} \\
&= \frac{-2 + 22 + 560 - 792 - 448 + 576}{-2} \\
&= 84/2 \\
&= 42,
\end{aligned}
$$

which is indeed equal to the number of lattice points in Figure 1.

This calculation may appear to be quite complex in relation to our relatively small example, but it would be quite easy to automate. Note also that the complexity of the calculation depends only on the shape of the polytope, and not its size. If we took a region of a similar shape but many times larger, all that would change would be the exponents of x and y in the numerator of the generating function; the calculation required to determine the number of lattice points would be essentially identical to that above.

We have only considered Barvinok's construction for integral polytopes here, but the theory can be extended to rational polytopes as well. it is also possible to recover most of the theory of Ehrhart polynomials as well, which is useful for the study of parametric bounds. This approach is developed detail by De Loera et al in [26], which describes the implementation of Barvinok's techniques in the `LattE` package. De Loera's work is applied to program analysis problems in [36], where much of Clauss' work is recast in terms of Barvinok's methods. See also [8] for an exposition of the Barvinok theory, along with a lot of other interesting material.

5 Vertex Enumeration

With luck, the discussion above should have persuaded the reader that Barvinok's method can be used to calculate the number of lattice points in a polytope quickly, even in the general d-dimensional case (and see [7] for detailed complexity bounds). This means that these techniques are a good candidate for consumer-side analysis of memory usage in mobile programs. However there is potentially a hidden bottleneck.

Barvinok's techniques depend on knowing the *vertices* of the polytope whereas the invariants arising from program analysis are constraints describing the *facets* (higher-dimensional faces) of the polytope. Thus in order to apply Barvinok's algorithm we must first perform Vertex Enumeration.

The problem of translating between the vertex and facet representations of polytopes have been intensively studied for at least 50 years, and many algorithms for converting one representation into the other are known (such algorithms are required in the theory of linear programming for example). See [27] for a survey of Vertex Enumeration algorithms (this paper dates from 1980, but it appears that no significantly better algorithms have appeared since then).

A difficulty here is that the number of vertices can be exponential in the number of constraints. It is easy to see that the d-cube $\{(x_1, \ldots, \mathbf{x}_d) \in \mathbb{R}^d : -1 \leq x + j \leq 1 \forall j\}$ can be described by $2d$ constraints but has 2^d vertices. For arbitrary polytopes, a corollary of McMullen's Upper Bound Theorem[29, 30] gives a sharp bound for the number of vertices. We require a couple of definitions before stating the result.

Definition 9. The *moment curve* in $\mathbb{R}^d$ is $\{(t^1, t^2, \ldots, t^d) : t \in \mathbb{R}\}$. The *cyclic polytope* $C(d, m)$ is the convex hull of m distinct points on the moment curve in $\mathbb{R}^d$.

The definition of $C(d, m)$ might appear to depend on the choice of points in the definition, but it can be shown that in fact the combinatorial type of the polytope is the same for all choices.

Theorem 10. *Let P be a polytope specified as the intersection of m halfspaces in $\mathbb{R}^d$. Then P can have as many as*

$$\binom{m - \lfloor (d+1)/2 \rfloor}{\lfloor d/2 \rfloor} + \binom{m - \lfloor (d+2)/2 \rfloor}{\lfloor (d-1)/2 \rfloor}$$

vertices, and this bound is attained by $C(d, m)$.

For $d = 2k$ even this number is equal to $\frac{m}{m-k}\binom{m-k}{k}$, and for $d = 2k+1$ odd, it is equal to $2\binom{m-k-1}{k}$. The following table gives some sample values of the upper bound, and demonstrates that the number of vertices can become extremely large as the dimension increases.

		m (number of constraints)					
		10	20	50	100	500	1000
	2	10	20	50	100	500	1000
	3	16	36	96	196	996	1996
	5	42	272	2162	9312	246512	993012
d	10	2	4004	1.36×10^6	6.10×10^7	2.45×10^{11}	8.09×10^{12}
	20	-	2	1.06×10^9	6.36×10^{12}	2.05×10^{20}	2.41×10^{23}
	40	-	-	5.01×10^7	4.42×10^{18}	1.21×10^{35}	2.30×10^{41}
	80	-	-	-	6.99×10^{15}	7.52×10^{57}	1.09×10^{71}

All known algorithms for Vertex Enumeration have worst-case running time $O(m^{\lfloor d/2 \rfloor})$ (m constraints involving d variables); see [3] for more information. The Upper Bound theorem shows that this time complexity is unavoidable, but it seems that in practice the upper bound is very seldom achieved.

On the other hand, the number of vertices can be as low as $(m-d)(d-1)+2$ [37, 8.38]. Due to this huge variation in the possible size of the output, VE algorithms are classified as *output-sensitive algorithms*, in which complexity is considered in terms of the input size plus the output size [12]. Thus for a polytope P in $\mathbb{R}^d$ with m facets[5] and n vertices, complexity is usually measured with respect to $size(P) = d(m+n)$. Even in this setting, the precise complexity is still open: it is unknown whether the VE problem can be solved in a time polynomial in $size(P)$, or whether it even lies in NP (but see [22] for recent progress in the case of unbounded polyhedra). See [3] for a survey of results in this area.

A further difficulty is that even checking the result of VE is hard. Suppose that we have have a halfspace representation of a polytope P and a vertex representation of a polytope Q (perhaps the output of a VE algorithm applied to P). The *Polytope Verification* problem (PV) is to check whether $P = Q$. It is easy to check whether $Q \subseteq P$ (just check whether every vertex of Q satisfies every constraint of P), but proving equality is difficult. In fact, it can be shown [3] that a polynomial-time algorithm for PV would yield a polynomial time algorithm for VE (with respect to the size measure mentioned earlier). This may be contrasted with a problem such as boolean satisfiability, where it may take a long time to find a satisfying solution for a boolean formula, but once an answer is known it can be checked very quickly. For Vertex Enumeration, checking a solution is provably as hard as obtaining one in the first place.

The remarks above suggest that Vertex Enumeration could be a major bottleneck in the application of Barvinok's methods to program analysis. This may not be as bad as it seems: in realistic programs it is probable that the numbers d, m and n would all be relatively small. For example, d is equal (more or less) to the maximum depth of loop nesting, and it is unlikely that this number would ever exceed 3 or 4. Nevertheless, it would be well worthwhile to investigate the possibility of *certifying* Vertex Enumeration algorithms: given a polytope described by halfspaces, calculate its vertices and produce some evidence which would allow another party to check quickly that the result is indeed correct. In this connection, it should be noted that the problem of *Polytope Verification* (*ie*, given a polytope P described by halfspaces and a polytope Q described by vertices, check that $P = Q$) is polynomially equivalent to Vertex Enumeration, and hence it is again unknown whether this problem is even in NP (ref?). It is easy to check whether $Q \subseteq P$ simply by checking that each vertex satisfies every constraint, but there is no known way to check the reverse inclusion other than performing Vertex Enumeration on P and seeing if the answer is in fact Q. The fact that the complexity of Polytope Verification is unknown suggests that a certification method for Vertex Enumeration might be of considerable interest, independent of its applications to program analysis.

6 Implementation

We have implemented a Java compiler which uses lattice point enumeration techniques to calculate resource bounds for simple imperative programs. This is a preliminary implementation, but the results it produces are quite promising; it

[5] A *facet* is the higher-dimensional equivalent of the notion of a face of a 3-dimensional polyhedron.

can successfully (and automatically) produce precise bounds for realistic matrix manipulation programs, for example. We will give a brief description of the structure of the compiler and then outline the methods we have used to perform our analysis.

6.1 The compiler

Our compiler is provisionally called `raj`, a name which is supposed to suggest *resource aware Java*.

The compiler itself is entirely implemented in OCaml, and handles the full Java language with the exception of generics and (at present) certain aspects of inner classes and `enum` classes. The compiler uses the `ocfgc` tool of Tse and Zdancewic to parse source code and convert it into a standard abstract syntax tree (AST), and then the following steps are performed.

- The AST is converted to an expression-based form which reflects the structure of the original Java program, but with all ambiguities resolved. Types are also inferred during this phase. We also replace all `for` loops with equivalent `while` loops.
- The expression-based form is converted to a linear SSA-like form.
- The linear version is broken into basic blocks
- The block-based form is converted into an abstract OCaml representation of Java bytecode.
- We use tools which we have developed previously to converted the abstract representation into executable Java classfiles.

The first phase, in which names and types are resolved, is by far the most complicated. Java source code is surprisingly ambiguous: for example, in a Java expression `x+1`, `x` may represent a local variable or parameter, a field of the current class, a field of some superclass or superinterface of the current class, a field in some enclosing class (or a superclass/superinterface thereof) if the current class is a nested class, or a field imported via a static import statement. Things become considerably more complex when qualified names of the form `x.y.z` are considered.

6.2 Inferring linear constraints

We perform our analysis on the expression-based form obtained in the first phase of the compilation process outlined above. This form is very similar to the source program, and preserves the explicit control-flow structures of Java.

Our first task is to infer systems of linear constraints on program variables. The expression-based form is converted into a control-flow graph and then between every pair of expressions we infer a polyhedron which bounds the values of the integral variables in the program. This is done using the abstract interpretation technique of Cousot and Halbwachs described in [16], whose details we will not describe here.

A number of polyhedral operations are required to perform this process. It is necessary to have some representation of polyhedra and the means to convert

between vertex and facet representations, and methods for combining polyhedra in various ways (intersection, join (polyhedral hull), widening, ...) are also needed. These can be difficult to program, but fortunately there are a number of high-quality libraries available. We have chosen to use the Parma Polyhedra Library (PPL) [5], which is a large (more than 100,000 lines) C++ library providing (amongst many others) all of the operations we require, including a widening operator (see [4]), which is more precise than the standard polyhedral widening operator proposed by Cousot and Halbwachs in [16] and which was very useful for our application. The PPL also provides an OCaml interface which was convenient for linking with our OCaml-based compiler.

Using the PPL it was a relatively straightforward task to implement the Cousot-Halbwachs technique and obtain linear bounds on program variables; see Appendix B for some examples.

6.3 Polytopes and loop bounds

Having obtained bounds on program variables, we now wish to use them to obtain bounds on the number of times a loop is executed. For simple examples, such as the one we looked at earlier, it is quite clear how to do this.

```
for (i=1; i<=9; i++)
    for (j=1; j<=i && j<=7; j++) B
```

The Cousot-Halbwachs technique yields the constraints $\{j \leq 7, j \geq 1, i-j \geq 0, i \leq 9\}$ and the structure of the program makes it clear that each lattice point in the corresponding polytope is visited once and once only during execution of the program.

Unfortunately, for more complex programs (and ones where the loop structure is less explicit) it is less easy to see what to do. To bound the number of times a particular location l is visited, it suffices to identify a set of variables $x_1, \ldots, \mathbf{x}_d$ such that the vector $\mathbf{x} = (x_1, \ldots, \mathbf{x}_d)$ never has the same value twice when the program reaches l; if we can determine a polyhedron P which contains all values of $\mathbf{x}$ then we can be sure that l is visited at most $|P \cap \mathbb{Z}^d|$ times. Typically, the variables $x_1, \ldots, \mathbf{x}_d$ will be variables controlling `for`-loops. However, we must ensure that the set $\mathbf{x} = (x_1, \ldots, \mathbf{x}_d)$ is not too large: if we include variables which are irrelevant to the progress and termination of the loop then we run the risk of obtaining too large a polytope, and thus too large a bound for the number of executions of the loop.

It is not clear how to do this for general `while` loops, and we have been unable to find any answers in the literature. We have also been unable to find any proofs of correctness for program analysis techniques based on lattice point enumeration. This may be due to the fact that these methods usually seem to be applied to source code, where the loop structure is explicit and it is clear which variables are involved in the behaviour of the loop. However, we are ultimately interested in applying these methods to unstructured bytecode, and we are already working on a slightly lower level than the source code.

We hope to do further research on finding provably correct methods for attacking these problems, but for the moment we are applying heuristics to decide which variables to look at. Our technique is fairly simple, and is as follows:

- At each program location l we have a list $V(l)$ consisting of all variables which are involved in the termination conditions of all enclosing loops.
- Having performed the Cousot-Halbwachs analysis, we obtain a set of constraints which give bounds on the values of every variable at the location l
- We regard a constraint C as *relevant* if (i) each of its variables either lies in $V(l)$ or is an argument of the current method, and (ii) not all variables of C are method arguments. This selects constraints which relate program variables to one another and to method arguments, but rejects ones which merely provide relations between arguments.
- We construct a polyhedron $P(l)$ given by all of the relevant constraints at l, and count the number of lattice points in $P(l)$ as described in the next section.

This technique is neither sound nor optimal, but it does give correct results in most of the examples we have looked at. It is possible to produce examples which cause the analysis to give incorrect results, but this is reasonably difficult to do. In particular, we believe that our method will always give reliable results for programs using standard `for` loops.

6.4 Lattice point enumeration

We have used the `barvinok` library[6] of Verdoolaege to perform lattice point enumeration. within our compiler. This is a library which implements the generating function methods of §4. The `barvinok` library is again implemented in C++, but this time there is no OCaml interface; moreover, the internal representations of constraints are different from those used in PPL. This necessitated the implementation of our own OCaml interface, together with a fair amount of code to translate from one representation to another, but this was fairly routine.

Using the library is fairly straightforward. At each location l of interest (for instance, l might involve memory allocation or invocation of a particular method), we construct the polyhedron $P(l)$ as described earlier and call a function from the library to determine the number of lattice points in $P(l)$. The structure of $P(l)$ may depend parametrically on one or more method arguments, but we have recorded which arguments are involved and can ask `barvinok` to give us a formula for the number of lattice points which is parametric with respect to these arguments.

We give several examples of the output of this phase in Appendix B.

7 Further work

There is a great deal of further work which could be done in this area. Some possibilities are given below.

[6] `http://freshmeat.net/projects/barvinok/`

Theorems. Prove some theorems about the correctness of the methods which we have described here. In particular, obtain provably correct methods for selecting program a set $V(l)$ of variables at each program location l such that valid linear constraints on those variables give a polyhedron P with the property that any lattice point in P corresponds to a most one visit to the location l.

Certifying algorithms. Develop certifying versions of algorithms for polyhedral computations (in particular, Vertex Enumeration) and lattice-point enumeration. This would be useful both from at least two points of view:

- Resource certification for mobile code. We would like to equip mobile programs with easily-verified certificates of resource usage, but without the overheads required by PCC techniques.
- Verification of the results of polyhedral computations. The libraries which we have used are very large and are implemented in C++; furthermore they rely on several other C++ libraries for things such as unlimited-precision arithmetic and calculations in number theory and linear algebra. This provides a lot of scope for errors to creep in. Certifying versions of these algorithms should make it possible to provide checkers for the results of analyses which are small (and hence hopefully easier to trust) and independent of the original analyses. This would increase confidence in the correctness of the analyses. See [31, 23] for more on this point of view.

Perform the analysis on compiled bytecode. The implementation which we have described here is performed on a language which is quite close to Java source, and where the loop structure is manifest. This was largely because our aim was to quickly develop a working prototype in order to examine the viability of these methods, but it would be desirable (especially in the context of mobile code certification) to be able to deal with unstructured bytecode. Being able to analyse bytecode would also mean that we were no longer tied to Java as a source language, and it is likely that methods which work for bytecode would also be applicable to languages such as C and even machine language without major changes.

Interprocedural analysis. Our current analysis is entirely intraprocedural. Extensions to make it interprocedural would enable us to deal with larger programs, and a greater variety of them. In particular we are unable to deal with recursive programs at the moment. The analysis of Braberman et al. [11] is somewhat more sophisticated in this respect (and this might provide a useful starting point for us), but they also comment on the desirability of being able to deal with recursion. One avenue which might be worth exploring here would be the possibility of integrating our methods into some existing tool, such as the COSTA tool which has been developed in Madrid for the cost analysis of Java bytecode [1, 2]. Another possibility in a rather different area would be the cost analysis tool developed in the Embounded Project for the Hume language for resource-bounded programs for embedded devices: see [20, 19], for example.

"Dependent" allocation. One of our motivations is to measure memory consumption of Java programs. A common assumption in research on this topic is that all objects from a given class are of the same size. However, this will not always be the case: for example, the Java `BigInteger` class represents integers with unlimited precision, and the size of an object will depend on the integer involved. Furthermore, the size of an object returned by a method may depend on the method arguments (consider the `BigInteger multiply` method). We are not aware of any previous research which is able to deal with this type of behaviour. However, there is some recent work on *weighted generating functions* for polytopes [35] in which instead of considering the usual generating function $\sum\{\mathbf{x}^{\mathbf{v}} : \mathbf{v} \in P \cap \mathbb{Z}^d\}$, one considers a function of the form $\sum\{f(\mathbf{v})\mathbf{x}^{\mathbf{v}} : \mathbf{v} \in P \cap \mathbb{Z}^d\}$ in which each lattice point is weighted according to some function f. This corresponds to the situation in which a nest of loops indexed by $i_1, \ldots, i_d$ allocates an amount of memory given by the function $f(i_1, \ldots, i_d)$. It seems plausible that this work (possibly in combination with techniques such as *sized types* [21]) would be useful for attacking the problem of dependent allocation of the type discussed above.

8 Acknowledgments

This work was funded by in part by the ReQueST grant (EP/C537068) from the UK Engineering and Physical Sciences Research Council.
This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905.
This report reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein.

Appendix A Multiple Parameters

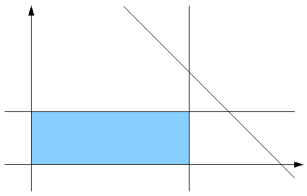
Consider this Java method:

```
public static void m2 (int p, int q)
  for (int i=0; i<=p; j++)
     for (int j=0; j<=9 && i+j<=q; j++)
        System.out.println ("Hello");
```

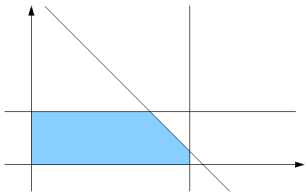
with corresponding constraints

$$0 \le i \le p, \qquad 0 \le j \le q, \qquad i + j \le 7.$$

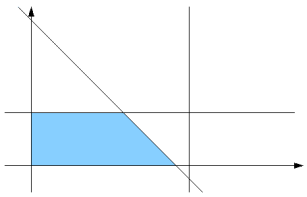
The iteration is controlled by the parameters p and q, and the shape (not just the size) of the *iteration space* depends on the relative values of these. If this case, there are five regions in the parameter space for which the corresponding polytope is nonempty.



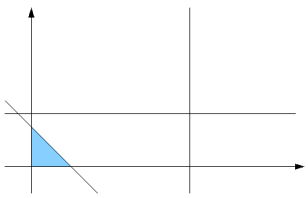
$0 \le p$
$p + 10 \le q$



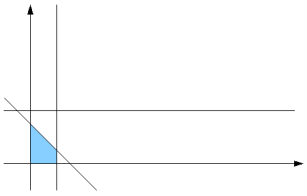
$p + 1 \le q$
$10 \le q \le p + 9$



$10 \le q \le p$



$0 \le q \le 9$
$q \le p$



$0 \le p$
$p + 1 \le q \le 9$

Analysis of multiple parameters.

- A different quasipolynomial in p and q is required to describe the number of lattice points in each of the five figures above.
- As the parameters p and q in this example vary, the set of points (i, j, p, q), where i and j satisfy the inequalities determined by p and q, forms a polyhedron in $\mathbb{R}^4$.
- The pictures of the iteration spaces above can be obtained as two-dimensional slices through this four-dimensional polyhedron.
- The enumerating function for each configuration can be *automatically* obtained from the generating function for the four-dimensional polyhedron.
- The `barvinok` library can supply these answers with relatively little effort on the part of the user.

Appendix B Examples

B.1 Simple example with nested loops

Our first example is the trapezoidal example considered earlier.

```
public class Trapezoid {

  public static void test () {
    for (int i=1; i<=9; i++)
      for (int j=1; j<=i && j<=7; j++)
        System.out.println ("Hello");
  }
  public static void main (String[] args) {
    test ();
  }

}
```

When invoked with the `-g` argument, `raj` outputs a rather crude representation of internal representation of the program, annotated with constraints inferred by the Cousot-Halbwachs technique. Note that the explicit `for` statements in the source program have been replaced by equivalent `while` statements in the internal representation. Note also that the constraints fall *between* statements; they are both a postcondtion for the preceding statement and a precondition for the following one.

```
==================== Method test ==================
Dimension = 2
5 iterations

--    {}
i <- 1

--    {i >= 1, i <= 10}
while (le[boolean] i 9) {

--    {i >= 1, i <= 9}
 j <- 1

--    {i >= 1, i-j >= -1, j >= 1, j <= 8, i <= 9}
 while (&&[boolean] le[boolean] j i le[boolean] j 7) {

--    {j <= 7, j >= 1, i-j >= 0, i <= 9}
  [invokevirtual getstatic  <[java.io].PrintStream.out java/langSystem>
     <void java/ioPrintStream.println([java.lang].String)> ("Hello")]

--    {j <= 7, j >= 1, i-j >= 0, i <= 9}
  j <- j+1

--    {j <= 8, j >= 2, i-j >= -1, i <= 9}
 }

--    {i >= 1, i-j >= -1, j >= 1, j <= 8, i <= 9}
 i <- i+1

--    {i >= 2, i-j >= 0, j >= 1, j <= 8, i <= 10}
}

--    {i = 10}
```

If we invoke the compiler with the `-p` option then it prints out the number of times the `println` statement is invoked (we have chosen to count method calls here in order to make the examples easy to understand, but we could easily modify the compiler to count the number of objects created, or the total number of statements executed for example).

```
==== method test ====

 Calls to java.io.PrintStream.println (java.lang.String):
  42

  {0 <= 1}

  ---
```

The reader can count the number of points in diagram 1 and check that 42 is the correct answer. Note the condition `{0 <= 1}` here; this is a representation of the polytope comprising all of $\mathbb{R}^2$ and says that the bound 42 is valid for all inputs. We will see examples later where bounds vary depending on the values of method arguments.

B.2 Gaussian Elimination

This example involves some code for solving simultaneous equations by Gaussian elimination. This code was downloaded from the internet, and has required minor amendments to make it amenable to our analysis (in particular we have provided the array size as a parameter, rather than looking it up from the array: it should be easy to modify the compiler to deal with this special case automatically). We have also added `println` statements at various points so that we can see how any times these points are visited.

```
public class GaussianElimination {
  private static final double EPSILON = 1e-10;

  // Gaussian elimination with partial pivoting
  public static double[] lsolve(double[][] A, double[] b, int N) {
    // int N  = b.length;

    for (int p = 0; p < N; p++) {
      System.out.println ("Loop 1");
      // find pivot row and swap
      int max = p;
      for (int i = p; i < N; i++) {  // i=p+1
        System.out.println ("Loop 1a");
        if (Math.abs(A[i][p]) > Math.abs(A[max][p])) {
          max = i;
        }
      }
      double[] temp = A[p]; A[p] = A[max]; A[max] = temp;
      double   t    = b[p]; b[p] = b[max]; b[max] = t;

      // singular or nearly singular
      if (Math.abs(A[p][p]) <= EPSILON) {
        throw new RuntimeException("Matrix is singular or nearly singular");
      }

      // pivot within A and b
      for (int i = p + 1; i < N; i++) {
        System.out.println ("Loop 2");
        double alpha = A[i][p] / A[p][p];
        b[i] -= alpha * b[p];
        for (int j = p; j < N; j++) {
          System.out.println ("Loop 2a");
          A[i][j] -= alpha * A[p][j];
        }
      }
    }

    // back substitution
    double[] x = new double[N];
    for (int i = N - 1; i >= 0; i--) {
      System.out.println ("Loop 3");
      double sum = 0.0;
      for (int j = i + 1; j < N; j++) {
      System.out.println ("Loop 3a");
        sum += A[i][j] * x[j];
      }
      x[i] = (b[i] - sum) / A[i][i];
    }
    return x;
  }
}
```

Here is the output of the lattice-point enumeration analysis: the method calls occur in the same order as in the source program.

```
==== method lsolve ====

 Calls to java.io.PrintStream.println (java.lang.String):
  N

  {1 <= N, 0 <= 1}

  ---
 Calls to java.io.PrintStream.println (java.lang.String):
  N^2

  {1 <= N, 0 <= 1}

  ---
 Calls to java.io.PrintStream.println (java.lang.String):
  -N/2 + N^2/2

  {2 <= N, 0 <= 1}

  ---
 Calls to java.io.PrintStream.println (java.lang.String):
  -N/3 + 0 + N^3/3

  {2 <= N, 0 <= 1}

  ---
 Calls to java.io.PrintStream.println (java.lang.String):
  N

  {1 <= N, 0 <= 1}

  ---
 Calls to java.io.PrintStream.println (java.lang.String):
  -N/2 + N^2/2

  {2 <= N, 0 <= 1}

  ---
 Calls to java.lang.Math.abs (double):
  N^2

  {1 <= N, 0 <= 1}

  ---
 Calls to java.lang.Math.abs (double):
  N^2

  {1 <= N, 0 <= 1}

  ---
 Calls to java.lang.Math.abs (double):
  N

  {1 <= N, 0 <= 1}

  ---
```

B.3 Multi-parameter example

Here is the example discussed in Appendix A. This demonstrates the ability of our analysis (exploiting the capabilities of the `barvinok` library) to automatically discover how resource bounds vary with input parameters.

```
public class Two_args {

  public static void f (int p, int q) {

    for (int i=0; i <= p; i++)
      for (int j=0; j <= 9 && i+j <= q; j++)
        System.out.println ("Hello");

  }

  public static void main (String[] args) {
    int k = Integer.parseInt (args[0]);
    int l = Integer.parseInt (args[1]);
    f(k,l);
  }
}
```

The analysis outputs five cases, in each case giving us a parametric bound on the number of method calls together with a set of constraints telling us for which values of the input parameters that bound applies.

```
==== method f ====

 Calls to java.io.PrintStream.println (java.lang.String):
  5 domains in R^2

  -35 + 10q

  {q <= p, 10 <= q, 0 <= 1}

  ---
  1 + (3/2)q + q^2/2

  {q <= p, 0 <= q, q <= 9}

  ---
  (1 + q) + (1/2 + q)p + -p^2/2

  {q <= 9, 0 <= p, p+1 <= q}

  ---
  10 + 10p

  {0 <= p, p+10 <= q, 0 <= 1}

  ---
  (-35 + (19/2)q + -q^2/2) + (1/2 + q)p + -p^2/2

  {p+1 <= q, q <= p+9, 10 <= q}

  ---
```

B.4 A deeply nested example

The preceding examples have been fairly small, and in each case the entire analysis (and compilation to bytecode) took less than two seconds. A large part of this time was occupied in indexing the `rt.jar` file containing the standard Java API classes (17335 of them), which are required for the resolution of external methods and fields.

Here is a rather complex example which is somewhat more challenging for the analysis.

```
public class Deep2 {

  public static void f (int p, int q, int r, int s, int t, int u) {
    for (int j1=0; j1<=5*p+90000; j1++)
      for (int j2=1; 5*j2-2378*q<=40000; j2++)
        for (int j3=7; 3234*r + j3<=20000; j3++)
          for (int j4=10; j4<=10000-234*s; j4++)
            for (int j5=50; j5<=8000 && j5 <= j4+u+4*t; j5++)
              for (int j6=3000; j6<=4000 && j6 <j5+u; j6++)
                for (int j7=50; j7<=8000; j7++)
                  for (int j8=3000; j8<=4000&&j8<j2 &&j8<j5 ; j8++)
                    System.out.println ("Hello");
  }

}
```

This takes over a minute to analyse, and produces over 838 kilobytes of textual output. The output begins as follows:

```
==== method f ====

 Calls to java.io.PrintStream.println (java.lang.String):
  101 domains in R^6

  ((((((343130944627677489317943788880 0 + 13737876631608179097487440000{(4/5) + (2/5)q})
+ ((98456152026103767052064287 38/5) + 7883745207679366381235888{(4/5) + (2/5)q})u
+ ((13425228674983468985782 14/5) + 1075007300715335627646{(4/5) + (2/5)q})u^2)
+ ((32220548819960325565877136 00 + 1290008760858402753168000{(4/5) + (2/5)q})
+ ((35800609799955917295419 04/5) + 2866686135240895007046{(4/5) + (2/5)q})u)t)
+ ((-18849021059676790456038124 5600 + -7546551251021656106032 8000{(4/5) + (2/5)q})
+ ((-20943356732974211617820 1384/5) + -16770113891159235791184{(4/5) + (2/5)q})u)s)
+ ((((-55500924023502500772943393 6800 + -2222081275713756687069 84000{(4/5) + (2/5)q})
+ ((-15925137323818124596897 0218/5) + -12751841553283520494606 8{(4/5) + (2/5)q})u
+ ((-21715109300238340852265 4/5) + -17388084477910350204{(4/5) + (2/5)q})u^2)
+ ((-52116262320572018045436 9600 + -2086570137349242024480 00{(4/5) + (2/5)q})
+ ((-57906958133968908939374 4/5) + -463682252744276005 44{(4/5) + (2/5)q})u)t)
+ ((30488013457534630556580621600 + 12206435303493065843208000{(4/5) + (2/5)q})
+ ((33875570508371811729534024/5) + 271254117855401463182 4{(4/5) + (2/5)q})u)s)r)
+ (((((65281148842083624445970134 1600 + 13068004149971009693053 6000{(4/5) + (2/5)q})
+ ((18731422553011135445716898 66/5) + 7499325976929213424197 2{(4/5) + (2/5)q})u
+ ((25541687950108361591359 8/5) + 10225888791776573916{(4/5) + (2/5)q})u^2)
+ ((61300051080260067819263 5200 + 1227106655013188869920 00{(4/5) + (2/5)q})
+ ((68111167866955630910292 8/5) + 272690367780708637 76{(4/5) + (2/5)q})u)t)
+ ((-35860529881952139674269159200 + -7178573931827154889032000{(4/5) + (2/5)q})
+ ((-39845033202169044082521288/5) + -15952386515171455308 96{(4/5) + (2/5)q})u)s)
+ ((((-10559129506616907145056887 7600 + -2113730390167362476109 6000{(4/5) + (2/5)q})
+ ((-30297799608101436446658 2226/5) + -12130049119430367217092{(4/5) + (2/5)q})u
+ ((-41313303406347124830678/5) + -1654022424357579276{(4/5) + (2/5)q})u^2)
+ ((-99151928175233099593627200 + -1984826909229095131 2000{(4/5) + (2/5)q})
+ ((-11016880908359233288180 8/5) + -4410726464953544736{(4/5) + (2/5)q})u)t)
+ ((58003877982511363262271912 00 + 1161123741899020651752000{(4/5) + (2/5)q})
```

```
+ ((64448753313901514735857 68/5) + 258027498199782367056{(4/5) + (2/5)q})u)s)r)q
+ ((((3104957786033111903069535 3600 + (8909199260591905547946 2736/5)u
+ (12148355884630569812208/5)u^2) + (2915605412311336754929 9200
+ (3239561569234818616588 8/5)u)t)
+ (-17056291662021320016340 03200 + (-18951435180023688907044 48/5)u)s)
+ (((-50222234070376532432 36409600 + (-14410498353883276253905296/5)u
+ (-19649786401368041798 88/5)u^2) + (-47159487363283300317 31200
+ (-52399430403648111463 68/5)u)t)
+ (27588300107520730685 6275200 + (3065366678613414520 62528/5)u)s)r)q^2)
+ ((((((19062618450221524722 9444000 + 763206888346139437 20000{(4/5) + (2/5)q})
+ (1093945089789044200 08738 + 43798097841576017940{(4/5) + (2/5)q})u
+ (14916755008259318214 + 5972196424013820{(4/5) + (2/5)q})u^2)
+ ((17900106009911181856 8000 + 716663570881658400 00{(4/5) + (2/5)q})
+ (3977801355358181904 + 1592585713070352 0{(4/5) + (2/5)q})u)t)
+ ((-10471562015798041386228000 + -41924818896577016400 00{(4/5) + (2/5)q})
+ (-2327013781288453641384 + -931662642146155920{(4/5) + (2/5)q})u)s)
+ ((((-30833504085233775609684000 + -12344758812200734920000{(4/5) + (2/5)q})
+ (-17694400422015449350218 + -7084277704294130340{(4/5) + (2/5)q})u
+ (-2412763113769662654 + -965993959951020{(4/5) + (2/5)q})u^2)
+ ((-28953157365235951848000 + -115919275194122400 00{(4/5) + (2/5)q})
+ (-6434034970052433744 + -2575983893202720{(4/5) + (2/5)q})u)t)
+ ((16937597058663031831080 00 + 67812775988561604 0000{(4/5) + (2/5)q})
+ (376391045748067374024 + 150695057752359120{(4/5) + (2/5)q})u)s)r)
+ (((((36266901946691494786708000 + 725992163974345268 0000{(4/5) + (2/5)q})
+ (20812460476007083749866 + 4166245917783809860{(4/5) + (2/5)q})u
+ (2837933795192093598 + 5680986206695 80{(4/5) + (2/5)q})u^2)
+ ((3405520554230512317 6000 + 681718344803496 0000{(4/5) + (2/5)q})
+ (7567823453845582928 + 1514929655118880{(4/5) + (2/5)q})u)t)
+ ((-19922295242248497057960 00 + -39880523171004516 0000{(4/5) + (2/5)q})
+ (-442717672049966601288 + -88623384824454480{(4/5) + (2/5)q})u)s)
+ ((((-58661178801440579243880 00 + -11742816136306054 80000{(4/5) + (2/5)q})
+ (-3366384774402666242226 + -673884130144685460{(4/5) + (2/5)q})u
+ (-459031604163810678 + -91889113696380{(4/5) + (2/5)q})u^2)
+ ((-5508379249965728136000 + -1102669364356560000{(4/5) + (2/5)q})
+ (-122408427777016180 8 + -245037636523680{(4/5) + (2/5)q})u)t)
+ ((32224018612299509595600 0 + 6450615781458 8760000{(4/5) + (2/5)q})
+ (71608930249554465768 + 14334701736635280{(4/5) + (2/5)q})u)s)r)q
+ ((((17249573816030443567680 00 + 9899000300654332227 36u + 134980232271092208u^2)
+ (1619762787253106496000 + 359947286056245888u)t)
+ (-9475612305430673001600 0 + -21056916234290384448u)s)
+ (((-27900931139863186204800 0 + -16011486932237726529 6u + -21832853414259888u^2)
+ (-26199424097111865600 0 + -58220942438026368u)t)
+ (153266630968104413760 00 + 3405925132624542528u)s)r)q^2)p

  {2t+u <= 1495, 0 <= p+18000, 234s <= 4t+u+1999, 0 <= u+3999,
   2378q+5u+24995 <= 0, 0 <= q+10, r <= 6, q+9 <= 0}

  ---
```

This is followed by another 100 similar results. The symbols `{}` appearing round some terms denote fractional parts, and demonstrate the quasipolynomial nature of the expression, which arises from the fact that we have a polytope with non-integral vertices.

Fortunately this example is artificial and it is unlikely that such code would arise naturally, but the fact that this result was obtained entirely automatically serves to demonstrate the power of the methods used in the analysis.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *In 16th European Symposium on Programming, ESOP07, Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pages 157–172. Springer–Verlag, 2007.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Cost relation systems: A language-independent target language for cost analysis. In *Spanish Conference on Programming and Computer Languages (PROLE'08)*, 2008. To appear.

[3] David Avis and David Bremner. How good are convex hull algorithms? *Computational Geometry: Theory and Applications*, 7:265–301, 1997.

[4] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, 2005.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[6] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Applications of polyhedral computations to the analysis and verification of hardware and software systems. Technical report, Department of Mathematics, University of Parma, Italy, 2007. Technical Report. Quaderno 458 (2007). Updated version to appear in TCS.

[7] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of *Math. Sci. Res. Inst. Publ.*, pages 91–147. Cambridge Univ. Press, Cambridge, 1999.

[8] Matthias Beck and Sinai Robins. *Computing the continuous discretely.* Undergraduate Texts in Mathematics. Springer–Verlag, New York, 2007.

[9] Matthias Beck, Steven Sam, and Kevin Woods. Maximal periods of (Ehrhart) quasi-polynomials. *J. Combin. Theory Ser. A*, 115:517–525, 2008.

[10] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Symbolic prediction of dynamic memory requirements. In *ISMM 2008*.

[11] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, Jun 2006.

[12] David Bremner. Incremental convex hull algorithms are not output sensitive. In *ISAAC '96: Proceedings of the 7th International Symposium on Algorithms and Computation*, volume 1178 of *Lecture Notes in Computer Science*, pages 26–35, London, UK, 1996. Springer–Verlag.

[13] Michel Brion. Points entiers dans les polyèdres convexes. *Ann. Sci. École Norm. Sup. (4)*, 21(4):653–663, 1988.

[14] Philippe Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 278–285, New York, NY, USA, 1996. ACM.

[15] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, 1998.

[16] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[17] Eugène Ehrhart. Sur un problème de géométrie diophantienne linéaire. I. Polyèdres et réseaux. *J. Reine Angew. Math.*, 226:1–29, 1967.

[18] Eugène Ehrhart. Sur un problème de géométrie diophantienne linéaire. II. Systèmes diophantiens linéaires. *J. Reine Angew. Math.*, 227:25–49, 1967.

[19] Kevin Hammond, Greg Michaelson, and others. The EmBounded project. See `http://www.embounded.org/`.

[20] Kevin Hammond, Greg Michaelson, and others. The Hume language. See `http://www-fp.cs.st-andrews.ac.uk/hume/index.shtml`.

[21] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of 23rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '96)*, pages 410–423. ACM Press, 1996.

[22] Leonid Khachiyan, Endre Boros, Konrad Borys, Khaled M. Elbassioni, and Vladimir Gurvich. Generating all vertices of a polyhedron is hard. *Discrete & Computational Geometry*, 39(1-3):174–190, 2008.

[23] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 158–167, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[24] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR '93, LNCS 715*, pages 398–416. Springer-Verlag, 1993.

[25] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25:25–6, 1997.

[26] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of symbolic computation*, 38:1273–1302, 2004.

[27] T. H. Matheiss and David S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5(2):167–185, 1980.

[28] Tyrrell B. McAllister. Coefficient functions of the Ehrhart quasi-polynomials of rational polygons. In Matthias Dehmer, Michael Drmota, and Frank Emmert-Streib, editors, *ITSL*, pages 114–118. CSREA Press, 2008.

[29] P. McMullen. The maximum numbers of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.

[30] P. McMullen and G.C. Shephard. *Convex polytopes and the upper bound conjecture*. Cambridge University Press, 1971.

[31] Kurt Mehlhorn, Arno Eigenwillig, Kanela Kanegossi, Dieter Kratsch, Ross Mcconnel, Uli Meyer, and Jeremy Spinrad. Certifying algorithms (a paper under construction), 2005. Available at `http://www.mpi-inf.mpg.de/~mehlhorn/ftp/CertifyingAlgorithms.pdf`.

[32] Benoît Meister. Approximations of polytope enumerators using linear expansions. Technical report, Universite Louis Pasteur, May 2007. Revised edition of a paper submitted in 2005.

[33] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part II, multidimensional time. *SIGPLAN Not.*, 43(6):90–100, 2008.

[34] William Pugh. Counting solutions to Presburger formulas: how and why. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 121–134, New York, NY, USA, 1994. ACM.

[35] Sven Verdoolaege and Maurice Bruynooghe. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In *The 2008 International Conference on Information Theory and Statistical Learning*, pages 60–66, 2008.

[36] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations. In M.J. Irwin, W. Zhao, L. Lavagno, and S. Mahlke, editors, *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems (CASES)*, pages 248–258, Washington DC, USA, September 2004. ACM.

[37] Günter M. Ziegler. *Lectures on polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer–Verlag, New York, 1995.

Deciding Extensions of the Theories of Vectors and Bags

Patrick Maier

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh, Scotland
Patrick.Maier@ed.ac.uk

Abstract. Vectors and bags are basic collection data structures, which are used frequently in programs and specifications. Reasoning about these data structures is supported by established algorithms for deciding ground satisfiability in the theories of arrays (for vectors) and multisets (for bags), respectively. Yet, these decision procedures are only able to reason about vectors and bags in isolation, not about their combination.
This paper presents a decision procedure for the combination of the theories of vectors and bags, even when extended with a function bagof bridging between vectors and bags. The function bagof converts vectors into the bags of their elements, thus admitting vector/bag comparisons. Moreover, for certain syntactically restricted classes of ground formulae decidability is retained if the theory of vectors is extended further with a map function which applies uninterpreted functions to all elements of a vector.

1 Introduction

Vectors and bags are basic collection data structures, which are used frequently in programs and specifications. Reasoning about these data structures is supported by decision procedures for deciding the satisfiability of quantifier-free formulae in the theories of arrays (for vectors) and multisets (for bags), respectively. However, known decision procedures are essentially only able to reason about vectors and bags in isolation, whereas practical software verification problems often require non-trivial combinations.

Let us illustrate this problem with an example. Figure 1 shows a Java method `sendBulk` taking a message text `msg`, a group of recipients `group` (represented as an array of phone numbers) and a *resource manager* `mgr` holding (symbolic representations of) the resources required to send text messages to the recipients. As the cost of sending text messages may vary depending on the recipient, the state of a resource manager cannot be simply the number of messages that may be sent; instead it should be a multiset of resources, representing exactly how many messages may be sent to whom. In order to enforce the resource limit, at least at run-time, actual use of resources must be preceded by a call to the resource manager's `use` method, which checks whether the required resource is present and if so, deduces it, otherwise aborts the program. This is what's happening in the body of method `sendBulk`, which iterates over `group`, sending `msg` to each member by calling `SMS.send`, but only after checking for and using up the associated resource by calling `mgr.use`. This approach to run-time monitoring of resources via explicit resource managers has been described in [1], for example.

```
void sendBulk(String msg, PhoneNum[] group, ResourceMgr mgr) {
    for (int i=0; i < group.length; i++) {
        mgr.use(MessageResource(group[i]));
        SMS.send(msg, group[i]);
    }
}
```

$$
\begin{aligned}
\mathit{PreCond} &\equiv \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group})) \subseteq \texttt{mgr} \\
\mathit{PostCond} &\equiv \texttt{\textbackslash old(mgr)} = \texttt{mgr} \uplus \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group})) \\
\mathit{LoopInv} &\equiv 0 \leq \texttt{i} \leq \texttt{group.length} \wedge \\
&\quad \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group[i:group.length]})) \subseteq \texttt{mgr} \wedge \\
&\quad \texttt{\textbackslash old(mgr)} = \texttt{mgr} \uplus \mathrm{bagof}(\mathrm{map}_{\texttt{MessageResource}}(\texttt{group[0:i]})) \\
\mathit{VC} &\equiv \mathit{LoopInv} \wedge \neg \mathit{LoopInv}[\texttt{i} + 1/\texttt{i}, \texttt{mgr}'/\texttt{mgr}] \wedge \texttt{i} < \texttt{group.length} \wedge \\
&\quad \mathrm{count}(\texttt{mgr}, \texttt{MessageResource(group[i])}) > 0 \wedge \\
&\quad \texttt{mgr} = \texttt{mgr}' \uplus [\![\texttt{MessageResource(group[i])}]\!]^{(1)}
\end{aligned}
$$

Fig. 1. Java bulk messaging example: code and specification of send loop.

Run-time monitoring provides dynamic guarantees of *resource safety*, as abuse of resources will be trapped. However, aborting a program midway is not always a desirable solution; it would be better if we could guarantee statically that a program will never even *attempt* to abuse resources. This is done in [2], which presents a type system for proving static resource safety in a programming language with explicit resource managers. When proving resource safety of a method like `sendBulk`, whether it is done via a type system as in [2] or in the more traditional way by generating verification conditions, the hard part is reasoning about constraints between the program variables. Ideally, we'd like to have fully automated theorem provers for this task.

Let us take a look at the constraints required to express invariants and pre- and postconditions for `sendBulk`, see the bottom half of Figure 1. Informally, the precondition states that `mgr` is a super-multiset of the vector `group`, when the latter is viewed as a multiset of resources. To express this view, we first need to convert `group` into a vector of resources (by applying the map function) and then into a multiset of resources (by applying the bagof function). The postcondition states that the old `mgr` splits into two multisets: the new `mgr` and the multiset of resources corresponding to the vector `group`. The loop invariant essentially combines pre- and postcondition, but for different slices of the vector `group`. The first conjunct bounds the loop variable `i`, the second is the precondition for the remainder of the loop, i. e., for the subvector from index `i` to the end, and the third is the effect of the loop so far, i. e., the postcondition for the subvector from index 0 up to (but excluding) `i`. The (negated) verification condition conjoins the loop invariant before, the negated loop invariant after the execution of the loop (arising by substituting the variables `i` and `mgr`), the loop condition, and the precondition (`mgr` has some resources corresponding to number `group[i]`) and effect (`mgr`$'$ holds one unit of resource less than `mgr`) of the loop body. Hence, to verify the loop invariant of an example even this simple we must prove unsatisfiability of

constraints about bags, vectors, subvectors, the map function for transforming vectors pointwise, and the bagof function for transforming vectors into multisets.

Decision procedures for vectors (or arrays) exist for quite some time; early work goes back to the late 1970s [6, 10]. Recently, [4] and [3] found expressive yet decidable extensions of the theory arrays by injectivity predicates and by restricted quantification over array indices, respectively. Decision procedures for bags (or multisets) have been published recently in [12] and [7, 8], where the latter supports a cardinality operator. However, decision procedures combining vectors and bags and linking them via the bagof function (or something similar) do not exist.

The main contribution of this paper is a decision procedure for ground satisfiability in the combination of the theories of vectors and bags extended with the function bagof. For certain syntactically restricted classes of ground formulae decidability is retained if the theory of vectors is extended further with a map_f function for transforming vectors pointwise by applying the uninterpreted function f. The decision procedure reduces formulae containing $\mathrm{bagof}(\cdot)$ to formulae without by instantiating universally quantified variables in the axiomatisation of the bagof function, eventually reducing the problem to the theories of vectors and bags. It relies on a decision procedure for the Array Property Fragment described in [3] and on a decision procedure for multisets with cardinality described in [7, 8].

Plan. Section 2 introduces some basic notation. Section 3 presents the theories of bags, vectors, map and bagof functions. Section 4 utilises known results to construct a decision procedure for the combination of the theories of bags and vectors (including map). Section 5 presents our main result: an extension of the decision procedure (and its proof of correctness) to cope with bagof.

2 Preliminaries

We work in the framework of *many-sorted first-order logic with equality*, assuming familiarity with the basic syntactic and semantic concepts. Below we fix some notation.

Throughout the paper, we fix three countably infinite and pairwise disjoint universes: a set $\mathcal{S}$ of *sorts*, a set $\mathcal{F}$ of *function symbols* and a set $\mathcal{X}$ of *variable symbols*. By S^+ we denote the set of non-empty words over a set S.

Signatures. A *decorated variable* x_s is a pair consisting of a variable $x \in \mathcal{X}$ and a sort $s \in \mathcal{S}$. A *decorated function symbol* f_w is a pair consisting of a function symbol $f \in \mathcal{F}$ and an *arity* $w \in \mathcal{S}^+$. A decorated function symbol c_s of arity $s \in \mathcal{S}$ is called a *decorated constant*. For the sake of readability, we may write decorated constants and function symbols in the form $c : s$ and $f : s_1 \times \ldots \times s_n \to s_0$ instead of c_s and $f_{s_0 s_1 \ldots s_n}$, respectively. We may drop decorations entirely if they are clear from the context.

A *(many-sorted) signature* Σ is a pair $\Sigma = \langle S, F \rangle$ where $S \subseteq \mathcal{S}$ is a non-empty finite set of sorts and $F \subseteq \mathcal{F} \times S^+$ is a set of decorated function symbols. We may write Σ^{S} and Σ^{F} to refer to S and F, respectively. If Σ_1 and Σ_2 are signatures then the *union* $\Sigma_1 \cup \Sigma_2 = \langle \Sigma_1^{\mathrm{S}} \cup \Sigma_2^{\mathrm{S}}, \Sigma_1^{\mathrm{F}} \cup \Sigma_2^{\mathrm{F}} \rangle$ and *intersection* $\Sigma_1 \cap \Sigma_2 = \langle \Sigma_1^{\mathrm{S}} \cap \Sigma_2^{\mathrm{S}}, \Sigma_1^{\mathrm{F}} \cap \Sigma_2^{\mathrm{F}} \rangle$ are signatures, too. Two signatures Σ_1 and Σ_2 are *disjoint* if $\Sigma_1^{\mathrm{F}} \cap \Sigma_2^{\mathrm{F}} = \emptyset$, i. e., disjoint signatures do not share decorated function symbols but may share sorts.

Union and intersection induce a lattice structure on signatures. We denote the induced partial order by $\supseteq$, where $\Sigma_2 \supseteq \Sigma_1$ (in words: Σ_2 *extends* Σ_1) if $\Sigma_2^{\mathrm{S}} \supseteq \Sigma_1^{\mathrm{S}}$ and $\Sigma_2^{\mathrm{F}} \supseteq \Sigma_1^{\mathrm{F}}$. The *constant expansion* of Σ, denoted by $\hat{\Sigma}$, is the greatest signature extending Σ such that $\hat{\Sigma}^{\mathrm{S}} = \Sigma^{\mathrm{S}}$ and all function symbols in $\hat{\Sigma}^{\mathrm{F}} \setminus \Sigma^{\mathrm{F}}$ are constants, i. e., $\hat{\Sigma}$ provides infinitely many constants per sort.

Terms and formulae. Let Σ be a signature. *Σ-terms* are well-sorted terms constructed from decorated function symbols in Σ^{F} and decorated variables in $\mathcal{X} \times \Sigma^{\mathrm{S}}$. A *ground* Σ-term is a variable-free Σ-term. If Σ is clear from the context, we may drop the prefix and write "term" instead of "Σ-term". We may refer to terms of sort $s \in \Sigma^{\mathrm{S}}$ as *s-terms*.

A Σ-atom is an equality[1] $t = t'$, where t and t' are Σ-terms of the same sort. A Σ-literal is a Σ-atom $t = t'$ or its negation $\neg(t = t')$, often written as $t \neq t'$. If we want to stress that the sort of left- and right-hand sides of a Σ-atom (resp.-literal) is s, we may refer to the atom (resp. literal) as s-atom (resp. s-literal). Σ-formulae are formed from Σ-atoms by the usual connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) and quantifiers ($\forall$, $\exists$) of first-order logic, inducing the usual notion of bound and free variables. A *Σ-sentence* is a Σ-formula without free variables, and a *Σ-theory* is a set of Σ-sentences. Note that a Σ-theory $\mathcal{T}$ is also a Σ'-theory, for all Σ' extending Σ. A *ground* Σ-formula is a quantifier-free Σ-sentence.

Algebras and satisfiability. Let $\Sigma = \langle S, F \rangle$ be a signature. A *Σ-algebra* $\mathcal{A}$ is a pair $\langle S^{\mathcal{A}}, F^{\mathcal{A}} \rangle$ where $S^{\mathcal{A}}$ is a S-indexed family of carrier sets and $F^{\mathcal{A}}$ is a F-indexed family of functions on the carrier sets. More formally, $S^{\mathcal{A}} = \{s^{\mathcal{A}} | s \in \Sigma^{\mathrm{S}}\}$ is a family of non-empty and pairwise disjoint sets $s^{\mathcal{A}}$, and $F^{\mathcal{A}} = \{f^{\mathcal{A}}_{s_0 s_1 \ldots s_n} | f_{s_0 s_1 \ldots s_n} \in F\}$ is a family of functions $f^{\mathcal{A}}_{s_0 s_1 \ldots s_n}$ from $s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}}$ to $s_0^{\mathcal{A}}$. We extend the interpretation of function symbols in a Σ-algebra $\mathcal{A}$ homomorphically to ground Σ-terms t in the usual way, denoting the resulting element of the algebra by $t^{\mathcal{A}}$. Note that for all Σ' extending Σ, a Σ'-algebra $\mathcal{A}$ can also be viewed as a Σ-algebra.

The truth of a Σ-sentence ϕ in a Σ-algebra $\mathcal{A}$, denoted by $\mathcal{A} \models \phi$, is defined in the usual way. $\mathcal{A}$ is a *model* of a Σ-theory $\mathcal{T}$, also denoted by $\mathcal{A} \models \mathcal{T}$, if $\mathcal{A} \models \phi$ for all $\phi \in \mathcal{T}$. Given a Σ-algebra $\mathcal{A}$, the theory $\mathcal{T}(\mathcal{A})$ is the greatest Σ-theory which has $\mathcal{A}$ as a model. Given a class Δ of Σ-algebras, $\mathcal{T}(\Delta) = \bigcap_{\mathcal{A} \in \Delta} \mathcal{T}(\mathcal{A})$ is the greatest Σ-theory which has all algebras $\mathcal{A} \in \Delta$ as models.

Let $\mathcal{T}$ be a Σ-theory. A Σ-algebra $\mathcal{A}$ is a $\mathcal{T}$-model if $\mathcal{A} \models \mathcal{T}$. A $\hat{\Sigma}$-sentence ϕ is *$\mathcal{T}$-satisfiable* if there is a $\mathcal{T}$-model $\mathcal{A}$ which is a model of ϕ; note that $\mathcal{A}$ must be a $\hat{\Sigma}$-algebra. Two $\hat{\Sigma}$-sentences ϕ and ψ are *$\mathcal{T}$-equisatisfiable* if both are $\mathcal{T}$-satisfiable or neither is.

Given a subset $S' \subseteq \Sigma^{\mathrm{S}}$ of sorts, a Σ-theory $\mathcal{T}$ is *stably infinite w. r. t. S'* if every $\mathcal{T}$-satisfiable ground $\hat{\Sigma}$-formula ϕ has a $\mathcal{T}$-model $\mathcal{A}$ such that $s^{\mathcal{A}}$ is infinite for all $s \in S'$. $\mathcal{T}$ is *stably infinite* if it is stably infinite w. r. t. the set of all sorts Σ^{S}.

3 Theories

We introduce the signatures and theories used throughout this paper, see also Figure 2.

[1] We consider equality the only predicate symbol of the logic. Other predicates can be encoded as functions with a non-trivial codomain.

Σ_{E}-theory $\mathcal{T}_{\mathrm{E}}$ of elements

Σ_{E} arbitrary signature disjoint from all signatures below,
$\mathcal{T}_{\mathrm{E}}$ arbitrary stably infinite theory with decidable ground $\mathcal{T}_{\mathrm{E}}$-satisfiability problem.

Σ_{INT}-theory $\mathcal{T}_{\mathrm{INT}}$ of Presburger arithmetic

$$\Sigma^{\mathrm{S}}_{\mathrm{INT}} = \{\mathrm{INT}\}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{INT}} = \big\{0, 1 : \mathrm{INT},\ +, -, \min, \max : \mathrm{INT} \times \mathrm{INT} \to \mathrm{INT}\big\}$$
$$\mathcal{T}_{\mathrm{INT}} = \mathcal{T}(\mathcal{A}_{\mathrm{INT}}) \text{ where } \mathcal{A}_{\mathrm{INT}} \text{ is the standard } \Sigma_{\mathrm{INT}}\text{-algebra.}$$

Σ_{BAG}-theory $\mathcal{T}_{\mathrm{BAG}}$ of multisets with cardinality

$$\Sigma^{\mathrm{S}}_{\mathrm{BAG}} = \Sigma^{\mathrm{S}}_{\mathrm{INT}} \cup \Sigma^{\mathrm{S}}_{\mathrm{E}} \cup \{\mathrm{BAG}_s \mid s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{BAG}} = \Sigma^{\mathrm{F}}_{\mathrm{INT}} \cup \big\{|\cdot| : \mathrm{BAG}_s \to \mathrm{INT},\ \mathrm{count} : \mathrm{BAG}_s \times s \to \mathrm{INT},\ [\![]\!] : \mathrm{BAG}_s,\ [\![\cdot]\!]^{(\cdot)} : s \times \mathrm{INT} \to \mathrm{BAG}_s,\ \cap, \cup, \uplus : \mathrm{BAG}_s \times \mathrm{BAG}_s \to \mathrm{BAG}_s \;\big|\; s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\big\}$$
$$\mathcal{T}_{\mathrm{BAG}} = \mathcal{T}(\Delta_{\mathrm{BAG}}) \text{ where } \Delta_{\mathrm{BAG}} \text{ is the class of standard } \Sigma_{\mathrm{BAG}}\text{-algebras.}$$

Σ_{VEC}-theory $\mathcal{T}_{\mathrm{VEC}}$ of vectors

$$\Sigma^{\mathrm{S}}_{\mathrm{VEC}} = \Sigma^{\mathrm{S}}_{\mathrm{INT}} \cup \Sigma^{\mathrm{S}}_{\mathrm{E}} \cup \{\mathrm{VEC}_s \mid s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{VEC}} = \Sigma^{\mathrm{F}}_{\mathrm{INT}} \cup \big\{\mathrm{fst}, \mathrm{end} : \mathrm{VEC}_s \to \mathrm{INT},\ \cdot[\cdot] : \mathrm{VEC}_s \times \mathrm{INT} \to s,\ \mathrm{const} : s \times \mathrm{INT} \times \mathrm{INT} \to \mathrm{VEC}_s,\ \cdot[\cdot{:}\cdot] : \mathrm{VEC}_s \times \mathrm{INT} \times \mathrm{INT} \to \mathrm{VEC}_s,\ \cdot\{\cdot \leftarrow \cdot\} : \mathrm{VEC}_s \times \mathrm{INT} \times s \to \mathrm{VEC}_s \;\big|\; s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\big\}$$
$$\begin{aligned}\mathcal{T}_{\mathrm{VEC}} = \big\{&\forall u, v : \mathrm{fst}(u) = \mathrm{fst}(v) \wedge \mathrm{end}(u) = \mathrm{end}(v) \wedge \\ &\quad (\forall k : \mathrm{fst}(u) \le k < \mathrm{end}(u) \Rightarrow u[k] = v[k]) \Rightarrow u = v,\\ &\forall x, i, j : \mathrm{fst}(\mathrm{const}(x, i, j)) = i \wedge \mathrm{end}(\mathrm{const}(x, i, j)) = j,\\ &\forall x, i, j, k : i \le k < j \Rightarrow \mathrm{const}(x, i, j)[k] = x,\\ &\forall v, i, j : \mathrm{fst}(v[i{:}j]) = \max(i, \mathrm{fst}(v)) \wedge \mathrm{end}(v[i{:}j]) = \min(j, \mathrm{end}(v)),\\ &\forall v, i, j, k : \mathrm{fst}(v[i{:}j]) \le k < \mathrm{end}(v[i{:}j]) \Rightarrow v[i{:}j][k] = v[k],\\ &\forall v, i, x : \mathrm{fst}(v\{i \leftarrow x\}) = \mathrm{fst}(v) \wedge \mathrm{end}(v\{i \leftarrow x\}) = \mathrm{end}(v),\\ &\forall v, i, x : \mathrm{fst}(v) \le i < \mathrm{end}(v) \Rightarrow v\{i \leftarrow x\}[i] = x,\\ &\forall v, i, x, k : \mathrm{fst}(v) \le k < \mathrm{end}(v) \wedge i \ne k \Rightarrow v\{i \leftarrow x\}[k] = v[k]\big\}\end{aligned}$$

Σ_{BAGOF}-theory $\mathcal{T}_{\mathrm{BAGOF}}$ of bagof function on vectors

$$\Sigma^{\mathrm{S}}_{\mathrm{BAGOF}} = \Sigma^{\mathrm{S}}_{\mathrm{VEC}} \cup \Sigma^{\mathrm{S}}_{\mathrm{BAG}}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{BAGOF}} = \Sigma^{\mathrm{F}}_{\mathrm{VEC}} \cup \Sigma^{\mathrm{F}}_{\mathrm{BAG}} \cup \{\mathrm{bagof} : \mathrm{VEC}_s \to \mathrm{BAG}_s \mid s \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\begin{aligned}\mathcal{T}_{\mathrm{BAGOF}} = \big\{&\forall v : |\mathrm{bagof}(v)| = \max(\mathrm{end}(v) - \mathrm{fst}(v), 0),\\ &\forall v : \mathrm{end}(v) - \mathrm{fst}(v) = 1 \Rightarrow \mathrm{bagof}(v) = [\![v[\mathrm{fst}(v)]]\!]^{(1)},\\ &\forall x, i, j : i \le j \Rightarrow \mathrm{bagof}(\mathrm{const}(x, i, j)) = [\![x]\!]^{(j-i)},\\ &\forall v, k : \mathrm{fst}(v) \le k \le \mathrm{end}(v) \Rightarrow\\ &\quad \mathrm{bagof}(v) = \mathrm{bagof}(v[\mathrm{fst}(v){:}k]) \uplus \mathrm{bagof}(v[k{:}\mathrm{end}(v)])\big\}\end{aligned}$$

Σ_{MAP}-theory $\mathcal{T}_{\mathrm{MAP}}$ of map function on vectors

$$\Sigma^{\mathrm{S}}_{\mathrm{MAP}} = \Sigma^{\mathrm{S}}_{\mathrm{VEC}}$$
$$\Sigma^{\mathrm{F}}_{\mathrm{MAP}} = \Sigma^{\mathrm{F}}_{\mathrm{VEC}} \cup \{f : s \to s' \mid (f{:}s \to s') \in \Sigma^{\mathrm{F}}_{\mathrm{MAP}} \wedge s, s' \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\} \cup \{\mathrm{map}_f : \mathrm{VEC}_s \to \mathrm{VEC}_{s'} \mid (f{:}s \to s') \in \Sigma^{\mathrm{F}}_{\mathrm{MAP}} \wedge s, s' \in \Sigma^{\mathrm{S}}_{\mathrm{E}}\}$$
$$\begin{aligned}\mathcal{T}_{\mathrm{MAP}} = \big\{&\forall v : \mathrm{fst}(\mathrm{map}_f(v)) = \mathrm{fst}(v) \wedge \mathrm{end}(\mathrm{map}_f(v)) = \mathrm{end}(v),\\ &\forall v, k : \mathrm{fst}(v) \le k < \mathrm{end}(v) \Rightarrow \mathrm{map}_f(v)[k] = f(v[k]) \;\big|\; \mathrm{map}_f \in \Sigma^{\mathrm{F}}_{\mathrm{MAP}}\big\}\end{aligned}$$

Fig. 2. Theories of vectors and bags; see Section 3 for details.

Elements. $\mathcal{T}_{\mathrm{E}}$ is a given theory of elements (of vectors and bags). Its signature Σ_{E} is arbitrary but must be disjoint from all other signatures introduced in this section. The theory $\mathcal{T}_{\mathrm{E}}$ is arbitrary, too, but must be decidable and stably infinite so it can be coupled with the theory of multisets, see Section 4.1.

Presburger arithmetic. Σ_{INT} is the signature of Presburger arithmetic, with one sort, two constants and four binary function symbols (for addition, subtraction, minimum and maximum). We introduce the binary predicate symbols $\leq$ and $<$ as abbreviations; we may write $s \leq t$ instead of $\min(s,t) = s$ and $s < t$ instead of $s \leq t \wedge s \neq t$.

The theory $\mathcal{T}_{\mathrm{INT}}$ of Presburger arithmetic is defined as the set of all Σ_{INT}-sentences which are true in $\mathcal{A}_{\mathrm{INT}}$, the standard Σ_{INT}-algebra which interprets the sort INT as the integers and constants and function symbols by their usual meaning.

Multisets. The signature Σ_{BAG} of multisets (with cardinality) extends the signature of Presburger arithmetic with element sorts and multiset sorts BAG_s, one per element sort s. For each element sort, Σ_{BAG} extends Σ_{INT} with a constant $[\![]\!]$ for the empty multiset, a singleton constructor $[\![\cdot]\!]^{(\cdot)}$ (taking an element and its multiplicity), the usual binary operations $\cap$, $\cup$, $\uplus$ for intersection, union and sum, a destructor $\mathrm{count}(\cdot,\cdot)$ for counting the frequency of an element in a multiset, and a destructor $|\cdot|$ for measuring the cardinality (i. e., the number of elements, taking into account their multiplicities) of a multiset. We introduce the binary predicate symbol $\subseteq$ as an abbreviation; we may write $s \subseteq t$ instead of $s \cap t = s$.

Due to the cardinality function, the theory of multisets cannot be finitely axiomatised in our logic.[2] Therefore, the theory $\mathcal{T}_{\mathrm{BAG}}$ of multisets is defined as the set of all Σ_{BAG}-sentences that are true of Δ_{BAG}, the class of standard Σ_{BAG}-algebras. $\mathcal{A}$ is a standard Σ_{BAG}-algebra if it interprets the sort INT as the integers, the sorts BAG_s as the finite multisets over the interpretations of the sorts s, and the constants and function symbols by their usual meanings. Note that the theory $\mathcal{T}_{\mathrm{INT}}$ is contained in $\mathcal{T}_{\mathrm{BAG}}$; stable infiniteness will be relevant in Section 4.1.

Lemma 1. *$\mathcal{T}_{\mathrm{BAG}}$ is stably infinite.*

Vectors. We represent vectors by finite arrays of elements indexed by consecutive integers. The signature Σ_{VEC} of vectors extends the signature of Presburger arithmetic with element sorts and vector sorts VEC_s, one per element sort s. For each element sort, Σ_{VEC} extends Σ_{INT} with two destructors $\mathrm{fst}(\cdot)$ and $\mathrm{end}(\cdot)$ for accessing the first and last (more precisely, the first beyond the last) index of a vector, a destructor $\cdot[\cdot]$ for reading an element of a vector, a constructor $\mathrm{const}(\cdot,\cdot,\cdot)$ for creating a vector filled with a multiple occurrences of the same element, a constructor $\cdot[\cdot{:}\cdot]$ for slicing the subvector in between two indices out of a vector, and a constructor $\cdot\{\cdot \leftarrow \cdot\}$ for updating a vector at an index.

The theory $\mathcal{T}_{\mathrm{VEC}}$ axiomatises vectors. The first axiom is extensionality, equating all vectors that behave equally under the destructors. The remaining axioms define the constructors (uniquely due to extensionality) in terms of the destructors. Note Σ_{VEC} provides no $\mathrm{append}(\cdot,\cdot)$ because $\mathcal{T}_{\mathrm{VEC}}$ forces vector concatenation to be partial.

[2] See [7] for an axiomatisation in a first-order logic extended with an *infinite sum* quantifier.

Given a signature Σ extending Σ_{VEC}, a Σ-algebra $\mathcal{A}$ is called *vector complete* if for all element sorts $s \in \Sigma_{\mathrm{E}}^{\mathrm{S}}$, all integers i, and all finite sequences $x_0, \ldots, x_{k-1} \in s^{\mathcal{A}}$ there is a vector $v \in \mathrm{VEC}_s^{\mathcal{A}}$ such that $\mathrm{fst}(v)^{\mathcal{A}} = i$ and $\mathrm{end}(v)^{\mathcal{A}} = i + k$ and $v[i + l]^{\mathcal{A}} = x_l$ for all integers l with $0 \leq l < k$. A Σ-theory $\mathcal{T}$ is *vector complete* if every $\mathcal{T}$-satisfiable ground $\hat{\Sigma}$-formula has a vector complete model.

Bagof function. The signature Σ_{BAGOF} extends the union of the signature Σ_{VEC} and Σ_{BAG} with functions $\mathrm{bagof}(\cdot)$ mapping vectors to the multisets of their elements. The theory $\mathcal{T}_{\mathrm{BAGOF}}$ axiomatises these functions. The first axiom equates the length of the argument vector with the cardinality of the resulting multiset. The next two axioms define $\mathrm{bagof}(\cdot)$ for the special cases that the argument vector is of length one or constant. The last axiom admits recursive computation of $\mathrm{bagof}(\cdot)$ by splitting the argument into two subvectors and summing the results.

Map function. The signature Σ_{MAP} extends Σ_{VEC} by adding a set F of unary functions on elements (i. e., $(f{:}s \to s') \in F$ implies $s, s' \in \Sigma_{\mathrm{E}}^{\mathrm{S}}$) and a set F_{map} of map functions on vectors such that $(\mathrm{map}_f : \mathrm{VEC}_s \to \mathrm{VEC}_{s'}) \in F_{\mathrm{map}}$ if and only if $(f{:}s \to s') \in F$. Note that Figure 2 specifies Σ_{MAP} by a fixpoint equation which has infinitely many solutions.[3]

The theory $\mathcal{T}_{\mathrm{MAP}}$ axiomatises the functions map_f, in terms of the vector destructors, thus uniquely defining these functions. Note that $\mathcal{T}_{\mathrm{MAP}}$ does not define the unary functions on elements; these functions are intended to be free.

Base theory. We define the Σ-theory $\mathcal{T}_{\mathrm{BASE}} = \mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}} \cup \mathcal{T}_{\mathrm{VEC}} \cup \mathcal{T}_{\mathrm{MAP}}$ as the union of the above theories excluding $\mathcal{T}_{\mathrm{BAGOF}}$, where $\Sigma = \Sigma_{\mathrm{E}} \cup \Sigma_{\mathrm{BAG}} \cup \Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}} \cup \Sigma_{\mathrm{BAGOF}}$ is the union of the above signatures (including Σ_{BAGOF}, i. e., $\mathcal{T}_{\mathrm{BASE}}$ leaves the bagof functions uninterpreted). The following model-theoretic properties will become relevant in Section 5.

Lemma 2. *$\mathcal{T}_{\mathrm{BASE}}$ is vector complete and stably infinite.*

4 Known Decision Procedures Applied to Bags and Vectors

This section employs known results to obtain a decision procedure for ground satisfiability in the combination of the theories of elements, multisets and vectors (including the theory of map functions). We will make repeated use of the following result on the combination of arbitrary theories with free functions.

Proposition 3 (Sofronie-Stokkermans 2005 [9]). *Let $\Sigma' \supseteq \Sigma$ be signatures and let $\mathcal{T}$ be a Σ-theory. If $\mathcal{T}$-satisfiability is decidable for ground $\hat{\Sigma}$-formulae then $\mathcal{T}$-satisfiability is decidable for ground $\hat{\Sigma}'$-formulae.*

[3] Extremal solutions are uninteresting. The least solution would yield $\Sigma_{\mathrm{MAP}} = \Sigma_{\mathrm{VEC}}$, and the greatest solution would likely violate the requirement that Σ_{E} and Σ_{MAP} be disjoint.

The decision procedure behind Proposition 3 reduces a ground $\hat{\Sigma'}$-formula in negation normal form[4] (NNF) to a $\mathcal{T}$-equisatisfiable ground $\hat{\Sigma}$-formula in NNF; the reduction may cause a quadratic blowup.

4.1 Combining the Theories of Elements and Multisets

A decision procedure for the theory $\mathcal{T}_{\mathrm{BAG}}$ of multisets with cardinality is known [7]. We combine this decision procedure with an arbitrary decision procedure for the theory $\mathcal{T}_{\mathrm{E}}$ of elements, using the Nelson-Oppen combination method [6, 11]. This is possible because $\mathcal{T}_{\mathrm{E}}$ and $\mathcal{T}_{\mathrm{BAG}}$ are stably infinite theories (cf. Figure 2 and Lemma 1) over disjoint signatures.

Proposition 4. *Ground $(\mathcal{T}_{\mathrm{E}} \cup \mathcal{T}_{\mathrm{BAG}})$-satisfiability is decidable.*

4.2 Deciding the Theory of Vectors (Including Map)

We use a decision procedure for the Array Property Fragment [3] to decide ground satisfiability in the union of the theories of vectors and map functions. The procedure reduces the satisfiability problem to ground satisfiability in the combination of the theories of Presburger arithmetic, uninterpreted functions and an unspecified theory of vector elements.

Proposition 5. *Let $\mathcal{T}_0$ be a Σ_0-theory where the signature Σ_0 shares no non-constant function symbols with $\Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}$ except for the function symbols in Σ_{INT}, formally $\Sigma_0 \cap (\Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}) \subseteq \hat{\Sigma_{\mathrm{INT}}}$. Let $\Sigma_1 = \Sigma_0 \cup \Sigma_{\mathrm{INT}} \cup \Sigma_{\mathrm{VEC}} \cup \Sigma_{\mathrm{MAP}}$ and $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{T}_{\mathrm{INT}} \cup \mathcal{T}_{\mathrm{VEC}} \cup \mathcal{T}_{\mathrm{MAP}}$. If $(\mathcal{T}_0 \cup \mathcal{T}_{\mathrm{INT}})$-satisfiability is decidable for ground $\hat{\Sigma}$-formulae, where Σ extends $\Sigma_0 \cup \Sigma_{\mathrm{INT}}$, then $\mathcal{T}_1$-satisfiability is decidable for ground $\hat{\Sigma_1}$-formulae.*

Proof. Let ϕ be a ground $\hat{\Sigma_1}$-formula (in NNF). Perform the following reductions.

1. Eliminate disequalities and updates: Normalise ϕ w. r. t. the rewrite rules NOTEQ and UPDATE from Figure 3. NOTEQ expresses disequalities $s \neq t$ using extensionality and Skolemisation. UPDATE is based on expressing equations $v = u\{i \leftarrow x\}$ by splitting u and v into three subvectors each (a prefix up to index i, a middle part of length 1 at index i and a suffix from index $i+1$) and equating these accordingly (in particular, equating the middle part of v to a constant vector). The resulting ground $\hat{\Sigma_1}$-formula ϕ' is $\mathcal{T}_1$-equisatisfiable to ϕ but contains no vector disequalities and updates.
2. Purify w. r. t. vector sorts: In a bottom up manner, rewrite $\phi'[t]$ to $\phi'[c] \wedge c = t$, where c is a fresh constant and t a non-constant vector term. The result of normalising ϕ' w. r. t. the above rule is a $\mathcal{T}_1$-equisatisfiable $\hat{\Sigma_1}$-formula ϕ'' such that
   - for all terms of the form $\mathrm{fst}(u)$ or $\mathrm{end}(u)$ or $u[i]$, u is a constant, and
   - all vector atoms are of the form $u = v$ or $v = u[i{:}j]$ or $v = \mathrm{const}(x, i, j)$ or $v = \mathrm{map}_f(u)$, where u and v are constants.

[4] The procedure in [9] expects input in clause form; however, the reduction works just as well for formulae in NNF.

$$[\textsc{NotEq}]\ \frac{\phi\big[u \neq v\big]}{\phi\left[\begin{array}{l}\mathrm{fst}(u) \neq \mathrm{fst}(v) \vee \mathrm{end}(u) \neq \mathrm{end}(v) \vee \\ (\mathrm{fst}(u) \leq k < \mathrm{end}(u) \wedge u[k] \neq v[k])\end{array}\right]}\ \text{if } u, v \text{ vectors} \wedge k \text{ fresh}$$

$$[\textsc{Read}]\ \frac{\phi\big[u[i]\big]}{\phi\big[x\big] \wedge u[i{:}i+1] = \mathrm{const}(x, i, i+1)}\ \text{if } x \text{ fresh}$$

$$[\textsc{Update}]\ \frac{\phi\big[u\{i \leftarrow x\}\big]}{\phi\big[v\big] \wedge \psi(v, u, i, x)}\ \text{if } v \text{ fresh}$$

$$\text{where } \psi(v,u,i,x) \equiv \left\{\begin{array}{l}\big(\mathrm{fst}(u) \leq i < \mathrm{end}(u) \vee u = v\big) \wedge \big(i < \mathrm{fst}(u) \vee \mathrm{end}(u) \leq i \vee \\ \quad \big(\mathrm{fst}(v) = \mathrm{fst}(u) \wedge \mathrm{end}(v) = \mathrm{end}(u) \wedge \\ \quad\ v[\mathrm{fst}(u){:}i] = u[\mathrm{fst}(u){:}i] \wedge v[i{:}i+1] = \mathrm{const}(x, i, i+1) \wedge \\ \quad\ v[i+1{:}\mathrm{end}(u)] = u[i+1{:}\mathrm{end}(u)]\big)\big)\end{array}\right.$$

$$[\textsc{Bagof}]\ \frac{\phi\big[\mathrm{bagof}(u)\big]}{\phi\big[b\big] \wedge b = \mathrm{bagof}(u)}\ \text{if } b \text{ fresh}$$

$$[\textsc{SubConst}]\ \frac{\phi\big[v = u[k{:}l] \wedge u = \mathrm{const}(x, i, j)\big]}{\phi\big[v = \mathrm{const}(x, \max(k, i), \min(l, j)) \wedge u = \mathrm{const}(x, i, j)\big]}$$

$$[\textsc{MapConst}]\ \frac{\phi\big[v = \mathrm{map}_f(u) \wedge u = \mathrm{const}(x, i, j)\big]}{\phi\big[v = \mathrm{const}(f(x), i, j) \wedge u = \mathrm{const}(x, i, j)\big]}$$

Fig. 3. Vector transformations; see sections 4.2 and 5.1 for details.

3. Eliminate all subterms of the form $\mathrm{fst}(u)$ and $\mathrm{end}(u)$ in ϕ'' by replacing them with INT-constants fst_u and end_u, respectively, introducing two new INT-constants fst_u, end_u per vector constant u. Then normalise ϕ'' w. r. t. all rewrite rules in Figure 4. This results in a $\mathcal{T}_1$-equisatisfiable $\hat{\Sigma}_1$-formula ϕ''', which falls into the Array Property Fragment [3].
4. Use decision procedure for the Array Property Fragment outlined in [3]:
   - Instantiate universal quantifiers in ϕ'''.
   - Replace all constants u of sort VEC_s by unary functions $f_u : \mathrm{INT} \to s$, and replace all terms of the form $u[i]$ by $f_u(i)$.

   The resulting ground $\hat{\Sigma}$-formula ϕ'''' is $(\mathcal{T}_0 \cup \mathcal{T}_{\mathrm{INT}})$-satisfiable if and only if ϕ''' is $\mathcal{T}_1$-satisfiable, where Σ extends $\Sigma_0 \cup \Sigma_{\mathrm{INT}}$ with the above unary functions f_u and with the unary functions f on element sorts from signature Σ_{MAP}. □

4.3 Deciding the Base Theory

Finally, we pull the results of the previous subsections together to obtain a decision procedure for $\mathcal{T}_{\mathrm{BASE}}$, the union of all theories introduced in Section 3 excluding $\mathcal{T}_{\mathrm{BAGOF}}$. Recall that the signature Σ of $\mathcal{T}_{\mathrm{BASE}}$ includes Σ_{BAGOF}, i. e., $\mathcal{T}_{\mathrm{BASE}}$ treats the bagof functions as free.

$$[\text{EQ}]\ \frac{\phi\big[u = v\big]}{\phi\big[\mathit{fst}_u = \mathit{fst}_v \wedge \mathit{end}_u = \mathit{end}_v \wedge \forall k : \mathit{fst}_u \leq k < \mathit{end}_u \Rightarrow u[k] = v[k]\big]}$$

$$[\text{SUB}]\ \frac{\phi\big[v = u[i{:}j]\big]}{\phi\left[\begin{array}{l}\mathit{fst}_v = \max(i, \mathit{fst}_u) \wedge \mathit{end}_v = \min(j, \mathit{end}_u) \wedge \\ \forall k : \mathit{fst}_v \leq k < \mathit{end}_v \Rightarrow v[k] = u[k]\end{array}\right]}$$

$$[\text{CONST}]\ \frac{\phi\big[v = \text{const}(x, i, j)\big]}{\phi\big[\mathit{fst}_v = i \wedge \mathit{end}_v = j \wedge \forall k : \mathit{fst}_v \leq k < \mathit{end}_v \Rightarrow v[k] = x\big]}$$

$$[\text{MAP}]\ \frac{\phi\big[v = \text{map}_f(u)\big]}{\phi\big[\mathit{fst}_v = \mathit{fst}_u \wedge \mathit{end}_v = \mathit{end}_u \wedge \forall k : \mathit{fst}_v \leq k < \mathit{end}_v \Rightarrow v[k] = f(u[k])\big]}$$

Fig. 4. Translating to the Array Property Fragment; see Section 4.2 for details.

Proposition 6. *Ground $\mathcal{T}_{\text{BASE}}$-satisfiability is decidable.*

Proof. Let ϕ be $\hat{\Sigma}$-formula (in NNF).

1. Reduce ϕ to a $\mathcal{T}$-equisatisfiable ground $\hat{\Sigma'}$-formula ϕ' where $\Sigma' = \Sigma_{\text{E}} \cup \Sigma_{\text{BAG}} \cup \Sigma_{\text{VEC}} \cup \Sigma_{\text{MAP}}$, using the decision procedure for free functions (Proposition 3).
2. Reduce ϕ' to a ground $\hat{\Sigma''}$-formula ϕ'' using the decision procedure for vectors (Proposition 5; the Σ_0-theory $\mathcal{T}_0$ there is $\mathcal{T}_{\text{E}} \cup \mathcal{T}_{\text{BAG}}$ here). The resulting signature Σ'' extends $\Sigma_{\text{E}} \cup \Sigma_{\text{BAG}}$ by free unary functions on element sorts (stemming from signature Σ_{MAP}) and free unary functions from INT to element sorts (arising from encoding arrays as unary functions). The formula ϕ'' is $(\mathcal{T}_{\text{E}} \cup \mathcal{T}_{\text{BAG}})$-satisfiable iff ϕ' is $\mathcal{T}$-satisfiable.
3. Reduce ϕ'' to a $(\mathcal{T}_{\text{E}} \cup \mathcal{T}_{\text{BAG}})$-equisatisfiable ground $\hat{\Sigma'''}$-formula ϕ''' where $\Sigma''' = \Sigma_{\text{E}} \cup \Sigma_{\text{BAG}}$, using the decision procedure for free functions (Proposition 3).
4. Check $(\mathcal{T}_{\text{E}} \cup \mathcal{T}_{\text{BAG}})$-satisfiability of ϕ''' using the combined decision procedure for elements and multisets (Proposition 4). □

5 A Decision Procedure for Bags, Vectors and Bagof Functions

Recall the Σ-theory $\mathcal{T}_{\text{BASE}}$, defined in Section 3 as the union of all theories excluding $\mathcal{T}_{\text{BAGOF}}$, where Σ is the union of all signatures (including Σ_{BAGOF}). For this section, let $\mathcal{T} = \mathcal{T}_{\text{BASE}} \cup \mathcal{T}_{\text{BAGOF}}$ be the Σ-theory extending $\mathcal{T}_{\text{BASE}}$ with the axioms for the bagof functions.

5.1 Decision Procedure

The decision procedure relies on reducing ground $\mathcal{T}$-satisfiability to ground $\mathcal{T}_{\text{BASE}}$-satisfiability by instantiating axioms of $\mathcal{T}_{\text{BAGOF}}$. The reduction is shown in Figure 5. Termination is obvious. Soundness is established by the lemma below.

Input: Ground $\hat{\Sigma}$-formula ϕ_0 (in NNF).
Output: Ground $\hat{\Sigma}$-formula ϕ_6.
Algorithm:

1. Eliminate definable vector operators, purify and simplify:
   (a) Construct ϕ_1 by normalising ϕ_0 w. r. t. the rule NOTEQ (Figure 3).
   (b) Construct ϕ_2 by normalising ϕ_1 w. r. t. the rules READ, UPDATE and BAGOF (Figure 3).
   (c) Construct ϕ_3 by purifying ϕ_2 w. r. t. vector sorts: In a bottom up manner, rewrite $\phi_2[t]$ to $\phi_2[c] \wedge c = t$, where c is a fresh constant and t a non-constant vector term.
   (d) Construct ϕ_4 by converting ϕ_3 into disjunctive normal form (DNF).
   (e) Construct ϕ_5 by normalising ϕ_4 w. r. t. the rules SUBCONST and MAPCONST (Figure 3).
2. Determine the sets of *vector constants* C, *element terms* E and *index terms* I:

$$\begin{aligned} C &= \{v \mid v \text{ vector constant occurring in } \phi_5\} \\ E &= \{x \mid \exists i,j : \mathrm{const}(x,i,j) \text{ occurs in } \phi_5\} \text{ and} \\ I &= \{\mathrm{fst}(u), \mathrm{end}(u) \mid u \in C\} \cup \\ &\quad \{i,j \mid \exists x : \mathrm{const}(x,i,j) \text{ occurs in } \phi_5 \vee \exists u : u[i{:}j] \text{ occurs in } \phi_5\} \ . \end{aligned}$$

3. Instantiate (variants of) the $\mathcal{T}_{\mathrm{BAGOF}}$ axioms with terms generated from C, E and I:

$$\phi_6 \equiv \phi_5 \wedge \bigwedge_{u\in C; i,j\in I} \mathrm{Ax}_1^{u,i,j} \wedge \bigwedge_{x\in E; i,j\in I} \mathrm{Ax}_3^{x,i,j} \wedge \bigwedge_{u\in C; i,j,k\in I} \mathrm{Ax}_4^{u,i,j,k}$$

$$\begin{aligned} \text{where} \quad \mathrm{Ax}_1^{u,i,j} &\equiv \mathrm{fst}(u) \le i \le j \le \mathrm{end}(u) \Rightarrow |\mathrm{bagof}(u[i{:}j])| = j - i \\ \mathrm{Ax}_3^{x,i,j} &\equiv i \le j \Rightarrow \mathrm{bagof}(\mathrm{const}(x,i,j)) = [\![x]\!]^{(j-i)} \\ \mathrm{Ax}_4^{u,i,j,k} &\equiv \mathrm{fst}(u) \le i \le k \le j \le \mathrm{end}(u) \Rightarrow \\ &\quad \mathrm{bagof}(u[i{:}j]) = \mathrm{bagof}(u[i{:}k]) \uplus \mathrm{bagof}(u[k{:}j]) \end{aligned}$$

Fig. 5. Reduction to base theory by instantiating $\mathcal{T}_{\mathrm{BAGOF}}$ axioms.

Lemma 7 (Soundness). *If ϕ_0 is $\mathcal{T}$-satisfiable then ϕ_6 is $\mathcal{T}_{\mathrm{BASE}}$-satisfiable.*

Proof. As ϕ_0 and ϕ_5 are $\mathcal{T}$-equisatisfiable, it suffices to show that every $\mathcal{T}$-model is a model of the instances $\mathrm{Ax}_1^{u,i,j}$, $\mathrm{Ax}_3^{x,i,j}$ and $\mathrm{Ax}_4^{u,i,j,k}$, for all $u \in C$, $x \in E$ and $i,j,k \in I$.

- $\mathrm{Ax}_1^{u,i,j}$ follows from the first $\mathcal{T}_{\mathrm{BAGOF}}$ axiom (after instantiating v with $u[i{:}j]$) as in $\mathcal{T}_{\mathrm{VEC}}$, $\mathrm{fst}(u) \le i \le j \le \mathrm{end}(u)$ implies $\max(\mathrm{end}(u[i{:}j]) - \mathrm{fst}(u[i{:}j]), 0) = j - i$.
- $\mathrm{Ax}_3^{x,i,j}$ is an instance of the third $\mathcal{T}_{\mathrm{BAGOF}}$ axiom.
- $\mathrm{Ax}_4^{u,i,j,k}$ follows from the fourth $\mathcal{T}_{\mathrm{BAGOF}}$ axiom (after instantiating v with $u[i{:}j]$ and k with k) because in $\mathcal{T}_{\mathrm{VEC}}$, the antecedent $\mathrm{fst}(u) \le i \le k \le j \le \mathrm{end}(u)$ implies $u[i{:}j][\mathrm{fst}(u[i{:}j]){:}k] = u[i{:}k]$ and $u[i{:}j][k{:}\mathrm{end}(u[i{:}j])] = u[k{:}j]$. □

Before we show completeness of the reduction, we point out that step 1 converts the input formula ϕ_0 to a ground DNF formula ϕ_5 such that

- bagof occurs only in atoms of the form $b = \mathrm{bagof}(u)$, where b and u are constants,

- all vector atoms are of the form $u = v$ or $v = u[i{:}j]$ or $v = \mathrm{const}(x, i, j)$ or $v = \mathrm{map}_f(u)$, where u and v are constants,
- all other vector terms are of the form $\mathrm{fst}(u)$ or $\mathrm{end}(u)$, where u is a constant, and
- the arguments of map_f are non-constant, i. e., whenever $\mathrm{map}_f(u)$ occurs in a disjunct ψ then there are no terms x, i and j such that the atom $u = \mathrm{const}(x, i, j)$ would logically follow from ψ in theory $\mathcal{T}_{\mathrm{BASE}}$. Note that this last property is achieved by conversion to DNF and propagation of constant vectors within each disjunct (steps 1d and 1e in Figure 5).

5.2 Completeness in the Absence of Map Functions

We call the signature Σ_{MAP} *trivial* if $\Sigma_{\mathrm{MAP}} = \Sigma_{\mathrm{VEC}}$, i. e., there are no unary functions on elements and no map functions. By model-theoretic arguments, we prove completeness of the reduction shown in Figure 5, given that Σ_{MAP} is trivial.

Lemma 8 (Completeness without map**).** *Assume Σ_{MAP} trivial. If ϕ_6 is $\mathcal{T}_{\mathrm{BASE}}$-satisfiable then ϕ_0 is $\mathcal{T}$-satisfiable.*

Proof. Assume a $\hat{\Sigma}$-algebra $\mathcal{A}$ which is a $\mathcal{T}_{\mathrm{BASE}}$-model of ϕ_6; w. l. o. g. we assume that $\mathcal{A}$ is vector complete (cf. Lemma 2). It suffices to construct a $\hat{\Sigma}$-algebra $\mathcal{A}'$ which is a $\mathcal{T}$-model of one disjunct ψ of ϕ_5; we assume that $\mathcal{A} \models \psi$.

Recall that C is the set of vector constants occurring in ϕ_5. We choose $\mathcal{A}'$ so that

1. $\mathcal{A}$ and $\mathcal{A}'$ agree on the interpretations of all sorts, all constants except vector constants occurring in ϕ_5, and all function symbols except the bagof functions,
2. $\mathcal{A}'$ interprets $\mathrm{bagof} : \mathrm{VEC}_s \to \mathrm{BAG}_s$ as functions mapping vectors in $\mathrm{VEC}_s^{\mathcal{A}'}$ to the multisets of their elements in $\mathrm{BAG}_s^{\mathcal{A}'}$,
3. $\mathcal{A}'$ interprets vector constants u occurring in ϕ_5 such that $\mathcal{A}$ and $\mathcal{A}'$ agree
   (a) on the interpretations of the ground terms $\mathrm{fst}(u)$ and $\mathrm{end}(u)$, and
   (b) on the interpretations of the ground term $\mathrm{bagof}(u)$.

We have to explain how the interpretations of vector constants can be chosen in such a way that item (3b) holds, i. e., how to keep the interpretations of ground terms $\mathrm{bagof}(u)$ invariant even though the interpretations of the bagof functions change.

Recall the set of index terms I defined in step 2 of the reduction (Figure 5). Let $\langle i_1, \ldots, i_n \rangle$ be an enumeration of I such that $\mathcal{A}$ orders their interpretations in ascending sequence $i_1^{\mathcal{A}} \leq \cdots \leq i_n^{\mathcal{A}}$. Items (1) to (3a) ensure that $\mathcal{A}$ and $\mathcal{A}'$ agree on the interpretations of index terms $i_j \in I$, hence $\mathcal{A}'$ orders their interpretation $i_j^{\mathcal{A}'}$ in the same sequence.

Item (3b) is achieved by an inductive process. Let $j < n$ be minimal such that there is $u \in C$ with $\mathrm{fst}(u)^{\mathcal{A}} \leq i_j^{\mathcal{A}} \leq i_{j+1}^{\mathcal{A}} \leq \mathrm{end}(u)^{\mathcal{A}}$ and $\mathrm{bagof}(u[i_j{:}i_{j+1}])^{\mathcal{A}}$ differing from the multiset of elements in $u[i_j{:}i_{j+1}]^{\mathcal{A}}$. Note that there can be no $x \in E$ — recall the set E of element terms occurring in ϕ_5 — such that $\mathrm{const}(x, i_j, i_{j+1})^{\mathcal{A}} = u[i_j{:}i_{j+1}]^{\mathcal{A}}$. For if there were such $x \in E$ then the $\mathcal{T}_{\mathrm{BAGOF}}$ instance $\mathrm{Ax}_3^{x,i_j,i_{j+1}}$ (appearing as a conjunct in ϕ_6) would ensure that $\mathrm{bagof}(u[i_j{:}i_{j+1}])^{\mathcal{A}}$ equals the multiset of elements

in $u[i_j{:}i_{j+1}]^{\mathcal{A}}$. Now let C_u be the set of vector constants whose slice between i_j and i_{j+1} happens to equal $u[i_j{:}i_{j+1}]$ in $\mathcal{A}$, formally

$$C_u = \{v \in C \mid \mathcal{A} \models \mathrm{fst}(v) \leq i_j \leq i_{j+1} \leq \mathrm{end}(v) \wedge u[i_j{:}i_{j+1}] = v[i_j{:}i_{j+1}]\} \ .$$

Let $\langle x_0, x_1, \ldots, x_{k-1}\rangle$ be an enumeration of the multiset $\mathrm{bagof}(u[i_j{:}i_{j+1}])^{\mathcal{A}}$. Note that the $\mathcal{T}_{\mathrm{BAGOF}}$ instance $\mathrm{Ax}_1^{u,i_j,i_{j+1}}$ constrains the size of the multiset so that $k = i_{j+1}^{\mathcal{A}} - i_j^{\mathcal{A}}$. As $\mathcal{A}$ is vector complete, we can choose the interpretations of all $v \in C_u$ such that for all $l < k$, $v^{\mathcal{A}'}$ stores x_l at index $i_j^{\mathcal{A}'} + l$. This ensures that $\mathcal{A}' \models u[i_j{:}i_{j+1}] = v[i_j{:}i_{j+1}]$. The construction proceeds from there by induction on j.

After the construction is completed, one can show that $\mathcal{A}$ and $\mathcal{A}'$ do in fact agree on the interpretation of $\mathrm{bagof}(u)$, for all $u \in C$. The proof is by induction on the length $\mathrm{end}(u) - \mathrm{fst}(u)$ of u and uses the $\mathcal{T}_{\mathrm{BAGOF}}$ instances $\mathrm{Ax}_4^{u,i,j,k}$, for all $i, j, k \in I$ such that $\mathcal{A} \models \mathrm{fst}(u) \leq i \leq k \leq j \leq \mathrm{end}(u)$.

Obviously, $\mathcal{A}'$ is a model of $\mathcal{T}_{\mathrm{BAGOF}}$ (and thus of $\mathcal{T}$) as that is how the interpretation of the bagof functions was chosen. To show that $\mathcal{A}' \models \psi$, it suffices to show that $\mathcal{A}'$ satisfies every vector atom that $\mathcal{A}$ satisfies (because $\mathcal{A}$ and $\mathcal{A}'$ agree on the interpretation of non-vector literals and all vector literals occurring in ψ are positive). In the case of atoms of the form $v = \mathrm{const}(x, i, j)$ this is so because the construction does not change the interpretation of v. In the case of atoms of the form $u = v$ or $v = u[i{:}j]$, the construction alters the interpretations of corresponding slices of u and v uniformly. □

The decidability of ground satisfiability in the theories of elements, multisets, vectors (excluding map functions) and the bagof function follows from soundness and completeness of the reduction (lemmas 7 and 8) and from decidability of the base theory (Proposition 6).

Theorem 9. *Assume Σ_{MAP} trivial. Then ground $\mathcal{T}$-satisfiability is decidable.*

We remark that the conversion to DNF (step 1d in Figure 5) during the reduction is not necessary if Σ_{MAP} is trivial; NNF is all that's required in that case.

5.3 Completeness in the Presence of Map Functions

To prove completeness of the reduction from Figure 5 when Σ_{MAP} is not trivial, we need syntactic restrictions on the occurrences of map functions in the input formula.

Given a set of element sorts $S \subseteq \Sigma_{\mathrm{E}}^{\mathrm{S}}$, we say a term t is a *S-term* (resp. *VEC_S-term*) if t is a s-term (resp. VEC_s-term) for some $s \in S$. A ground $\hat{\Sigma}$-formula ϕ is *stratified* if there is a partition $\{S_1, \ldots, S_m\}$ of the set of element sorts $\Sigma_{\mathrm{E}}^{\mathrm{S}}$ such that

- for every subterm $\mathrm{map}_f(u)$ of ϕ there are strata S_i and S_{i+1} such that u is a VEC_{S_i}-term and $\mathrm{map}_f(u)$ is a $\mathrm{VEC}_{S_{i+1}}$-term, and
- all arguments of $\mathrm{bagof}(\cdot)$ in ϕ are uniformly VEC_{S_m}-terms.

The verification condition VC from Figure 1 is an example of a stratified formula. Given the strata $S_1 = \{\texttt{String}\}$ and $S_2 = \{\texttt{Resource}\}$, it is easy to check that $\mathrm{map}_{\texttt{MessageResource}}$ maps vectors of strings to vectors of resources, and that all arguments of $\mathrm{bagof}(\cdot)$ are vectors of resources. On the other hand, a formula containing a function symbol $\mathrm{map}_f : \mathrm{VEC}_s \to \mathrm{VEC}_{s'}$ fails to be stratified if $s = s'$, for instance.

Lemma 10 (Completeness for stratified input). *Assume ϕ_0 stratified. If ϕ_6 is $\mathcal{T}_{\mathrm{BASE}}$-satisfiable then ϕ_0 is $\mathcal{T}$-satisfiable.*

Proof (Sketch). Let $S_1, \ldots, S_m$ be the strata for ϕ_0. As stratification is preserved by step 1 of the reduction, ϕ_5 is stratified w. r. t. the same strata. Recall the set C of vector constants defined in step 2 of the reduction. Stratification induces a partition $\{C_1, \ldots, C_m\}$ of C such that each C_i contains the VEC_{S_i}-constants occurring in ϕ_5. We modify step 3 of the reduction slightly by generating instances of $\mathrm{Ax}_1^{u,i,j}$ and $\mathrm{Ax}_4^{u,i,j,k}$ only for $u \in C_m$.

Now, assume a $\hat{\Sigma}$-algebra $\mathcal{A}$ (which due to Lemma 2 can be assumed vector complete and stably infinite[5]) which is a $\mathcal{T}_{\mathrm{BASE}}$-model of ϕ_6. The construction of a $\mathcal{T}$-model $\mathcal{A}'$ of a disjunct ψ of ϕ_5 is similar to the one in Lemma 8 except for the fact that now $\mathcal{A}'$ may not only change the interpretations of $\mathrm{bagof}(\cdot)$ and of vector constants but also the interpretations of function symbols from signature Σ_{MAP}. The construction proceeds in m phases, yielding a sequence $\langle \mathcal{A}_m, \mathcal{A}_{m-1}, \ldots, \mathcal{A}_1 \rangle$ of $\hat{\Sigma}$-algebras.

The first phase constructs a $\hat{\Sigma}$-algebra $\mathcal{A}_m$ fixing the interpretations of the bagof functions and the vector constants in C_m; this construction is analogous to the proof of Lemma 8. Changing the interpretation some constant $v \in C_m$ may falsify some atom of the form $v = \mathrm{map}_f(u)$. To rectify this, the second phase constructs a $\hat{\Sigma}$-algebra $\mathcal{A}_{m-1}$ fixing the interpretations of vector constants in C_{m-1} (and possibly changing the interpretations of functions in Σ_{MAP}) in order to restore the truth of $v = \mathrm{map}_f(u)$. This in turn may falsify some other map atom, whose truth is restored by constructing $\mathcal{A}_{m-2}$, and so on.

We present the construction of $\mathcal{A}_{m-1}$ in more detail; recall that we assume that $\mathcal{A} \models \psi$, and that ψ is a conjunction of literals. Let $\langle i_1, i_2, \ldots, i_n \rangle$ be the ascending enumeration of index terms as defined in the proof of Lemma 8. Let $j < n$ be minimal such that ψ contains some atom $v = \mathrm{map}_f(u)$ with $\mathcal{A}_m \not\models v[i_j{:}i_{j+1}] = \mathrm{map}_f(u[i_j{:}i_{j+1}])$. Because $\mathcal{A}$ and $\mathcal{A}_m$ essentially differ in the interpretations of vector constants in C_m, we conclude that $v \in C_m$, hence $u \in C_{m-1}$ due to stratification. In $\mathcal{A}_{m-1}$, we change the interpretation of u (and of all u' with $\mathcal{A}_m \models u'[i_j{:}i_{j+1}] = u[i_j{:}i_{j+1}]$) such that the elements of $u[i_j{:}i_{j+1}]^{\mathcal{A}_{m-1}}$ are fresh and pairwise distinct. Freshness means that the elements of $u[i_j{:}i_{j+1}]^{\mathcal{A}_{m-1}}$ occur neither in the $\mathcal{A}$- nor in the $\mathcal{A}_m$-interpretation of any element or vector constant. Because $\mathcal{A}$ and $\mathcal{A}_m$ (which features the same carriers) are stably infinite and vector complete, we can always find enough fresh elements and create arbitrary vectors from them. Next, we change the interpretation of the free function f. Define $f^{\mathcal{A}_{m-1}}$ such that $f^{\mathcal{A}_{m-1}}(u^{\mathcal{A}_{m-1}}[l]) = v^{\mathcal{A}_{m-1}}[l]$, for all integers l with $i_j^{\mathcal{A}_{m-1}} \leq l < i_{j+1}^{\mathcal{A}_{m-1}}$. Due to freshness of the elements in $u[i_j{:}i_{j+1}]^{\mathcal{A}_{m-1}}$, the function $f^{\mathcal{A}_{m-1}}$ is well-defined. The construction proceeds by induction on j.

It is obvious that $\mathcal{A}_{m-1} \models v[i_j{:}i_{j+1}] = \mathrm{map}_f(u[i_j{:}i_{j+1}])$. What remains to be shown is that the construction preserves the truth of other vector atoms occurring in ψ. In the case of atoms of the form $u' = u$ or $u' = u[i{:}j]$, the argument is the same as in the proof of Lemma 8: Both sides are altered uniformly. Finally, the case of atoms of the form $u = \mathrm{const}(x, i, j)$ cannot arise because if it did then step 1e of the reduction would

[5] By abuse of notation, we call a Σ-algebra $\mathcal{A}$ *stably infinite* if all its carriers are infinite.

have propagated the constant vector through map_f, replacing the atom $v = \mathrm{map}_f(u)$ with $v = \mathrm{const}(f(x), i, j)$. □

The decidability of satisfiability of stratified ground formulae in the theories of elements, multisets, vectors, map functions and the bagof function follows; the proof is similar to Theorem 9.

Theorem 11. *Ground $\mathcal{T}$-satisfiability is decidable for stratified ground $\hat{\Sigma}$-formulae.*

Relation to local theory extensions. The way the reduction in Figure 5 instantiates universal quantifiers with selected ground terms is reminiscent of local theory extensions [5], and one may wonder whether the theory $\mathcal{T}$ can be viewed as a local extension of the theory $\mathcal{T}_{\mathrm{BASE}}$. However, our model construction does not fit entirely into the framework of local theory extensions because not only does it extend partial extension functions (like the bagof functions) to total ones but also changes the interpretations of base constants and free base functions. It remains to be seen whether the framework of local theory extensions can be suitably generalised to encompass our construction.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein.

References

[1] D. Aspinall, P. Maier, and I. Stark. Monitoring external resources in Java MIDP. *Electr. Notes Theor. Comput. Sci.*, 197(1):17–30, 2008.

[2] D. Aspinall, P. Maier, and I. Stark. Safety guarantees from explicit resource management. In *Proc. FMCO 2007*, LNCS 5382, pages 52–71. Springer, 2008.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI 2006*, LNCS 3855, pages 427–442. Springer, 2006.

[4] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.

[5] C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *Proc. TACAS 2008*, LNCS 4963, pages 265–281. Springer, 2008.

[6] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[7] R. Piskac and V. Kuncak. Decision procedures for multisets with cardinality constraints. In *Proc. VMCAI 2008*, LNCS 4905, pages 218–232. Springer, 2008.

[8] R. Piskac and V. Kuncak. Linear arithmetic with stars. In *Proc. CAV 2008*, LNCS 5123, pages 268–280. Springer, 2008.

[9] V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In R. Nieuwenhuis, editor, *Proc. CADE-20*, LNCS 3632, pages 219–234. Springer, 2005.

[10] N. Suzuki and D. Jefferson. Verification decidability of presburger array programs. *J. ACM*, 27(1):191–205, 1980.

[11] C. Tinelli and C. G. Zarba. Combining decision procedures for sorted theories. In *Proc. JELIA 2004*, LNCS 3229, pages 641–653. Springer, 2004.

[12] C. G. Zarba. Combining multisets with integers. In *Proc. CADE-18*, LNCS 2392, pages 363–376. Springer, 2002.

Termination Analysis of Java Bytecode

Elvira Albert[1], Puri Arenas[1], Michael Codish[2],
Samir Genaim[3], Germán Puebla[3], and Damiano Zanardini[3]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CS, Ben-Gurion University of the Negev, Israel
[3] CLIP, Technical University of Madrid (UPM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming, and recently also for imperative languages. In existing approaches, termination is performed on source programs. However, there are many situations, including mobile code, where only the compiled code is available. In this work we present an automatic termination analysis for sequential Java Bytecode programs. Such analysis presents all of the challenges of analyzing a low-level language as well as those introduced by object-oriented languages. Interestingly, given a bytecode program, we produce a *constraint logic* program, CLP, whose termination entails termination of the bytecode program. This allows applying the large body of work in termination of CLP programs to termination of Java bytecode. A prototype analyzer is described and initial experimentation is reported.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination.

In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [15], and logic and functional languages [18,10,17]. Termination-proving techniques are also emerging in the imperative paradigm [6,11,15], even for dealing with large industrial code [11].

Static analysis of *Java ByteCode* (JBC for short) has received considerable attention lately [25,23,24,22,1]. The present paper presents a static analysis for sequential JBC which is, to the best of our knowledge, the first approach to proving termination. Bytecode is a low-level representation of a program, designed to be executed by a virtual machine rather than by dedicated hardware. As such, it is usually higher level than actual machine code, and independent of

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 2–18, 2008.

the specific hardware. This, together with its security features, makes JBC [19] the chosen language for many *mobile code* applications. In this context, analysis of JBC programs may enable performing a certain degree of static (i.e., before execution) verification on program components obtained from untrusted providers. *Proof-Carrying Code* [20] is a promising approach in this area: mobile code is equipped with certain *verifiable evidence* which allows deployers to independently verify properties of interest about the code. Termination analysis is also important since the verification of functional program properties is often split into separately proving partial correctness and termination.

Object-oriented languages in general, and their low-level (bytecode) counterparts in particular, present new challenges to termination analyzers: (1) loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions; (2) *size measures* must consider primitive types, user defined objects, and arrays; and (3) *tracking* data is more difficult, as data can be stored in variables, operand stack elements or heap locations.

Analyzing JBC is a necessity in many situations, including mobile code, where the user only has access to compiled code. Furthermore, it can be argued that analyzing low-level programs can have several advantages over analyzing their high-level (Java) counterparts. One advantage is that low-level languages typically remain stable, as their high-level counterparts continue to evolve — analyzers for bytecode programs need not be enhanced each time a new language construct is introduced. Another advantage is that analyzing low-level code narrows the gap between what is verified and what is actually executed. This is relevant, for example, in safety critical applications.

In this paper we take a semantic-based approach to termination analysis, based on two steps. The first step transforms the bytecode into a *rule-based* program where all loops and all variables are represented uniformly, and which is semantically equivalent to the bytecode. This rule-based representation is based on previous work [1] in cost analysis, and is presented in Sec. 2. In the second step (Sec. 3), we adapt directly to the rule-based program standard techniques which usually prove termination of high-level languages. Sec. 4 reports on our prototype implementation and validates it by proving termination of a series of object-oriented benchmarks, containing recursion, nested loops and data structures such as trees and arrays. Conclusions and related work are presented in Sec. 5.

2 Java Bytecode and Its Rule-Based Representation

We consider a subset of the Java Virtual Machine (JVM) language which handles integers and object creation and manipulation (by accessing fields and calling methods). For simplicity, exceptions, arrays, interfaces, and primitive types besides integers are omitted. Yet, these features can be easily handled within our setting: all of them are implemented in our prototype and included in benchmarks in Table 1. A full description of the JVM [19] is out of the scope of this paper.

A sequential JBC program consists of a set of *class files*, one for each class, partially ordered with respect to the subclass relation $\preceq$. A class file contains

information about its name, the class it extends, and the fields and methods it defines. Each method has a unique signature m from which we can obtain the class, denoted $class(m)$, where the method is defined, the name of the method, and its signature. When it is clear from the context, we ignore the class and the types parts of the signature. The bytecode associated with m is a sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n \rangle$, where each b_i is a *bytecode instruction*, and pc_i is its address. The local variables of a method are denoted by $\langle l_0, \ldots, l_{n-1} \rangle$, of which the first $k \leq n$ are the formal parameters, and l_0 corresponds to the *this* reference (unlike Java, in JBC, the *this* reference is explicit). Similarly, each field f has a unique signature, from which we can obtain its name and the name of the class it belongs to. The bytecode instructions we consider include:

$$
\begin{array}{rl}
bcInst ::= & \mathsf{istore}\ v \mid \mathsf{astore}\ v \mid \mathsf{iload}\ v \mid \mathsf{aload}\ v \mid \mathsf{iconst}\ i \mid \mathsf{aconst\_null} \\
\mid & \mathsf{iadd} \mid \mathsf{isub} \mid \mathsf{iinc}\ v\ n \mid \mathsf{imul} \mid \mathsf{idiv} \\
\mid & \mathsf{if\_}\phi\ pc \mid \mathsf{goto}\ pc \mid \mathsf{ireturn} \mid \mathsf{areturn} \\
\mid & \mathsf{new}\ c \mid \mathsf{invokevirtual}\ m \mid \mathsf{invokespecial}\ m \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f
\end{array}
$$

where c is a class, ϕ is a comparison condition on numbers (ne, le, icmpgt) or references (null, nonnull), v is a local variable, i is an integer, and pc is an instruction address. Briefly, instructions are: (row 1) stack operations referring to constants and local variables; (row 2) arithmetic operations; (row 3) jumps and method return; and (row 4) object-oriented instructions. All instructions in row 3, together with invokevirtual, are *branching* (the others are *sequential*). For simplicity, we will assume all methods to return a value. Fig. 1 depicts the bytecode for the iterative method *fact*, where indexes $0, \ldots, 3$ stands for local variables *this*, n, *ft* and i respectively. $next(pc)$ is the address immediately after the program counter pc. As instructions have different sizes, addresses do not always increase by one (e.g., $next(6){=}9$).

We assume an operational semantics which is a subset of the JVM specification [19]. The execution environment of a bytecode program consists of a *heap* h and a stack A of *activation records*. Each activation record contains a program counter, a local operand stack, and local variables. The heap contains all objects (and arrays) allocated in the memory. Each method invocation generates a new activation record according to its signature. Different activation records do not share information, but may contain references to the same object in the heap.

2.1 From Bytecode to Control Flow Graphs

The JVM language is unstructured. It allows conditional and unconditional jumps as well as other implicit sources of branching, such as virtual method invocation and exception throwing. The notion of a *Control Flow Graph* (CFG for short) is a well-known instrument which facilitates reasoning about programs in unstructured languages. A CFG is similar to the older notion of a flow chart, but CFGs include a concept of "call to" and "return from". Methods in the bytecode program are represented as CFGs, and calls from one method to another correspond to calls between these graphs. In order to build CFGs, the first step

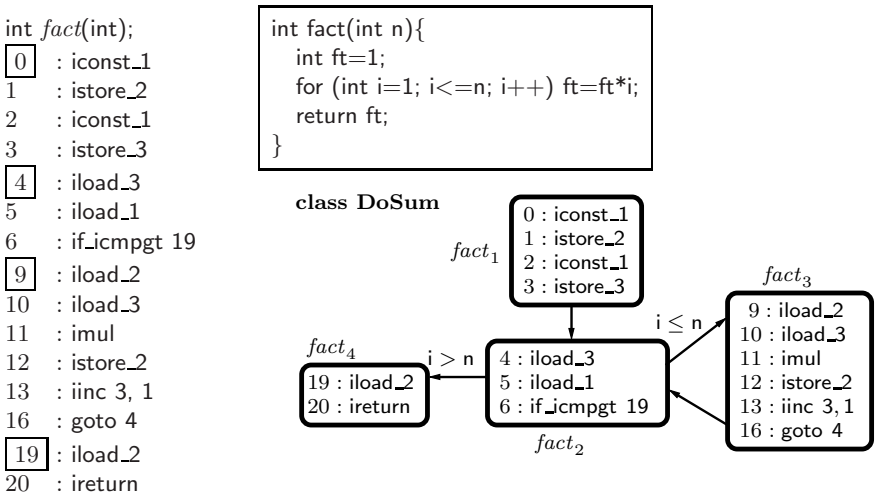


Fig. 1. A JBC method (left) with its corresponding source (center) and its CFG (right)

is to partition a sequence of bytecode instructions into a set of maximal subsequences, or basic blocks, of instructions which execute sequentially, i.e., without branching nor jumping. Given a bytecode instruction $pc{:}b$, we say that $pc'{:}b'$ is a *predecessor* of $pc{:}bc$ if one of the following conditions holds: (1) b'=goto pc, (2) b'=if_ϕ pc, (3) $next(pc')$=pc.

Definition 1 (partition to basic blocks). *Given a method m and its sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n\rangle$, a partition into basic blocks $m_1, \ldots, m_k$ takes the form*

$$\underbrace{pc_{i_1}{:}b_{i_1}, \ldots, pc_{f_1}{:}b_{f_1}}_{m_1}, \underbrace{pc_{i_2}{:}b_{i_2} \ldots, pc_{f_2}{:}b_{f_2}}_{m_2}, \ldots \underbrace{pc_{i_k}{:}b_{i_k} \ldots, pc_{f_k}{:}b_{f_k}}_{m_k}$$

where i_1=1, f_k=n and

1. *the number of basic blocks (i.e. k) is minimal;*
2. *in each basic block m_j, only the instruction b_{f_j} can be branching; and*
3. *in each basic block m_j, only the instruction b_{i_j} can have more than one predecessor.*

A partition to basic blocks can be obtained as follows: the first sequence m_1 starts at pc_1 and ends at pc_{f_1}=$min(pc_{e_1}, pc_{s_1})$, where pc_{e_1} is the address of the first branching instruction after pc_1, and pc_{s_1} is the first address after pc_1 s.t. the instruction at address $next(pc_{s_1})$ has more than one predecessor. The sequence m_2 is computed similarly starting at pc_{i_2}=$next(pc_{f_1})$, etc. Note that this partition can be computed in two passes: the first computes the predecessors, and the second defines the beginning and end of each sub-sequence.

Example 1. The JBC *fact* method on the left of Fig. 1 is partitioned into four basic blocks. The initial addresses (pc_{i_x}) of these blocks are shown within boxes. Each block is labeled by $fact_{id}$ where id is a unique block identifier. A directed edge indicates a control flow from the last instruction in the source node to the first instruction in the destination node. Edges may be labeled by a guard which states conditions under which the edge may be traversed during execution. □

In *invokevirtual*, due to dynamic dispatching, the actual method to be called may not be known at compile time. To facilitate termination analysis, we capture this information and introduce it explicitly in the CFG. This is done by adding new blocks, the *implicit basic blocks*, containing calls to *actual methods* which might be called at runtime. Moreover, access to these blocks is guarded by mutually exclusive conditions on the runtime class of the calling object.

Definition 2 (implicit basic block). *Let m be a method which contains an instruction of the form pc:b, where b=*invokevirtual *m'. Let M be a superset of the methods (signatures) that might actually be called at runtime when executing pc:b. The implicit basic block for $m''{\in}M$ is $m_{pc:c}$, where $c{=}class(m'')$ if $class(m''){\preceq}class(m')$, otherwise $c{=}class(m')$. The block includes the single special instruction* invoke(m'). *The guard of $m_{pc:c}$ is $m^g_{pc:c}{=}$*instanceof(n, c, D), *where $D{=}\{class(m'') \mid m''{\in}M,\ class(m''){\prec}c\}$, and n is the arity of m'.*

It can be seen that m is used to denote both methods and blocks in order to make them globally unique. The above condition instanceof(n, c, D) states that the $(n{+}1)$th stack element (from the top) is an instance of class c and not an instance of any class in D. Computing the set M in the above definition can be statically done by considering the class hierarchy and the method signature, which is clearly a safe approximation of the set of the actual methods that might be called at runtime when executing b. However, in some cases this might result in a much larger set than the actual one, which in turn affects the precision and performance of the corresponding static analysis. In such cases, class analysis [25] is usually applied to reduce this set as it gives information about the possible runtime classes of the object whose method is being called. Note that the instruction invoke(m') does not appear in the original bytecode, but it is instrumental to define our rule-based representation in Sec. 2.2. For example, consider the CFG in Fig. 2, which corresponds to the recursive method *doSum* and calls *fact*. This CFG contains two implicit blocks labeled $doSum_{11:DoSum}$ and $doSum_{19:DoSum}$.

The following definition formalizes the notion of a CFG for a method. Although the invokespecial bytecode instruction always corresponds to only one possible method call which can be identified from the symbolic method reference, in order to simplify the presentation, we treat it as invokevirtual, and associate it to a single implicit basic block with the *true* guard. Note that every bytecode instruction belongs to exactly one basic block. By $BlockId(pc, m){=}i$ we denote the fact that the instruction pc in m belongs to block m_i. In addition, for a given *invokevirtual* instruction pc:b in a method m, we use M^m_{pc} and G^m_{pc} to denote the set of its *implicit basic blocks* and their corresponding guards respectively.

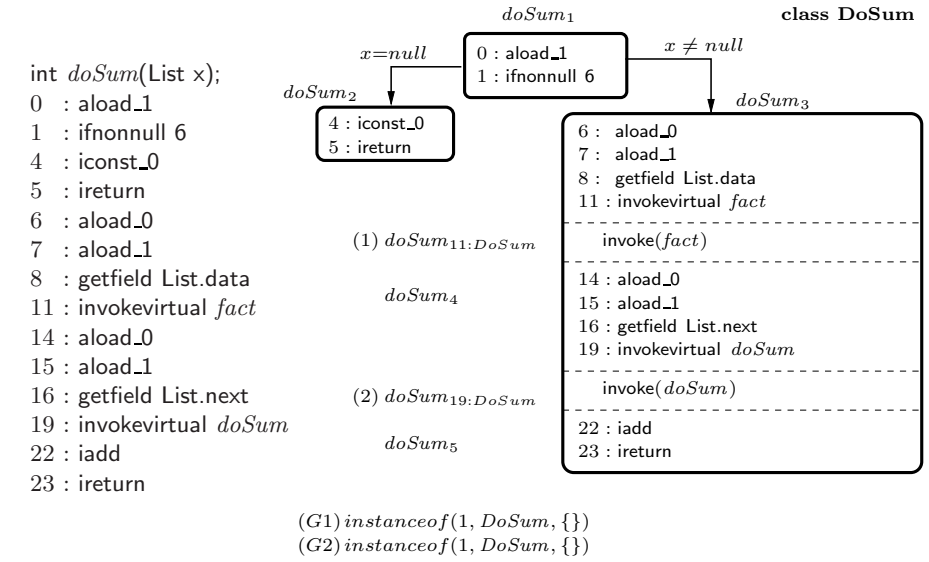


Fig. 2. The Control Flow Graph of the *doSum* example

Definition 3 (CFG). *The* control flow graph *for a method m is a graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$. Nodes $\mathcal{N}$ consist of:*

(a) basic blocks $m_1, \ldots, m_k$ of m; and
(b) implicit basic blocks corresponding to calls to methods.

Edges in $\mathcal{E}$ take the form $\langle m_i \rightarrow m_j, condition_{ij} \rangle$ where m_i and m_j are, resp., the source *and* destination *node, and $condition_{ij}$ is the Boolean condition labeling this transition. The set of edges is constructed, by considering each node $m_i \in \mathcal{N}$ which corresponds to a (non-implicit) basic block, whose last instruction is denoted as pc:b, as follows:*

1. *if b=*goto *pc′ and j=BlockId(pc′, m) then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$;*
2. *if b=*if_ϕ *pc′, j=BlockId(pc′, m) and i′=BlockId(next(pc), m) then we add both $\langle m_i \rightarrow m_j, \phi \rangle$ and $\langle m_i \rightarrow m_{i'}, \neg\phi \rangle$ to $\mathcal{E}$;*
3. *if b∈{*invokevirtual *m′,* invokespecial *m′}, and i′=BlockId(next(pc), m) then, for all $d \in M^m_{pc}$ and its corresponding $g^m_{pc:d} \in G^m_{pc}$, we add $\langle m_i \rightarrow m_{pc:d}, g^m_{pc:d} \rangle$ and $\langle m_{pc:d} \rightarrow m_{i'}, true \rangle$ to $\mathcal{E}$;*
4. *otherwise, if j=BlockId(next(pc), m) then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$.*

For conciseness, when a branching instruction b involving implicit blocks leads to a single successor block, we include the corresponding invoke instruction within the basic block b belongs to. For instance, consider that the classes *DoSum* and *List* are not extended by any other class. In this case, the branching instructions 11 and 19 have a single continuation. Their associated implicit blocks marked with (1) and (2) in Fig. 2 are, thus, just included within the basic block $doSum_3$. $G1$ and $G2$ at the bottom indicate the guards which should label the edge.

2.2 Rule-Based Representation

The CFG, while having advantages, is not optimal for our purposes. Therefore, we introduce a *Rule-Based Representation* (RBR) on which we demonstrate our approach to termination analysis. This RBR is based on a recursive representation presented in previous work [1], where it has been used for cost analysis.

The main advantages of the RBR are that: (1) all iterative constructs (loops) fit in the same setting, independently of whether they originate from recursive calls or iterative loops (conditional and unconditional jumps); and (2) all variables in the local scope of the method a block corresponds to (formal parameters, local variables, and stack values) are represented uniformly as explicit arguments. This is possible as in JBC the height of the *operand stack* at each program point is statically known. We prefer to use this rule-based representation, rather than other existing ones (e.g., BoogiePL [13] or those in Soot [26]), as in a simple post-processing phase we can eliminate almost all stack variables, which results, as we will see in Sec. 3.1, in a more efficient analysis.

A *Rule-Based Program* (RBP for short) defines a set of *procedures*, each of them defined by one or more rules. As we will see later, each block in the CFG generates one or two procedures. Each rule has the form $head(\bar{x}, \bar{y}) := guard, instr, cont$ where *head* is the name of the procedure the rule belongs to, $\bar{x}$ and $\bar{y}$ indicate sequences $\langle x_1, \ldots, x_n \rangle$, $n>0$ (resp. $\langle y_1, \ldots, y_k \rangle$, $k>0$) of input (resp. output) arguments, *guard* is of the form guard(ϕ), where ϕ is a Boolean condition on the variables in $\bar{x}$, *instr* is a sequence of (decorated) bytecode instructions, and *cont* indicates a possible call to another procedure representing the continuation of this procedure. In principle, $\bar{x}$ includes the method's local variables and the stack elements at the beginning of the block. In most cases, $\bar{y}$ only needs to store the return value of the method, which we denote by r. For simplicity, guards of the form guard($true$) are omitted. When a procedure p is defined by means of several rules, the corresponding guards must cover all cases and be pairwise exclusive.

Decorating Bytecode Instructions. In order to make all arguments explicit, each bytecode instruction in *instr* is *decorated* explicitly with the (local and stack) variables it operates on. We denote by $t{=}stack\_height(pc, m)$ the height of the stack immediately before the program point pc in a method m. Function dec in the following table shows how to *decorate* some selected instructions, where n is the number of arguments of m.

pc:b	dec(b)
iconst i	$iconst(i, s_{t+1})$
istore v	$istore(s_t, \ell_v)$
iload v	$iload(l_v, s_{t+1})$
new c	$new(c, s_{t+1})$
ireturn	$ireturn(s_t, r)$

pc:b	dec(b)
iadd	$iadd(s_{t-1}, s_t, s_{t-1})$
invoke(m)	$m(\langle s_{t-n}, \ldots, s_t\rangle, \langle s_{t-n}\rangle)$
getfield f	$getfield(f, s_t, s_t)$
putfield f	$putfield(f, s_{t-1}, s_t, s_{t-1})$
guard($icmpgt$)	guard$(icmpgt(s_{t-1}, s_t))$

Guards are translated according to the bytecode instruction they come from. Note that branching instructions do not need to appear in the RBR, since their effect is already captured by the branching at the RBR level and since invoke instructions are replaced by calls to the entry rule of the corresponding method.

Definition 4 (RBR). *Let m be a method with $l_0, \ldots, l_{n-1}$ local variables, of which $l_0, \ldots, l_{k-1}$ are the formal parameters together with the* this *reference l_0 ($k \leq n$), and let $\langle \mathcal{N}, \mathcal{E}\rangle$ be its CFG. The* rule-based representation *of $\langle \mathcal{N}, \mathcal{E}\rangle$ is* $\mathsf{rules}(\langle \mathcal{N}, \mathcal{E}\rangle) = \mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) \bigcup_{m_p \in \mathcal{N}} \mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$, *with:*

$$\mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) = \{m(\langle \ell_0, \ldots, \ell_{k-1}\rangle, \langle r\rangle) := \mathsf{init\_local\_vars}(\langle l_k, \ldots, l_{n-1}\rangle), m_1(\langle \ell_0, \ldots, \ell_{n-1}\rangle, \langle r\rangle)\}$$

where the call init_local_vars *initializes the local variables of the method, and*

$$\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle) = \begin{cases} \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) := TBC^p_m.\} & \nexists \langle m_p \mapsto \_, \_\rangle \in \mathcal{E} \\ \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) := TBC^p_m, m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle).\} \bigcup & \\ \quad \{m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle) := g, m_q(\langle \bar{l}, s_0, \ldots, s_{q_i-1}\rangle, \langle r\rangle). & \textit{otherwise} \\ \quad \mid \langle m_p \to m_q, \phi_q\rangle \in \mathcal{E} \wedge g = \mathsf{dec}(\phi_q)\} & \end{cases}$$

In the above formula, p_i (resp., p_o) denotes the height of the operand stack of m at the entry (resp., exit) of m_p. Also, q_i is the height of the stack at the entry of m_q, and TBC^p_m is the decorated bytecode for m_p. We use "_" to indicate that the value at the corresponding position is not relevant.

The function $\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$ is defined by cases. The first case is applied when m_p is a *sink* node with no out-edges. Otherwise, the second rule introduces an additional procedure m^c_p (c is for *continuation*), which is defined by as many rules as there are out-edges for m_p. These rules capture the different alternatives which execution can follow from m_p. We will unfold calls to m^c_p whenever it is deterministic (m_p has a single out-edge). This results in m_p calling m_q directly.

Example 2. The RBR of the CFG in Fig. 1 consists of the following rules where local variables have the same name as in the source code and o is the *this* object:

$$
\begin{array}{ll}
fact(\langle o, n\rangle, \langle r\rangle) & := \mathsf{init\_local\_vars}(\langle ft, i\rangle),\ fact_1(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_1(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iconst}(1, s_0),\ \mathsf{istore}(s_0, ft),\ \mathsf{iconst}(1, s_0),\\
 & \quad \mathsf{istore}(s_0, i),\ fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_2(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(i, s_0),\ \mathsf{iload}(n, s_1),\\
 & \quad fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle).\\
fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{icmpgt}(s_0, s_1)),\ fact_4(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{icmple}(s_0, s_1)),\ fact_3(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(ft, s_0),\ \mathsf{iload}(i, s_1),\ \mathsf{imul}(s_0, s_1, s_0),\\
 & \quad \mathsf{istore}(s_0, ft),\ \mathsf{iinc}(i, 1),\ fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_4(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(ft, s_0),\ \mathsf{ireturn}(s_0, r).
\end{array}
$$

The first rule corresponds to the entry. Block $fact_4$ is a sink block. Blocks $fact_1$ and $fact_3$ have a single out-edge and we have unfolded the continuation. Finally, block $fact_2$ has two out-edges and needs the procedure $fact_2^c$. The RBR from the CFG of *doSum* in Fig. 2 is ($doSum_3$ merges several blocks with one out-edge):

$$
\begin{array}{ll}
doSum(\langle o, x\rangle, \langle r\rangle) & := \mathsf{init\_local\_vars}(\langle\rangle),\ doSum_1(\langle o, x\rangle, \langle r\rangle).\\
doSum_1(\langle o, x\rangle, \langle r\rangle) & := \mathsf{aload}(x, s_0),\ doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle).\\
doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{nonnull}(s_0)),\ doSum_3(\langle o, x\rangle, \langle r\rangle).\\
doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{null}(s_0)),\ doSum_2(\langle o, x\rangle, \langle r\rangle).\\
doSum_2(\langle o, x\rangle, \langle r\rangle) & := \mathsf{iconst}(0, s_0), \mathsf{ireturn}(s_0, r).\\
doSum_3(\langle o, x\rangle, \langle r\rangle) & := \mathsf{aload}(o, s_0), \mathsf{aload}(x, s_1), \mathsf{getfield}(List.data, s_1, s_1),\\
 & \quad fact(\langle s_0, s_1\rangle, \langle s_0\rangle), \mathsf{aload}(o, s_1), \mathsf{aload}(x, s_2),\\
 & \quad \mathsf{getfield}(List.next, s_2, s_2), doSum(\langle s_1, s_2\rangle, \langle s_1\rangle),\\
 & \quad \mathsf{iadd}(s_0, s_1, s_0), \mathsf{ireturn}(s_0, r).
\end{array}
$$

We can see that a call to a different method, *fact*, occurs in $doSum_3$. This shows that our RBR allows simultaneously handling the two CFGs in our example. □

Rule-based Programs vs JBC Programs. Given a JBC program P, P_r denotes the RBP obtained from P. Note that, it is trivial to define an *interpreter* (or *abstract machine*) which can execute any P_r and obtain the same return value and termination behaviour as a JVM does for P. RBPs, in spite of their declarative appearance, are in fact imperative programs. As in the JVM, an interpreter for RBPs needs, in addition to a stack for activation records, a global heap. These activation records differ from those in the JVM in that the operand stack is no longer needed (as stack elements are explicit) and in that the scope of variables is no longer associated to methods but rather to rules. In RBPs all rules are treated uniformly, regardless of the method they originate from, so that method borders are somewhat blurred. As in the JVM, call-by-value is used for passing arguments in calls.

3 Proving Termination

This section describes how to prove termination of a JBC program given its RBR. The approach consists of two steps. In the first, we *abstract* the RBR rules by replacing all program data by their corresponding *size*, and replacing calls

corresponding to bytecode instructions by *size constraints* on the values their variables can take. This step results in a Constraint Logic Program (CLP) [16] over integers, where, for any bytecode trace t, there exists a CLP trace t' whose states are abstractions of t states. In particular, every infinite (non terminating) bytecode trace has a corresponding infinite CLP trace, so that termination of the CLP program implies termination of the bytecode program. Note that, unlike in bytecode traces which are always deterministic, the execution of a CLP program can be non-deterministic, due to the precision loss inherent to the abstraction.

In the second step, we apply techniques for proving termination of CLP programs [9], which consist of: (1) analyzing the rules for each method to infer input-output *size relations* between the method input and output variables; (2) using the input-output size relations for the methods in the program, we infer a set of abstract *direct calls-to pairs* which describe, in terms of size-change, all possible calls from one procedure to another; and (3) given this set of abstract direct calls-to pairs, we compute a set of all possible calls-to pairs (direct and indirect), describing all transitions from one procedure to another. Then we focus on the pairs which describe loops, and try to identify *ranking functions* which guarantee the termination of each loop and thus of the original program.

3.1 Abstracting the Rules

As mentioned above, rule abstraction replaces data by the *size* of data, and focuses on relations between data size. For integers, their size is just their integer value [12]. For references, we take their size to be their *path-length* [24], i.e., the length of the maximal path reachable from the reference. Then, bytecode instructions are replaced by constraints on sizes taking into account a Static Single Assignment (SSA) transformation. SSA is needed because variables in CLP programs cannot be assigned more than one value. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ will be abstracted to $s'_0{=}s_1{+}s_0$ where s'_0 refers to the value of s_0 after executing the instruction. Also, the bytecode $\mathsf{getfield}(f, s_0, s_0)$ is abstracted to $s_0{>}s'_0$ if it can be determined that s_0 (before executing the instruction) does not reference a cyclic data-structure, since the length of the longest-path reachable from s_0 is larger than the length of the longest path reachable from s'_0.

bytecode b	*abstract bytecode* b^α	ρ_{i+1}
$\mathsf{iload}(l_v, s_j)$	$s'_j{=}\rho_i(l_v)$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{iadd}(s_j, s_{j+1}, s_j)$	$s'_j{=}\rho_i(s_j){+}\rho_i(s_{j+1})$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{guard}(icmpgt(s_j, s_{j+1}))$	$\rho_i(s_j){>}\rho_i(s_{j+1})$	ρ_i
$\mathsf{getfield}(f, s_j, s_j)$	if f is of ref. type: $\rho_i(s_j){>}s'_j$ if s_j is not cyclic otherwise $\rho_i(s_j) \geq s'_j$. If f is not of ref. type: *true*	$\rho_i[s_j \mapsto s'_j]$
$\mathsf{putfield}(f, s_j, s_{j+1})$ s.t f is of ref. type	if s_j and s_{j+1} do not share, $s'_k{\leq}\rho_i(s_k){+}\rho_i(s_{j+1})$ for any s_k that shares with s_j, otherwise *true*.	$\rho_i[s_k \mapsto s'_k]$

To implement the SSA transformation we maintain a mapping ρ of variable names (as they appear in the rule) to new variable names (constraint variables). Such a mapping is referred to as a *renaming*. We let $\rho[x \mapsto y]$ denote the

modification of the renaming ρ such that it maps x to the new variable y. We denote by $\rho[\bar{x} \mapsto \bar{y}]$ the mapping of each element in $\bar{x}$ to a corresponding one in $\bar{y}$.

Definition 5 (abstract compilation). *Let* $\mathrm{R} \equiv p(\overline{x}, \overline{y}) := b_1, \ldots, b_n$ *be a rule. Let* ρ_i *be a renaming associated with the point before each* b_i *and let* ρ_1 *be the identity renaming (on the variables in the rule). The abstraction of* R *is denoted* R^α *and takes the form* $p(\overline{x}, \overline{y}') := b_1^\alpha, \ldots, b_n^\alpha$ *where* b_i^α *are computed iteratively from left to right as follows:*

1. *if* b_i *is a bytecode instruction or a guard, then* b_i^α *and* ρ_{i+1} *are obtained from a predefined lookup table similar to the one above.*
2. *if* b_i *is a call to a procedure* $q(\overline{w}, \overline{z})$*, then the abstraction* b_i^α *is* $q(\overline{w}', \overline{z}')$ *where each* $w'_k \in \overline{w}'$ *is* $\rho_i(w_k)$*, variables* $\overline{z}'$ *are fresh, and* $\rho_{i+1} = \rho_i[\overline{z} \mapsto \overline{z}', \overline{u} \mapsto \overline{u}']$ *where* $\overline{u}'$ *are also fresh variables and* $\overline{u}$ *is the set of all variables in* $\overline{w}$ *which reference data-structures that can be modified when executing* q *and those that share (i.e., might have common regions in the heap) with them.*
3. *at the end we define each* $y'_i \in \overline{y}'$ *to be the constrained variable* $\rho_{n+1}(y_i)$*.*

In addition, all reference *variables are (implicitly) assumed to be non-negative.*

Note that in point 2 above, the set of variables such that the data-structures they point to may be modified during the execution of q can be approximated by applying constancy analysis [14], which aims at detecting the method arguments that remain constant during execution, and sharing analysis [23] which aims at detecting reference variables that might have common regions on the heap. Also, the non-cyclicity condition required for the abstraction of getfield can be verified by non-cyclicity analysis [22]. In what follows, for simplicity, we assume that abstract rules are normalized to the form $p(\overline{x}, \overline{y}) := \varphi, p_1(\overline{x}_1, \overline{y}_1), \ldots, p_j(\overline{x}_j, \overline{y}_j)$ where φ is the conjunction of the (linear) size constraints introduced in the abstraction and each $p_i(\overline{x}_i, \overline{y}_i)$ is a call to a procedure (i.e., block or method).

Example 3. Recall the following rule from Ex. 2 (on the left) and its abstraction (on the right) where the renamings are indicated as comments.

$fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) :=$	$fact_3(\langle o, n, ft, i\rangle, \langle r'\rangle) :=$	$\%\ \rho_1 = id$
$\mathsf{iload}(ft, s_0),$	$s'_0 = ft,$	$\%\ \rho_2 = \rho_1[s_0 \mapsto s'_0]$
$\mathsf{iload}(i, s_1),$	$s'_1 = i,$	$\%\ \rho_3 = \rho_2[s_1 \mapsto s'_1]$
$\mathsf{imul}(s_0, s_1, s_0),$	$true,$	$\%\ \rho_4 = \rho_3[s_0 \mapsto s''_0]$
$\mathsf{istore}(s_0, ft),$	$ft' = s''_0,$	$\%\ \rho_5 = \rho_4[ft \mapsto ft']$
$\mathsf{iinc}(i, 1),$	$i' = i + 1,$	$\%\ \rho_6 = \rho_5[i \mapsto i']$
$fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).$	$fact_2(\langle o, n, ft', i'\rangle, \langle r'\rangle).$	$\%\ \rho_7 = \rho_6[r \mapsto r']$

Note that imul is abstracted to *true*, since it imposes a non-linear constraint. □

3.2 Input Output Size-Relations

We consider the abstract rules obtained in the previous step to infer an abstraction (w.r.t. size) of the input-output relation of the program blocks. Concretely,

we infer *input-output size relations* of the form $p(\overline{x},\overline{y}) \leftarrow \varphi$, where φ is a constraint describing the relation between the sizes of the input $\overline{x}$ and the output $\overline{y}$ upon exit from p. This information is needed since output of one call may be input to another call. E.g., consider the following contrived abstract rule $p(\langle x\rangle,\langle r\rangle)$:=$\{x{>}0,x{>}z\}, q(\langle z\rangle,\langle y\rangle), p(\langle y\rangle,\langle r\rangle)$. To prove termination, it is crucial to know the relation between x in the head and y in the recursive call to p. This requires knowledge about the input-output size relations for $q(\langle z\rangle,\langle y\rangle)$. Assuming this to be $q(\langle z\rangle,\langle y\rangle) \leftarrow z{>}y$, we can infer $x{>}y$. Since abstract programs are CLP programs, inferring relations can rely on standard techniques [4].

Computing an approximation of input-output size relation requires a global fixpoint. In practice, we can often take a trivial over-approximation where for all rules there is no information, namely, $p(\overline{x},\overline{y}) \leftarrow \mathit{true}$. This can prove termination of many programs, and results in a more efficient implementation. It is not enough in cases as the above abstract rule, which however, in our experience, often does not occur in imperative programs.

3.3 Call-to Pairs

Consider again the abstract rule from Ex. 3 which (ignoring the output variable) is of the form $\mathit{fact}_3(\bar{x}) := \varphi, \mathit{fact}_2(\bar{z})$. It means that whenever execution reaches a call to $\mathit{fact}_3(\bar{x})$ there will be a subsequent call to $\mathit{fact}_2(\bar{z})$ and the constraint φ holds. In general, subsequent calls may arise also from rules which are not binary. Given an abstract rule of the form $p_0 := \varphi, p_1, \ldots, p_n$, a call to p_0 may lead to a call to p_i, $1{\le}i{\le}n$. Given the input-output size relations for the individual calls $p_1,\ldots,p_{i-1}$, we can characterize the constraint for a transition between the subsequent calls p_0 and p_i by adding these relations to φ. We denote a pair of such subsequent calls by $\langle p_0(\overline{x}) \rightsquigarrow p_i(\overline{y}), \varphi_i\rangle$ and call it a *calls-to pair*.

Definition 6 (direct calls-to pairs). *Given a set of abstract rules $\mathcal{A}$ and its input-output size relations $I_{\mathcal{A}}$, the* direct calls-to pairs *induced by $\mathcal{A}$ and $I_{\mathcal{A}}$ are:*

$$C_{\mathcal{A}} = \left\{ \langle p(\overline{x}) \rightsquigarrow p_i(\overline{x}_i), \psi\rangle \;\middle|\; \begin{array}{l} p(\overline{x},\overline{y}) := \varphi, p_1(\overline{x}_1,\overline{y}_1),\ldots,p_j(\overline{x}_j,\overline{y}_j){\in}\mathcal{A}, \\ i{\in}\{1,\ldots,j\},\ \forall 0{<}k{<}i.\ p_k(\overline{x}_k,\overline{y}_k) \leftarrow \varphi_k{\in}I_{\mathcal{A}} \\ \psi = \bar{\exists}\overline{x} \cup \overline{x}_i.\varphi \wedge \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \end{array} \right\}$$

where $\bar{\exists}v$ means eliminating all variables but v from the corresponding constraint.

Example 4. Consider the rule for *doSum* in Ex. 2: note that input-output relations for *fact* and *doSum* are *true*. Direct calls-to pairs for those rules are:

$$\begin{array}{l} \langle \mathit{doSum}(o,x) \rightsquigarrow \mathit{doSum}_1(o',x'), \{x'{=}x, o'{=}o\}\rangle \\ \langle \mathit{doSum}_1(o,x) \rightsquigarrow \mathit{doSum}_1^c(o',x',s_0), \{x'{=}x, o'{=}o, s_0{=}x\}\rangle \\ \langle \mathit{doSum}_1^c(o,x,s_0) \rightsquigarrow \mathit{doSum}_3(o',x'), \{x'{=}x, o'{=}o, s_0{>}0\}\rangle \\ \langle \mathit{doSum}_1^c(o,x.s_0) \rightsquigarrow \mathit{doSum}_2(o',x'), \{x'{=}x, o'{=}o, s_0{=}0\}\rangle \\ \langle \mathit{doSum}_3(o,x) \rightsquigarrow \mathit{fact}(s_0',s_1''), \{s_0'{=}o\}\rangle \\ \langle \mathit{doSum}_3(o,x) \rightsquigarrow \mathit{doSum}(s_1''',s_2''), \{s_1'''{=}o, x{>}s_2''\}\rangle \end{array}$$

In the last rule, s_2'' corresponds to $x.\mathit{next}$, so that we have the constraint $x{>}s_2''$. It can be seen that since the list is not cyclic and does not share with other

variables, size analysis finds the above decreasing of its size $x{>}s_2''$. Note also that all variables corresponding to references are assumed to be non-negative. Similarly, we can obtain direct calls-to pairs for the rule of *fact*. □

It should be clear that the set of direct calls-to pairs relations $C_{\mathcal{A}}$ is also a binary CLP program that we can execute from a given goal. A key feature of this binary program is that if an infinite trace can be generated using the abstract program described in Sec. 3.1, then an infinite trace can be generated using this binary CLP program [10]. Therefore, absence of such infinite traces (i.e., termination) in the binary program $C_{\mathcal{A}}$ implies absence of infinite traces in the abstract bytecode program, as well as in the original bytecode program.

Theorem 1 (Soundness). *Let P be a JBC program and $C_{\mathcal{A}}$ the set of direct calls-to pairs computed from P. If there exists a non-terminating trace in P then there exists a non-terminating derivation in $C_{\mathcal{A}}$.*

Intuitively, the result follows from the following points. By construction, the RBP captures all possibly non-terminating traces in the original program. By the correctness of size analysis, we have that, given a trace in the RBP, there exists an equivalent one in $C_{\mathcal{A}}$, among possibly others. Therefore, termination in $C_{\mathcal{A}}$ entails termination in the JBC program.

3.4 Proving Termination of the Binary Program $C_{\mathcal{A}}$

Several automatic termination tools and methods for proving termination of such binary constraint programs exists [9,10,17]. They are based on the idea of first computing all possible calls-to pair from the direct ones, and then finding a ranking function for each recursive calls-to pairs, which is sufficient for proving termination. Computing all possible calls-to pairs, usually called the *transitive closure* $C_{\mathcal{A}}^*$, can be done by starting from the set of direct calls-to pairs $C_{\mathcal{A}}$, and iteratively adding to it all possible compositions of its elements until a fixed-point is reached. Composing two calls-to pairs $\langle p(\overline{x}) \rightsquigarrow q(\overline{y}), \varphi_1\rangle$ and $\langle q(\overline{w}) \rightsquigarrow r(\overline{z}), \varphi_2\rangle$ returns the new calls-to pair $\langle p(\overline{x}) \rightsquigarrow r(\overline{z}), \bar{\exists}\bar{x} \cup \bar{z}.\varphi_1 \wedge \varphi_2 \wedge (\bar{y} = \bar{w})\rangle$.

Example 5. Applying the transitive closure on the direct calls-to pairs of Ex. 4, we obtain, among many others, the following calls-to pairs. Note that x (resp. i) strictly decreases (resp. increases) at each iteration of its corresponding loop:

$$\langle doSum(o,x) \rightsquigarrow doSum(o',x'), \{o'{=}o, x{>}x', x{\geq}0\}\rangle$$
$$\langle fact_2(o,n,ft,i) \rightsquigarrow fact_2(o',n',ft',i'), \{o'{=}o, n'{=}n, i'{>}i, i{\geq}1, n{\geq}i'{-}1\}\rangle \quad \square$$

As already mentioned, in order to prove termination, we focus on *loops* in $C_{\mathcal{A}}^*$. Loops are the *recursive* entities of the form $\langle p(\overline{x}) \rightsquigarrow p(\overline{y}), \varphi\rangle$ which indicate that a call to a program point p with values $\bar{x}$ eventually leads to a call to the same program point with values $\bar{y}$ and that φ holds between $\overline{x}$ and $\overline{y}$. For each loop, we seek a *ranking function* F over a well-founded domain such that $\varphi \models \mathrm{F}(\overline{x}){>}\mathrm{F}(\overline{y})$. As shown in [9,10], finding a ranking function for every recursive calls-to pair implies termination. Computing such functions can be done, for instance, as described in [21]. As an example, for the loops in Ex. 5 we get the following ranking functions: $\mathrm{F}_1(o,x){=}x$ and $\mathrm{F}_2(o,n,ft,i){=}n{-}i{+}1$.

3.5 Improving Termination Analysis by Extracting Nested Loops

In proving termination of JBC programs, one important question is whether we can prove termination at the JBC level for a class of programs which is comparable to the class of Java *source* programs for which termination can be proved using similar technology. As can be seen in Sec. 4, directly obtaining the RBR of a bytecode program is non-optimal, in the sense that proving termination on it may be more complicated than on the source program. This happens because, while in source code it is easy to reason about a nested loop independently of the outer loop, loops are not directly visible when control flow is unstructured. Loop extraction is useful for our purposes since nested loops can be dealt with one at a time. As a result, finding a ranking function is easier, and computing the closure can be done locally in the strongly connected components. This can be crucial in proving the termination of programs with nested loops.

To improve the accuracy of our analysis, we include a component which can detect and extract loops from CFGs. Due to space limitations, we do not describe how to perform this step here (more details in work about decompilation [2], where loop extraction has received considerable attention). Very briefly, when a loop is extracted, a new CFG is created. As a result, a method can be converted into several CFGs. These ideas fit very nicely within our RBR, since calls to loops are handled much in the same way as calls to other methods.

4 Experimental Results

Our prototype implementation is based on the size analysis component of [1] and extends it with the additional components needed to prove termination. The analyzer can also output the set of direct call-pairs, which allows using existing termination analyzers based on similar ideas [10,17]. The system is implemented in Ciao Prolog, and uses the Parma Polyhedra Library (PPL) [3].

Table 1 shows the execution times of the different steps involved in proving the termination of JBC programs, computed as the arithmetic mean of five runs. Experiments have been performed on an Intel 1.86 GHz Pentium M with 1 GB of RAM, running Linux on a 2.6.17 kernel. The table shows a range of benchmarks for which our system can prove termination, and which are meant to illustrate different features. We show classical recursive programs such as *Hanoi*, *Fibonacci*, *MergeList* and *Power*. Iterative programs *DivByTwo* and *Concat* contain a single loop, while *Sum*, *MatMult* and *BubbleSort* are implemented with nested loops. We also include programs written in object-oriented style, like *Polynomial*, *Incr*, *Scoreboard*, and *Delete*. The remaining benchmarks use data structures: arrays (*ArrayReverse*, *MatMultVector*, and *Search*); linked lists (*Delete* and *ListReverse*); and binary trees (*BST*).

Columns **CFG**, **RBR**, **Size**, **TC**, **RF**, **Total**$_1$ contain the running times (in ms) required for the CFG (including loop extraction), the RBR, the size analysis (including input-output relations), the transitive closure, the ranking functions and the total time, respectively. Times are high, as the implementation has been developed to check if our approach is feasible, but is still preliminary. The most

Table 1. Measured time (in ms) of the different phases of proving termination

Benchmark	**CFG**	**RBR**	**Size**	**TC**	**RF**	**Total$_1$**	**Termin**	**Total$_2$**	**Ratio**
Polynomial	138	12	260	1453	26	1890	yes	2111	1.12
DivByTwo	52	4	168	234	4	462	yes	538	1.17
EvenDigits	59	7	383	1565	17	2030	yes	2210	1.09
Factorial	43	3	46	268	3	363	yes	353	0.97
ArrayReverse	58	5	208	339	24	635	yes	834	1.32
Concat	65	8	660	943	38	1715	yes	3815	2.23
Incr	35	12	854	4723	28	5652	yes	6590	1.17
ListReverse	21	5	141	310	5	481	yes	515	1.07
MergeList	107	23	130	5184	21	5464	yes	5505	1.01
Power	14	3	72	357	9	454	yes	459	1.01
Cons	25	7	65	1318	10	1424	yes	1494	1.05
ListInter	136	22	585	9769	49	10560	yes	27968	2.65
SelectOrd	154	16	1298	4076	48	5592	no	25721	4.60
DoSum	57	10	64	923	6	1060	yes	1069	1.01
Delete	121	14	54	2418	1	2608	yes	33662	12.91
MatMult	240	11	2411	4646	294	7602	no	32212	4.24
MatMultVector	254	15	2563	8744	242	11817	no	34688	2.94
Hanoi	39	5	172	979	3	1198	no	1198	1.00
Fibonacci	23	3	90	290	5	411	yes	401	0.98
BST	68	12	97	4643	18	4838	yes	4901	1.01
BubbleSort	152	12	1125	4366	83	5738	no	14526	2.53
Search	65	11	307	756	11	1150	yes	1430	1.24
Sum	64	7	480	1758	35	2343	no	5610	2.39
FactSumList	65	12	80	961	5	1123	yes	1306	1.16
Scoreboard	268	23	1597	4393	81	6362	no	32999	5.19

expensive steps are the size analysis and the transitive closure, since they require global analysis. Last three columns show the benefits of loop extraction. **Termin** tells if termination can be proven (using polyhedra) without extraction. In seven cases, termination is only proven if loop extraction is performed. **Total$_2$** shows the total time required to check termination without loop extraction. **Ratio** compares **Total$_2$** with **Total$_1$** (**Total$_2$**/**Total$_1$**), showing that, in addition to improving precision, loop extraction is beneficial for efficiency, since **Ratio** ≥ 1 in most cases, and can be as high as 12.91 in *Delete*. Note that termination of these programs may be proved without loop extraction by using other domains such as *monotonicity constraints* [7]. However, we argue that loop extraction is beneficial as it facilitates reasoning on the loops separately. Also, if it fails to prove termination, it reports the possibly non-terminating loops.

5 Conclusions and Related Work

We have presented a termination analysis for (sequential) JBC which is, to the best of our knowledge, the first approach in this direction. This analysis

successfully deals with the challenges related to the low-level nature of JBC, and adapts standard techniques used, in other settings, in decompilation and termination analysis. Also, we believe that many of the ideas presented in this paper are also applicable to termination analysis of low-level languages in general, and not only JBC. We have used the notion of path-length to measure the size of data structures on the heap. However, our approach is parametric on the abstract domain used to measure the size. As future work, we plan to implement non-cyclicity analysis [22], constancy analysis [14], and sharing analysis [23], and to enrich the transitive closure components with monotonicity constraints [7]. Unlike polyhedra, monotonicity constraints can handle disjunctive information which is often crucial for proving termination. In [5], a termination analysis for C programs, based on binary relations similar to ours, is proposed. It uses separation logic to approximate the heap structure, which in turn allows handling termination of programs manipulating cyclic data structures. We believe that, for programs whose termination does not depend on cyclic data-structures, both approaches deal with the same class of programs. However, ours might be more efficient, as it is based on a simpler abstract domains (a detailed comparison is planned for future work). Recently, a novel termination approach has been suggested [8]. It is based on cyclic proofs and separation logic, and can even handle complicated examples as the reversal of panhandle data-structures. It is not clear to us how practical this approach is.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. Samir Genaim was supported by a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. Part of this work was performed during a research stay of Michael Codish at UPM supported by a grant from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
2. Allen, F.: Control flow analysis. In: Symp. on Compiler optimization (1970)
3. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
4. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)

5. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
6. Bradley, A., Manna, Z., Sipma, H.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
7. Brodsky, A., Sagiv, Y.: Inference of Inequality Constraints in Logic Programs. In: Proceedings of PODS 1991, pp. 95–112. ACM Press, New York (1991)
8. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proceedings of POPL-35 (January 2008)
9. Bruynooghe, M., Codish, M., Gallagher, J., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM TOPLAS 29(2) (2007)
10. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. J. Log. Program. 41(1), 103–123 (1999)
11. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. POPL. ACM Press, New York (1978)
13. DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft (2005)
14. Genaim, S., Spoto, F.: Technical report, Personal Communication (2007)
15. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
16. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. Journal of Logic Programming 19(20), 503–581 (1994)
17. Lee, C., Jones, N., Ben-Amram, A.: The size-change principle for program termination. In: Proc. POPL. ACM Press, New York (2001)
18. Lindenstrauss, N., Sagiv, Y.: Automatic termination analysis of logic programs. In: ICLP (1997)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. A-W (1996)
20. Necula, G.: Proof-Carrying Code. In: POPL 1997. ACM Press, New York (1997)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937. Springer, Heidelberg (2004)
22. Rossignoli, S., Spoto, F.: Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855. Springer, Heidelberg (2005)
23. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
24. Spoto, F., Hill, P.M., Payet, E.: Path-length analysis for object-oriented programs. In: Proc. EAAI (2006)
25. Spoto, F., Jensen, T.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Syst. 25(5), 578–630 (2003)
26. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proc. of CASCON 1999, pp. 125–135 (1999)

PROLE 2009

Termination and Cost Analysis with COSTA and its User Interfaces

E. Albert[1] P. Arenas[1] S. Genaim[1] M. Gómez-Zamalloa[1]
G. Puebla[2] D. Ramírez[2] G. Román[2] D. Zanardini[2]

[1] *DSIC, Complutense University of Madrid,* {elvira,puri,samir.genaim,mzamalloa}@fdi.ucm.es

[2] *Technical University of Madrid,* {german,diana,groman,damiano}@clip.dia.fi.upm.es

Abstract

COSTA is a static analyzer for Java bytecode which is able to infer cost and termination information for large classes of programs. The analyzer takes as input a program and a resource of interest, in the form of a cost model, and aims at obtaining an upper bound on the execution cost with respect to the resource and at proving program termination. The COSTA system has reached a considerable degree of maturity in that (1) it includes state-of-the-art techniques for statically estimating the resource consumption and the termination behavior of programs, plus a number of specialized techniques which are required for achieving accurate results in the context of object-oriented programs, such as handling numeric fields in value analysis; (2) it provides several non-trivial notions of cost (resource consumption) including, in addition to the number of execution steps, the amount of memory allocated in the heap or the number of calls to some user-specified method; (3) it provides several user interfaces: a classical command line, a Web interface which allows experimenting remotely with the system without the need of installing it locally, and a recently developed Eclipse plugin which facilitates the usage of the analyzer, even during the development phase; (4) it can deal with both the Standard and Micro editions of Java. In the tool demonstration, we will show that COSTA is able to produce meaningful results for non-trivial programs, possibly using Java libraries. Such results can then be used in many applications, including program development, resource usage certification, program optimization, etc.

Keywords: Cost Analysis, Termination Analysis, Resource Usage.

1 Introduction and System Description

We start by describing the architecture of COSTA, an abstract-interpretation-based static analyzer for studying the *cost* [4] and *termination* [1] behavior of Java bytecode [7] programs. *Cost analysis* deals with statically estimating the amount of *resources* which can be consumed at runtime (i.e., the cost), given the notion of a specific resource of interest, while the goal of *termination analysis* is to prove, when it is the case, that a program terminates for every input.

⋆ This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The input provided to the analyzer consists of a program and a description of the resource of interest, which we refer to as *cost model*. COSTA tries to infer an *upper bound* of the resource consumption, and *sound information* on the termination behavior (i.e., if the system infers that the program terminates then it should definitely terminate). The system comes equipped with several notions of cost, such as the *heap consumption*, the *number of bytecode instructions* executed, and the *number of calls* to a specific method.

COSTA is based on the classical approach to static cost analysis [14] which consists of two phases. First, given a program and a description of the resource, the analysis produces *cost relations*, which are sets of recursive equations. Second, closed-form solutions are found, if possible. For this, COSTA uses PUBS [2].

Having both cost and termination analysis in the same tool is interesting since such analyses share most of the computing machinery, and thus a large part of the analyzer is common to both. As an example, proving termination needs reasoning about the number of iterations of every loop in the program, which is also an essential piece of information for computing its cost.

In spite of being still a prototype, COSTA includes state-of-the-art techniques for cost and termination analysis, plus a number of specialized components and auxiliary static analyses which are required in order to achieve accurate results in the context of *object-oriented* programs, such as handling *numeric fields* in value analysis. As for the usability, the system provides several *user interfaces*: (i) a classical *command-line interface* (Section 2.1); (ii) a *Web interface* which allows using COSTA from a remote location, without the need of installing it locally (Section 2.2), and permits to upload user-defined examples as well as testing programs from a representative set; and (iii) a recently developed *plugin* for the widely used programming environment Eclipse [6], which allows easily using the analyzer while developing software (Section 2.3). COSTA can deal with full sequential Java, either in the *Standard Edition* [13] or the *Micro Edition* [8]. Needless to say, the analyzer works on Java bytecode programs, and does not require them to come from the compilation of Java source code: instead, bytecode may have been implemented by hand, or obtained by compiling languages different from Java.

The *tool demonstration* will show that COSTA is able to read `.class` files and produce meaningful and reasonably precise results for non-trivial programs, possibly using Java *libraries*. Possible uses of such cost and termination results include:

- helping the programmer in the *development* process, as obtained by using COSTA from the Eclipse plugin;
- the COSTA results can be used as guarantees that the program will not take too much time or resources in its execution nor fail to terminate; furthermore, this can potentially be combined with the *Proof-carrying code* [10] paradigm by adding certificates to programs which make checking resource usage more efficient.
- program *optimization*, COSTA can be used for guiding program optimization or choosing the most efficient implementation among several alternatives.

The preliminary experimental results performed to date are very promising and they suggest that resource usage and termination analysis can be applied to a realistic object-oriented, bytecode programming language.

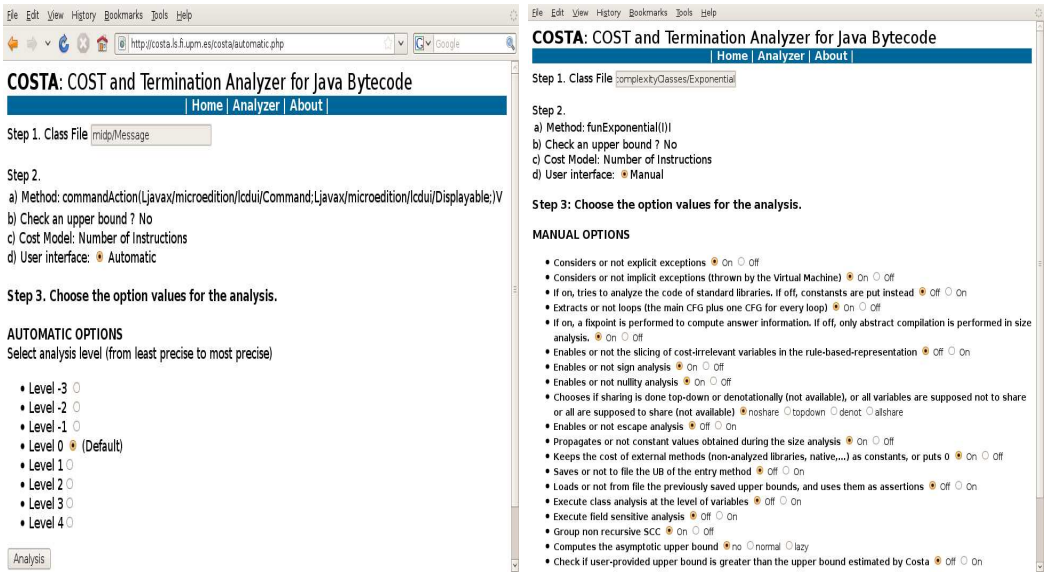


Fig. 1. Two ways of setting values for analysis options

2 User Interfaces of COSTA

2.1 Command-Line Interface

COSTA has a command-line interface for executing COSTA as a standalone application. Different switches allow controlling the different options of the analyzer. It facilitates the implementation of other interfaces, as discussed below. They collect user information and interact with COSTA by making calls to its command-line interface.

2.2 Web Interface

The COSTA web interface allows users to try out the system on a set of representative examples, and also to upload their own programs, which can be in the form of either Java source, or as Java bytecode, in which case it can be given as a `.class` or a `.jar` file. As the behavior of COSTA can be customized using a relatively large set of options, the web interface allows two alternatives modes of use.

The first alternative, which we call *automatic* (see Figure 1, left) allows the user to choose from a range of possibilities which differ in the analysis accuracy and overhead. Starting from level 0, the default, we can increase the analysis accuracy (and overhead) by using levels 1 through 3. We can also reduce analysis overhead (and accuracy) by going down to levels -1 through -3. The main advantage of the automatic mode is that it does not require the user to understand the different options implemented in the system and their implications in analysis accuracy and overhead. The second alternative is called *manual* (see Figure 1, right) and it is meant for expert users. There, the user has access to all the analysis options, allowing a fine-grained control over the behavior of the analyzer. For instance, these options allow deciding whether to analyze the Java standard libraries or not, whether to take exceptions into account, to perform or not a number of pre-analyses, to write/read analysis results to file in order to reuse them in later analyses, etc.

Figure 2 shows the output of COSTA on an example program with exponential

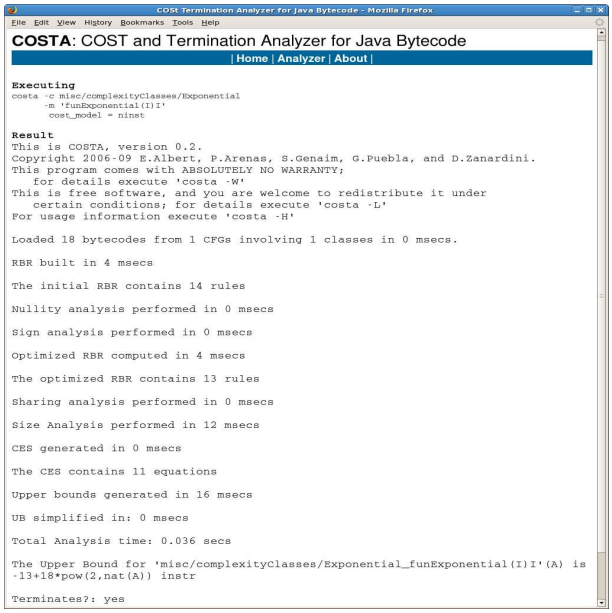


Fig. 2. Results

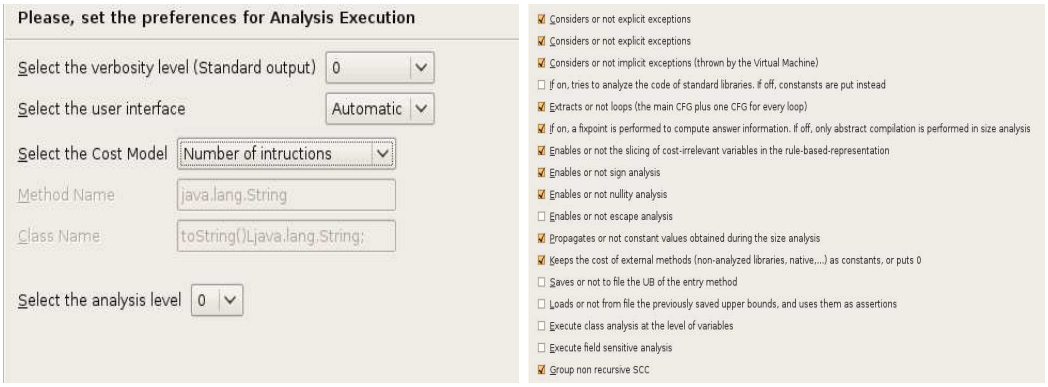


Fig. 3. COSTA Plugin Preferences

complexity. In addition to showing the result of termination analysis and an upper bound on the execution cost, COSTA (optionally) displays information about the time required by the intermediate steps performed by the analyzer in previous phases.

2.3 Eclipse Plugin

COSTA also has available an Eclipse plugin interface, which is fully integrated within the Eclipse development environment. This plugin allows programmers to analyze methods during the development process. It loads the `classpath` established for the project and uses for analysis the same classes and libraries specified by the user to compile and execute the program. As in the web interface, users can configure a large set of options by using the Eclipse preferences configuration window, as shown in Fig. 3. These options are saved and loaded at every Eclipse execution. Also, the user can choose either the automatic analysis or the expert mode which allows a more fine-grained customization, like in the web interface. By using this plugin,

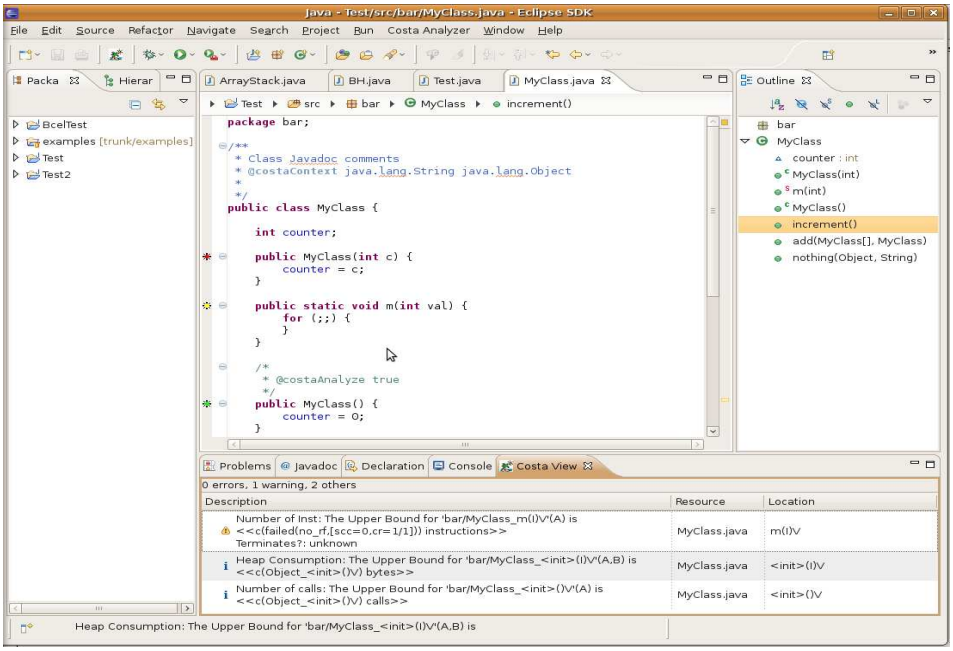


Fig. 4. COSTA Plugin Markers and View

one can analyze one or several methods from a class (see Fig. 5) or the whole class (by running the analysis on all its methods). The results of the analysis are shown using markers in the source code (see Fig. 4). Such markers are different depending on the cost model used for analysis. In addition, the plugin also shows all previous analysis results in an additional view, which we call "the COSTA view". The COSTA view also includes a warning icon for methods whose termination is not proved, in order to alert the programmer about potential problems. It can also read comments in the source code, written in Javadoc style, in order to set up analysis information.

Fig. 5. COSTA Plugin Methods Selection

3 Functionalities of COSTA

In this section, we explain the main functionalities of COSTA by means of several small examples. Some of these examples aim at illustrating the different cost models available in the system. The last two examples are related to termination issues. In particular, we start in Sect. 3.1 by showing a program whose execution requires

```
public static int funExp(int n) {
   if (n < 1) return 1;
   else return funExp(n - 1) + funExp(n - 2);
}
```

Fig. 6. Example for number of instructions

```
abstract class List {
  abstract List copy();
}

class Nil extends List {
  List copy() {
    return new Nil();
  }
}

class Cons extends List {
  int elem;
  List next;

  List copy(){
    Cons aux = new Cons();
    aux.elem = m(this.elem);
    aux.next = this.next.copy();
    return aux;
  }

  static int m(int n) {
    Integer aux = new Integer(n);
    return aux.intValue();
  }
}  // class Cons
```

Fig. 7. Example for memory consumption

an exponential number of bytecode instructions. Then, in Sect. 3.2, we present the cost model that bounds the total heap consumption of executing a program and the recent extension to account for the effect of garbage collection. Sect. 3.3 performs resource analysis on a MIDlet using the cost model "number of calls" to a given method. Finally, in Sect. 3.4, we prove termination on an example whose resource consumption cannot be bound by COSTA and, also, show the latest progress to handle numeric fields(Sect. 3.5) in termination analysis.

3.1 Number of Instructions

The cost model which counts the number of instructions which are executed is probably the most widely used within cost analyzers, as it is a first step towards estimating the runtime required to run a program. Let us consider the Java method in Fig. 6. The execution of this method has an exponential complexity as each call spawns two recursive calls until the base case is found. COSTA yields the upper bound(slightly pretty printed) `-13 + 18*2`$^{\texttt{nat(n)}}$ using its automatic mode which indicates, as expected, that the number of instructions which are executed grows exponentially with the value of the input argument `n`. This shows that COSTA is not restricted to polynomial complexities, in contrast to many other approaches to cost analysis.

3.2 Memory Consumption

Let us consider the Java program depicted in Figure 7. It consists of a set of Java classes which define a linked-list data structure in an object-oriented style. The class `Cons` is used for data nodes (in this case integer numbers) and the class `Nil` plays the role of *null* to indicate the end of a list. Both `Cons` and `Nil` extend the abstract class `List`. Thus, a `List` object can be either a `Cons` or a `Nil` instance. Both subclasses implement a `copy` method which is used to clone the corresponding

object. In the case of `Nil`, `copy` just returns a new instance of itself since it is the last element of the list. In the case of `Cons`, it returns a cloned instance where the data is cloned by calling the static method `m`, and the continuation is cloned by calling recursively the `copy` method on `next`.

The *heap* cost model of COSTA basically assigns, to each memory allocation instruction, the number of heap units it consumes. It can therefore be used to infer the total amount of memory allocated by the program. Running COSTA in automatic mode, level 0, yields the following upper bound for the `copy` method of class `Cons`:

$$\texttt{nat(this-1)*(12 + k}_1\texttt{ + k}_2\texttt{ + k}_3\texttt{) + 12 + 2*k}_1\texttt{ + k}_2\texttt{ + k}_3$$

It can be observed that the heap consumption is linear w.r.t. the input parameter `this`, which corresponds to the size of the *this* object of the method, i.e., the length of the list which is being cloned. This is because the abstraction being used by COSTA for object references is the *length of the longest reference chain*, which in this case corresponds to the length of the list. The expression also includes some constants. The symbolic constants $\texttt{k}_1$, $\texttt{k}_2$ and $\texttt{k}_3$ represent the memory consumption of the library methods which are transitively invoked. In particular, $\texttt{k}_1$ corresponds to the constructor of class `Object` and $\texttt{k}_2$ resp. $\texttt{k}_3$ to the constructor and `intValue` method of the class `Integer`. The numeric constant 12 is obtained by adding 8 and 4, being 8 the bytes taken by an instance of class `Cons`, and 4 the bytes taken by an `Integer` instance. Note that we are approximating the size of an object by the sum of the sizes of all of its fields. In particular, both an integer and a reference are assumed to consume 4 bytes.

Interestingly, we can activate the flag *go_into_java_api* and thus ask COSTA to analyze all library methods which are transitively invoked. In this case we obtain the upper bound `12*nat(this-1) + 12`, for the same method. This is because the library methods used do not allocate new objects on the heap.

3.2.1 Peak Heap Consumption

In the case of languages with automatic memory management (*garbage collection*) such as Java Bytecode, measuring the total amount of memory allocated, as done above, is not very accurate, since the actual memory usage is often much lower. *Peak heap consumption analysis* aims at approximating the size of the *live* data on the heap during a program's execution, which provides a much tighter estimation. We have recently developed and integrated in COSTA a peak memory consumption analysis [5]. Among other things, this has required the integration of an *escape analysis* which approximates the objects which do not *escape*, i.e., which are not reachable after a method's execution. The upper bound `ub(A) = 8*nat(A-1) + 24` is now obtained for the same example.

An interesting observation is that the *Integer* object which is created inside the *m* method is not reachable from outside and thus can be garbage collected. The peak heap analyzer accounts for this and therefore deletes the size of the *Integer* object from the recursive equation, thus obtaining 8 instead of 12 multiplying $\texttt{nat(A} - \texttt{1)}$. By looking at the upper bound above, it can be observed that COSTA is not being fully precise, as the actual peak consumption of this method is $\texttt{8} * \texttt{nat(A} - \texttt{1)} + \texttt{8}$

```
public void commandAction(Command c, Displayable s) {
 if (c == exitCommand) {
  destroyApp(false);
  notifyDestroyed();
 }
 if (c == sendMsgCommand) {
  try {
   TextMessage tmsg=(TextMessage)clientConn.newMessage(MessageConnection.TEXT_MESSAGE);
   tmsg.setAddress("sms://+34697396559");
   tmsg.setPayloadText(msgToSend);
   clientConn.send(tmsg);
  }
  catch (Exception exc) {
   exc.printStackTrace();
  }
 }
}
```

Fig. 8. Example for number of calls

(i.e. the size of the cloned list). The reason for this is that the upper bound solver has to consider an additional case introduced by the peak heap analysis to ensure soundness, hence making the second constant increase to 24.

3.3 Number of Calls – Java Micro Edition

The Java Micro Edition (*Java ME*) [8] technology provides a limited environment to create Java applications which can be run on small devices with limited memory, display and power capacity. It is based on three elements: a *configuration* that provides the most basic set of libraries and virtual machine capabilities, a *profile* which is a set of APIs supported by mobile devices and an *optional package* (set of technology-specific APIs). MIDP (Mobile Information Device Profile) [12] is the profile that limits the set of APIs to only those functional areas considered as absolute requirements to achieve broad portability and successful deployments. A MIDlet is an application meeting the specifications for the Java ME technology, such as a game or a business application. Each MIDlet is an object of class MIDlet which follows a lifecycle [9], which is a state automaton managed by the Application Management System (*AMS*).

COSTA is able to perfom resource analysis on MIDlets by considering all classes used on each method called during the lifecycle of the MIDlet. Such methods are the constructor of the class, the `startApp()` and the `commandAction(Command c, Displayable d)` methods. In particular, the classes used during the analysis of the class constructor are added to the analysis of the `startApp()` method. After analyzing `startApp()` method, the current classes are used for analyzing the `commandAction(Command c, Displayable d)` method. As a result, the analyzer obtains a more precise cost and resource analysis for MIDP applications. Fig. 8 shows a simple but real example MIDlet that sends a text message: the text message is created (`newMessage` method), the recipient phone number set (`setAddress` method) and the text message is sent using the method `send(Message tmsg)` of the Wireless Messaging API.

We analyze this example using the cost model that counts the number of calls

```
static int factorial(int n) {
 int fact=1;
 for (int i=1; i<=n; i++) fact=fact*i;
 return fact;
};
```

```
static int doSum(List x) {
 if (x==null) return 0;
 else return factorial(x.elem)*doSum(x.next);
}
```

Fig. 9. Example for termination

(*ncalls*) to a particular method. We apply it to obtain an upper bound on how many times the `send(Message tmsg)` method is called during the execution of `commandAction` method in a mobile device. COSTA outputs `1` as result, as it is to be expected.

3.4 Termination

Fig. 9 shows two methods which belong to the same class. The method `doSum` computes the sum of all factorial numbers contained in the elements of a linked list `x`, where `List` is defined as in Fig. 7. COSTA is able to ensure the termination of method `doSum` but no upper bound can be found by the system for the cost model *ninst*. The information that COSTA yields when computing an upper bound is:

```
The Upper Bound for 'doSum'(x) is nat(x)*(19+c(maximize_failed)*9)+4
Terminates?: yes
```

Intuitively, the cost of the calls to `factorial` cannot be bound because the value of `x.elem` is unknown at analysis time. However, we can still prove that the execution of the two methods always terminates by finding a so-called ranking function [11]. The technical details about how COSTA deals with termination can be found in [1].

3.5 Numeric Fields

Fig. 10 shows a Java program involving a numeric field in the condition of the loop of method `m`. This loop terminates in sequential execution because the field `size` is decreased at each iteration, at instruction `x.f.setSize(x.f.getSize()` $-$ `1)`, and, for any initial value of `size`, there are only a finite number of values which `size` can take before reaching zero. Unfortunately, applying standard value analyses on numeric fields can produce wrong results because numeric variables are stored in a shared mutable data structure, i.e., the heap. This implies that they can be modified using different references which are aliases and point to such memory location. Hence, further conditions are required to safely infer termination. COSTA incorporates a novel approach for approximating the value of heap allocated numeric variables [3] which greatly improves the precision over existing field-insensitive value analyses while introducing a reasonable overhead. For the example in Fig. 10, COSTA not only guarantees termination of method `m` but is also able to compute the (pretty printed) upper bound for `m(this,x,y,size) is 33+nat(size)*35` by using the cost model *ninst*.

4 Discussion and Future Work

COSTA is, to the best of our knowledge, the first tool for fully automatic cost analysis of object-oriented programs. Currently, the system can be tried online through

```
class B {
 private int size;
 public int getSize(){return size;};
 public void setSize(int n){size=n;};
};
```

```
class A {
 private B f;

 int m(A x,B y) {
  int i=0;
  while (x.f.getSize()>0) {
   i=i+y.getSize();
   x.f.setSize(x.f.getSize()-1);
  }
  return i;
 }
};
```

Fig. 10. Example for termination in presence of numeric fields

the COSTA web site: http://costa.ls.fi.upm.es. We plan to distribute it soon under a GPL license. The fact that COSTA analyzes bytecode, i.e., compiled code, makes it more widely applicable, since it is customary in Java applications to distribute compiled programs, often bundled in jars, for which the Java source is not available.

As future work we plan to: (1) define new cost models to measure the consumption of new resources; (2) support other complexity schemes such as the inference of lower-bounds; (3) improve both the precision and performance of the underlying static analyses; and (4) handle the analysis of concurrent programs.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18, 2008.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*, LNCS 5079, 2008.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Dealing with numeric fields in termination analysis of java-like languages. In *FTfJP*, 2008.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[5] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

[6] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. A-W, 1996.

[8] Java ME. http://java.sun.com/javame/technology/index.jsp.

[9] MIDP. http://java.sun.com/javame/reference/apis/jsr118/javax/-microedition/midlet/package-summary.html.

[10] G. Necula. Proof-Carrying Code. In *Proc. of ACM Symposium on Principles of programming languages (POPL)*, pages 106–119. ACM Press, 1997.

[11] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[12] Java Community Process MIDP Release. http://jcp.org/aboutJava/communityprocess/final/jsr118-/index.html.

[13] Java SE. http://java.sun.com/javase/technologies/index.jsp.

[14] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

Noname manuscript No.
(will be inserted by the editor)

Closed-Form Upper-Bounds in Static Cost Analysis

Elvira Albert · Puri Arenas
Samir Genaim · Germán Puebla

Received: November 11, 2008 / Accepted: ???

Abstract The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, the analysis first produces *cost relations* (*CRs*), i.e., recursive equations which capture the cost of the program in terms of the size of its input data. Second, *CRs* are converted into *closed-form* without recurrences. Whereas the first phase has received considerable attention, with a number of cost analyses available for a variety of programming languages, the second phase has been comparatively less studied. This article presents, to our knowledge, the first practical framework for the generation of closed-form upper-bounds for *CRs* which (1) is fully automatic, (2) can handle the distinctive features of *CRs* originated from cost analysis of realistic programming languages, (3) is not restricted to simple complexity classes, and (4) produces reasonably accurate solutions. A key idea in our approach is to view *CRs* as programs, which allows applying semantic-based static analyses and transformations to bound them, namely our method is based on the inference of *ranking functions* and *loop invariants* and on the use of *partial evaluation*.

Keywords Cost analysis, closed-form upper-bounds, resource analysis, automatic complexity analysis, static analysis, abstract interpretation, programming languages.

1 Introduction

Having information about the execution cost of programs, i.e., the amount of resources that the execution will require, is quite useful for many different purposes. Also, reasoning about execution cost is difficult and error-prone. Therefore, it is widely recognized that *cost analysis*, sometimes also referred to as *resource analysis* or *automatic complexity analysis*, is quite important. In this work we are interested in *static cost analysis*,

Elvira Albert · Puri Arenas · Samir Genaim
DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
E-mail: {elvira,puri,samir.genaim}@sip.ucm.es

Germán Puebla
CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain
E-mail: german@clip.dia.fi.upm.es

i.e., the analysis results for a program P should allow bounding the cost of executing P on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$.

The classical approach to static cost analysis consists of two phases. First, given a program and a *cost model*, the analysis produces *cost relations* (*CRs* for short), i.e., a system of recursive equations which capture the cost of the program in terms of the size of its input data. As a simple example, consider the following Java method m which traverses an array v and, depending whether the array elements are odd or even, invokes a different method m2 or m1:

```
public void m(int[ ] v) {
  int i=0;
  for (i=0; i<v.length; i++)
    if (v[i]%2==0) m1();
    else m2();
}
```

The following cost relations capture the cost of executing this program:

$$\begin{array}{lll}
(a)\ C_m(v) = k_1 + C_{for}(v,0) & \{v{\geq}0\} \\
(b)\ C_{for}(v,i) = k_2 & \{i{\geq}v, v{\geq}0\} \\
(c)\ C_{for}(v,i) = k_3 + C_{m1}() + C_{for}(v,i{+}1) & \{i{<}v, v{\geq}0\} \\
(d)\ C_{for}(v,i) = k_4 + C_{m2}() + C_{for}(v,i{+}1) & \{i{<}v, v{\geq}0\}
\end{array}$$

where v denotes the length of the array v, i stands for the counter of the loop and C_m, C_{m1} and C_{m2} approximate, respectively, the costs of executing the methods m, m1 and m2. The constraints attached to the equations contain their applicability conditions. For instance, equation (a) corresponds to the cost of executing the method m with an array of length greater that 0 (stated in the condition $\{v{\geq}0\}$), where a cost k_1 is accumulated to the cost of executing the loop, given by C_{for}. The constants $k_1, \ldots, k_4$ take different values depending on the cost model that one selects. For instance, if the cost model is the number of executed instructions, then k_1 is 1 which corresponds to the execution of the Java instruction "int i = 0;". If the cost model is the heap consumption, then k_1 is 0 since the previous instruction does not allocate any memory. Equations (c) and (d) capture, respectively, the costs of the then and the else branches. Note that, even if the program is deterministic, they are non-deterministic equations which contain the same applicability conditions. This is due to the fact that the array v is abstracted to its length and hence the values of its elements are unknown statically. Equation (b) captures the cost of exiting the loop.

Some interesting features of cost relations are that: (1) They are programming language independent: there are analyzers for many different languages which produce cost relations. (2) They are not limited to any complexity class. The same techniques can be used to infer cost which is logarithmic, exponential, etc. (3) They can be used for capturing a variety of non-trivial notions of resources, such as heap consumption, number of calls to a specific method, etc.

Though cost relations are simpler than the programs they originate from, since all variables are of integer type, in several respects they are not as static as one would expect from the result of a static analysis. One reason is that they are recursive and thus we may need to iterate for computing their value for concrete input values. Another reason is that even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in cost relations which are non-deterministic. This happens in the above example: since the array v has been abstracted

to its length v, the values of v[i] are unknown statically. Hence, the last two equations (c) and (d) become non-deterministic choices. In general, for finding the worst-case cost we may need to compute and compare (infinitely) many results. For both reasons, it is clear that it is interesting to compute *closed-form* upper-bounds for the cost relation, whenever this is possible, i.e., upper-bounds which are not in recursive form. For instance, for the above example, we aim at inferring the closed-form upper bound $k_1{+}k_2{+}v{*}max(\{k_3 + C_{m1}, k_4 + C_{m2}\})$ where C_{m1} and C_{m2} are in turn closed-form upper-bounds for the corresponding methods.

Since cost relations are syntactically quite close to *Recurrence Relations* (*RRs* for short), in most cost analysis frameworks, it has been assumed that cost relations can be easily converted into *RRs*. This has led to the belief that it is possible to use existing *Computer Algebra Systems* (CAS for short) for finding closed-forms in cost analysis. As we will show, cost relations are far from *RRs*. In this article, we present, to the best of our knowledge, the first practical framework for the fully automatic inference of reasonably accurate closed-form upper-bounds for *CRs* originating from a wide range of programs. The main novelty of our approach is that, by providing a semantics for *CRs*, we can view *CRs* as programs and, thus, apply semantic-based static analyses and transformations to automatically infer upper bounds for them. In particular, our main contributions are summarized as follows:

- We identify the differences between *CRs* and *RRs*, in Section 2.
- We provide a formal definition of *CRs* and their semantics in terms of *evaluation trees*, in Section 3. These notions are independent of the language and cost model.
- We present a general approximation scheme to infer closed-form upper-bounds in Section 4. Basically, it is based on the idea of bounding the cost of the corresponding evaluation trees. This requires computing upper bounds both on the *depth* of tree and also on the *cost* of nodes.
- In Section 5, we propose to use a specific form of ranking functions, which have been extensively studied in termination analysis (see e.g. [41]), to bound the depth of the evaluation tree.
- In Section 6, we present how to bound the cost of nodes by relying on loop invariants [21] and maximization operations.
- In Section 7, we develop an extension of our method to obtain more accurate upper bounds for divide and conquer programs which is based on counting *levels* in the evaluation tree rather than counting nodes.
- Our method can be used when *CRs* are directly recursive. We present in Section 8 an automatic program transformation, formalized in terms of *partial evaluation* (see e.g. [30]), which converts *CRs* into an equivalent directly recursive form.
- We report on a prototype implementation and apply it to obtain closed-form upper-bounds for *CRs* automatically generated from Java bytecode programs.

A preliminary version of this work appeared in the Proceedings of SAS'08 [10]. We have pursued cost relations as a language-independent target language for cost analysis in [3]. Our remaining previous work on cost analysis [4,5,7,8] is not related to this article but to the first phase in cost analysis which obtains, from a program and a cost model, a cost relation.

1.1 Applications of Upper Bounds of Cost Relations

Automatic cost analysis requires the inference of closed-form upper-bounds in order to be used within its large application field, which includes the following applications:

Resource Bound Certification. This research area deals with security properties involving resource usage requirements; i.e., the (untrusted) code must adhere to specific bounds on its resource consumption. The present work enables the automatic generation of non-trivial closed-form upper-bounds on cost. Such upper bounds could then be translated to *certificates*, in the proof-carrying code style. Previous work in this direction was restricted to *linear bounds* [23,11,29] and to *semi-automatic techniques* [19].

Performance Debugging and Validation. This application is based on automating the process of checking whether certain *assertions* about the efficiency of the program, possibly written by the programmer, hold or not. This application was already mentioned as future work in [50] and is available in the CiaoPP system for Prolog programs [27]. Our closed-form upper-bounds can be used to check whether the overall cost of an application meets the resource-consumption constraints specified in the assertions.

Program Synthesis and Optimization. This application was already mentioned as one of the motivations for [50]. Both in program synthesis and in semantic-preserving optimizations, such as partial evaluation (see e.g. [22,42]), there are multiple programs which may be produced in the process, with possibly different efficiency levels. Here, upper bounds on the cost can be used for guiding the selection process among a set of candidates.

2 Cost Relations vs. Recurrence Relations

The aim of this section is to identify the differences between cost relations and traditional recurrence relations. For this purpose, we take a close look at the *CRs* which appear in cost analysis of real programs. Figure 1 shows a Java program which we use as running example. We explain in detail, in Section 2.1 below, the *CRs* produced for this program by the automatic cost analysis of [4]. Then, in Section 2.2 we discuss the differences with *RRs*.

2.1 Cost Relations for the Running Example

Consider the Java code in Figure 1. It uses a List class for (non sorted) linked lists of integers which is implemented in the usual way. The del method receives as input: l, a list without repetitions; p, an integer value (the *pivot*); a and b, two sorted arrays of integers; and la and lb, two integers which indicate, respectively, the number of positions occupied in a and b. The a (resp. b) array is expected to contain values which are smaller (resp. greater or equal) than p, the pivot. Under the assumption that all values in l are contained in either a or b, the method del removes all values in l from the corresponding arrays. The rm_vec auxiliary method removes a given value e from an array a of length la and returns a's new length, la−1.

```
static void del(List l, int p, int a[], int la, int b[], int lb){
   while (l!=null){                    // cost equations (2),(3),(4)
      if (l.data < p)  la=rm_vec(l.data,a,la);
      else             lb=rm_vec(l.data,b,lb);
      l=l.next;
   }
}
static int rm_vec(int e, int a[], int la){
   int i=0;
   while (i < la && a[i]<e) {i++;};    // cost equations (5),(6),(7)
   for (int j=i; j<la-1; j++) a[j]=a[j+1]; // cost equations (8),(9)
   return la-1;
}
```

Fig. 1 Java code of running example

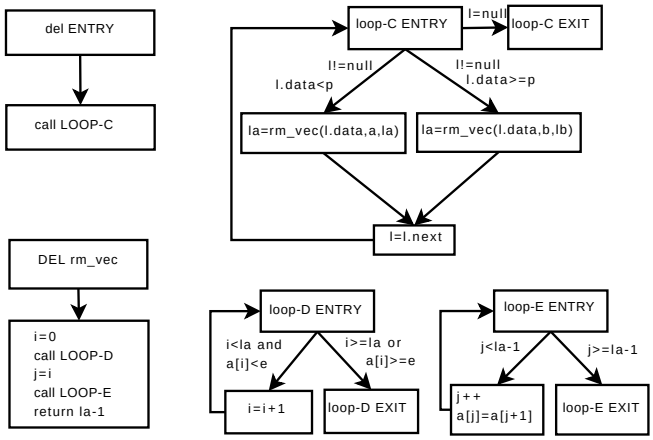


Fig. 2 Control flow graphs for running example

Example 1 In Figure 2, we show the control flow graphs (CFG) constructed by COSTA in order to generate automatically the *CRs*. Such CFGs correspond to the graphs for the two methods (del and rm_vec) and separate CFGs for the loops, as in COSTA loop extraction is performed mainly for efficiency issues (see [2]). Although [4] analyzes Java bytecode and not Java source, we show the source for clarity of the presentation.

Figure 3 shows the *CRs* automatically generated by the system for the del method in Figure 1 using the CFGs in Figure 2. The syntax and semantics of *CRs* is explained in detail in Section 3. Briefly, cost relations are defined by means of equations, each of which has an associated set of constraints which is shown to the right of the equation. Intuitively, the *CRs* are obtained from the program after performing the following three main steps:

1. In the first step, the recursive structure of the cost relation is determined by observing the iterative constructs in the program. In the case of imperative programs, both loops and recursion produce recursive calls in the cost relation. The *CR* matches the structure of the program such that when the program contains an iterative construct, its *CR* has a recursion. To carry out this step, analyzers usually build

(1) $Del(l,a,la,b,lb) = 1+C(l,a,la,b,lb)$	$\{l \geq 0, a \geq la, la \geq 0, b \geq lb, lb \geq 0\}$
(2) $C(l,a,la,b,lb) = 2$	$\{l = 0, a \geq la, a \geq 0, b \geq lb, b \geq 0\}$
(3) $C(l,a,la,b,lb) = 25+D(a,la,0)+E(la,j)+$ $C(l',a,la-1,b,lb)$	$\{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, j \geq 0, l > l'\}$
(4) $C(l,a,la,b,lb) = 24+D(b,lb,0)+E(lb,j)+$ $C(l',a,la,b,lb-1)$	$\{l > 0, a \geq la, a \geq 0, b \geq lb, b \geq 0, j \geq 0, l > l'\}$
(5) $D(a,la,i) = 3$	$\{i \geq la, a \geq la, i \geq 0\}$
(6) $D(a,la,i) = 8$	$\{i < la, a \geq la, i \geq 0\}$
(7) $D(a,la,i) = 10+D(a,la,i+1)$	$\{i < la, a \geq la, i \geq 0\}$
(8) $E(la,j) = 5$	$\{j \geq la-1, j \geq 0\}$
(9) $E(la,j) = 15+E(la,j+1)$	$\{j < la-1, j \geq 0\}$

Fig. 3 Cost relations generated by cost analysis of running example

CFGs. In our example, we have three recursive cost relations C, D and E which correspond to the three CFGs for the loops in Figure 2:

- C : cost of the while loop in del,
- D : cost of the while loop in rm_vec,
- E : cost of the for loop in rm_vec.

For readability, the *CRs* in Figure 3 are shown after performing *partial evaluation*, as we will explain in Section 8. This explains why there is no relation for the method rm_vec: the calls to rm_vec have been unfolded within its calling context, i.e., they have been replaced by the right hand side of the corresponding equation.

2. In the second step, static analysis techniques are used in order to approximate how the *size* of variables change from one call in the cost relation to another. Each program variable is abstracted using a *size measure* such that every non-integer value is represented as a natural number. Classical size measures used for non-integer types are: array length for arrays, list length for lists, the length of the longest reference path for linked data structures, etc. In the above example, l represents the *path-length* [47] of the corresponding dynamic structure, which in this case coincides with the length of the list; a and b are the lengths of the corresponding arrays. Since la and lb are numeric (integer) variables, the *CR* directly handles those values, i.e., no abstraction is required for them. Analysis is often done by obtaining an *abstract* version of the program by relying on abstract interpretation [20]. Essentially, the abstraction consists in inferring *size constraints*, sometimes also referred to as *size relations*, between the program variables at different program points. In Figure 3, such size relations are shown to the right of the equations. They are usually expressed by means of linear constraints. We refer to such abstraction by *size abstraction* and to an analysis that infers such relations by *size analysis*.
3. In the last step, instructions in the original program are replaced by the cost they represent. In the running example, we count the number of bytecode instructions executed such that each Java instruction corresponds to several bytecodes. It is not a concern of this paper to understand how bytecode instructions are related to Java statements. Hence, we omit explanations about the inferred constants in the equations.

After applying the above steps, the analyzer can set up the *CRs* shown in Figure 3 which we explain below. Equation (1) defines the cost of method del as 1 bytecode instruction plus the cost of the call to C. Observe also that the set of constraints contain applicability conditions (i.e., *guards*) for each equation, if any, by providing constraints

which only affect a subset of the variables in the left hand side (lhs for short). For clarity, we have inlined equality constraints (e.g., inlining equality $lb' = lb - 1$ is done by replacing all occurrences of lb' by $lb - 1$). The constraints attached to (1) are the (abstract) preconditions of the program. Among them, we have $a \geq la$ (resp. $b \geq lb$), which requires that the number of elements occupied in each array is less or equal than its length. Such preconditions are propagated properly to the rest of the equations.

In addition to Del, we have three recursive relations. As regards E, Equation (8) is its base case and it corresponds to the exit from the for loop, whereas Equation (9) counts the cost of each iteration in the loop. As expected, the value of j is increased by one at the recursive call to E. As regards the cost relation D, we have two base cases, Equations (5) and (6), which correspond to the exits from the loop because $\mathsf{i} \geq \mathsf{la}$ and because $\mathsf{a}[\mathsf{i}] \geq \mathsf{e}$, respectively. The important point here is that the second condition does not appear in the constraints of Equation (6) because this condition is not observable after abstracting the array a to its length, i.e., the value in a[i] is *unknown*. For the selected cost model, we count 3 bytecode instructions in the first base case and 8 in the second one. The cost of executing an iteration of the loop is captured by (7), where the condition $i < la$ must be satisfied and variable i is increased by one at each recursive call.

Finally, in relation C, Equation (2) corresponds to the case of an empty list, indicated by the condition $l = 0$. Equations (3) and (4) correspond, respectively, to the then and else branches of the if-then-else construct within the while loop. Hence, both of them contain the relation $l > 0$. Note that, as before, the conditions $\mathsf{l.data} < \mathsf{p}$ and $\mathsf{l.data} \geq \mathsf{p}$ in the Java program do not appear in the constraints attached to Equations (3) and (4) as they are not preserved by the corresponding size abstraction. The calls to D and E in (3) capture the cost of executing the method rm_vec for a and la. In the constraints, la decreases by one upon exit from rm_vec. l' corresponds to the length of the list when we perform the recursive call. It is ensured that the size of l has decreased ($l > l'$), but due to the size abstraction, we do not know how much. This is because the size analysis for heap allocated data structures used in [4] is based on path-length analysis, where size relations are expressed using $>$ and $\geq$ only. Equation (4) is similar to (3) but for b and lb instead of a and la. Note that when calling E in equations (3) and (4), a fresh variable j is used since we do not know the value that j can take after executing the while loop. We only know that $j \geq 0$, as it appears in the attached constraint. □

Importantly, if the program was written in a different programming language, the first phase in cost analysis would produce a similar cost relation which differs essentially only on intermediate equations and on the constants which are counted. This step is outside the scope of this article (see Section 11 for references to this phase in several programming languages). Our approach for computing closed-form upper-bounds takes as input cost relations which originate from programs written in any programming language.

2.2 Why Cost Relations are not Recurrence Relations ?

As can be seen in the *CRs* in the example, *CRs* differ from standard *RRs* in the following ways:

(a) Non-determinism. In contrast to *RRs*, *CRs* are highly non-deterministic: equations for the same relation are not required to be mutually exclusive. Even if the programming language is deterministic, *size abstractions* introduce a loss of precision: some guards which make the original program deterministic may not be observable when using the size of arguments instead of their actual values. In Example 1, this happens between Equations (3) and (4) and also between (6) and (7).

(b) Inexact constraints. *CRs* may have constraints other than equalities, such as $l > l'$. When dealing with realistic programming languages which contain non-linear data structures, such as trees, it is often the case that size analysis does not produce exact results. E.g., analysis may infer that the size of a data structure strictly decreases from one iteration to another, but it may be unable to provide the precise reduction. This happens in Example 1 in Equations (3) and (4).

(c) Multiple arguments. *CRs* usually depend on several arguments that may increase (variable i in Equation (7)) or decrease (variable l in Equation (2)) at each iteration. In fact, the number of times that a relation is executed can be a combination of several of its arguments. E.g., relation E is executed $la - j - 1$ times.

Point (a) is an obvious source of non-determinism and it was already detected in [50]. Point (b) is another source of non-determinism. Though it may not be so evident in small examples, it is almost unavoidable in programs handling trees or when numeric value analysis loses precision. As a result of (a) and (b), strictly speaking, *CRs* do not define functions, but rather relations: given a relation C and input values $\bar{v}$, there may exist multiple output values for $C(\bar{v})$. As regards point (c), most existing solvers can only handle single-argument recurrences (Mathematica is an exception). Sometimes it is possible to automatically convert relations with several arguments into relations with only one. However, this approach only provides correct results when the *CR*, in addition to the recursive calls themselves, only has constant value expression in the right hand side (rhs for short). Note that this is in general not the case except for toy *CRs*.

The above differences make existing methods for solving *RRs* insufficient to bound *CRs*, since they do not cover points (a), (b), and (c) above. On the other hand, CASs can solve complex recurrences (e.g., coefficients to function calls can be polynomials) which our framework cannot handle. However, this additional power is not needed in cost analysis, since such recurrences do not occur as the result of cost analysis.

Given a (non-deterministic) cost relation, it is sometimes useful to define a cost *function*. A relatively straightforward way of obtaining a cost function from non-deterministic *CRs* would be to introduce a maximization operator. Unfortunately, the cost functions thus produced are not very useful since existing CAS do not support the maximization operator. Adding it is far from trivial, since computing the maximum when the equations are not mutually exclusive requires taking into account multiple possibilities, which results in a highly combinatorial problem. This combinatorial explosion also affects the use of such cost-bound function in dynamic approaches, i.e., those based on executing cost-bound functions, such as [26].

Another approach is to obtain a cost-bound function by eliminating non-determinism. For this, we need to remove equations from *CRs* as well as sometimes to replace inexact constraints by exact ones while preserving the worst-case solution. However, this is not possible in general. E.g., in Figure 3, the maximum cost is obtained when the execution interleaves Equations (3) and (4), and therefore the worst case cannot be

achieved if we remove either equation. In other words, the upper bound obtained by removing either of Equations (3) and (4) is not an upper bound of the original *CR*.

Finally, let us observe that the properties listed above are all evident properties of constraint programs whose arguments are integer values. This explains the fact that we treat *CR* as programs and apply analysis and transformations delevoped for programming languages on them.

3 Cost Relations: Syntax and Semantics

Let us introduce some notation and preliminary definitions. The sets of natural, integer and real values are denoted respectively by $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$. The sets of non-negative integer and real values are denoted respectively by $\mathbb{Z}_+$ and $\mathbb{R}_+$. We use v and w for values from $\mathbb{Z}$ and $\mathbb{Z}_+$, r for values from $\mathbb{R}$ and $\mathbb{R}_+$, and n for values from $\mathbb{N}$. We write x, y, and z, to denote variables which range over $\mathbb{Z}$. Given any entity t, $vars(t)$ refers to the set of variables occurring in t. The notation $\bar{t}$ stands for a sequence of entities $t_1, \ldots, t_n$, for some $n > 0$. For simplicity, we sometimes interpret these sequences as sets. We use $t[\bar{y}/\bar{x}]$ to denote the renaming of the variables $\bar{x}$ by $\bar{y}$.

A *linear expression* has the form $v_0 + v_1 x_1 + \cdots + v_n x_n$. A *linear constraint* c (over $\mathbb{Z}$) has the form $l_1 \leq l_2$ where l_1 and l_2 are linear expressions. For simplicity, we write $l_1 = l_2$ instead of $l_1 \leq l_2 \wedge l_2 \leq l_1$, and $l_1 < l_2$ instead of $l_1 + 1 \leq l_2$. Note that constraints with rational coefficients can be always transformed to equivalent constraints with integer coefficients, e.g., $\frac{1}{2}x > y$ is equivalent to $x > 2y$. We write φ, ψ or ϕ, possibly subscripted, to denote sets of linear constraints, i.e., of the form $\{c_1, \ldots, c_n\}$, which should be interpreted as the conjunction $c_1 \wedge \cdots \wedge c_n$. We write $\bar{x} = \bar{y}$ to denote $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$ and $\varphi_1 \models \varphi_2$ to indicate that the (set of) linear constraints φ_1 implies the (set of) linear constraints φ_2. An assignment σ over a tuple of variables $\bar{x}$ is a mapping from $\bar{x}$ to $\mathbb{Z}$. Sometimes we denote an assignment over $\bar{x}$ as $\bar{x} = \bar{v}$, therefore we might write $\sigma \models \varphi$ for $\bar{x} = \bar{v} \models \varphi$. The projection operator $\exists \bar{x}.\varphi$ (resp. $\bar{\exists} \bar{x}.\varphi$) projects the polyhedron defined by φ on the space $vars(\varphi) \setminus \bar{x}$ (resp. $\bar{x}$).

The following definition presents our notion of *basic cost expression*, which characterizes syntactically the kind of expressions we deal with. Such expressions will be crucial to characterize the cost relation systems defined in the next section.

Definition 1 (basic cost expression) A symbolic expression $\mathtt{exp}$ is a *basic cost expression* if it can be generated using the grammar below:

$$\mathtt{exp} ::= r \mid \mathsf{nat}(l) \mid \mathtt{exp} + \mathtt{exp} \mid \mathtt{exp} * \mathtt{exp} \mid \mathtt{exp}^r \mid \log_n(\mathtt{exp}) \mid n^{\mathtt{exp}} \mid \max(S) \mid \mathtt{exp} - r$$

where $r \in \mathbb{R}_+$, l is a linear expression, S is a non empty set of basic cost expressions, $\mathsf{nat} : \mathbb{Z} \to \mathbb{Z}_+$ is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$, and $\mathtt{exp}$ satisfies that for any assignment $\sigma : vars(\mathtt{exp}) \mapsto \mathbb{Z}$ we have that $[\![\mathtt{exp}]\!]_\sigma \in \mathbb{R}_+$, where $[\![\mathtt{exp}]\!]_\sigma$ is the result of evaluating $\mathtt{exp}$ w.r.t. σ.

Basic cost expressions are symbolic expressions which represent the resources we accumulate and are the non-recursive building blocks for defining cost relations and for the closed-form upper-bounds that we infer for them. Cost expressions enjoy two crucial properties: (1) By definition, they are always evaluated to non-negative values, for instance, the expression $\mathsf{nat}(x) - 1$ is not a cost expression, since its evaluated to negative numbers for $x \leq 0$, however, $\mathsf{nat}(x - 1)$ is a valid cost expression; and (2) They are

monotonic in their nat components, i.e., replacing a sub-expression $\mathtt{nat}(l)$ by $\mathtt{nat}(l')$ such that $l' \geq l$, results in an upper bound of the original expression. This is essential for defining the maximization procedure *ub_exp*, which is defined in Section 6.2.

Proposition 1 *Let* exp *be a basic cost expression, l and l' be linear expressions and φ be a set of linear constraints such that $\varphi \models l' \geq l$. Let* $\mathtt{exp}'$ *be the result of replacing an occurrence of* $\mathtt{nat}(l)$ *in* exp *by* $\mathtt{nat}(l')$*. Then for any assignment σ for $vars(\mathtt{exp}') \cup vars(\mathtt{exp})$, if $\sigma \models \varphi$ then $[\![\mathtt{exp}']\!]_\sigma \geq [\![\mathtt{exp}]\!]_\sigma$.*

Proof By structural induction on basic cost expressions: (1) for expressions of the form $\mathtt{nat}(l)$ the result follows from $\sigma \models \varphi$ and $\varphi \models l' \geq l$, which implies $[\![l']\!]_\sigma \geq [\![l]\!]_\sigma$; and (2) for the induction step, composing expressions as described in Definition 1 preserves trivially the monotonicity property. □

A *cost relation* C of arity n is a subset of $\mathbb{Z}^n \times \mathbb{R}_+$. We use C and D to denote cost relations. Although the standard techniques for solving recurrences focus on solving (i.e., finding a closed-form) for a recurrence at a time, the cost analysis of a program, as seen in Example 3, in general produces a bunch of interconnected cost relations. I.e., the cost relation for the main function contains calls to cost relations which represent the cost of other functions or program blocks. We refer to such sets of cost relations produced by cost analysis as *Cost Relation Systems* (*CRSs* for short), which are formally defined as follows.

Definition 2 (Cost Relation System) A *cost relation system* $\mathcal{S}$ is a finite set of equations of the form $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \rangle$ with $k \geq 0$, where C and all D_i are cost relation symbols, all variables $\bar{x} \cup \bar{y}_i$ are distinct variables; exp is a basic cost expression; and φ is a set of linear constraints over $\bar{x} \cup vars(\mathtt{exp}) \bigcup_{i=1}^{k} \bar{y}_i$.

In contrast to standard definitions of *RRs*, in *CRSs*, the variables which occur in the rhs of the equations do not need to be related to those in the left hand side (lhs for short) by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by $rel(\mathcal{S})$ the set of cost relation symbols which are defined in $\mathcal{S}$, i.e., which appear in the lhs of some equation in $\mathcal{S}$. Given a *CRS* $\mathcal{S}$ and a cost relation symbol C, the definition of C in $\mathcal{S}$, denoted $def(\mathcal{S}, C)$, is the subset of the equations in $\mathcal{S}$ whose lhs is of the form $C(\bar{x})$. Without loss of generality, we assume that all equations in $def(\mathcal{S}, C)$ have the same variable names in the lhs, and that $\mathcal{S}$ is self-contained in the sense that all cost relation symbols which appear in the rhs of an equation in $\mathcal{S}$ must be in $rel(\mathcal{S})$.

Intuitively, a cost equation $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \rangle$ states that the cost of $C(\bar{x})$ is exp plus the sum of the cost of all $D_i(\bar{y}_i)$ where the linear constraints φ contain the applicability conditions for the equation as well as size relations for the equation variables. We now provide a formal (denotational) semantics for *CRSs* in terms of calls and answers. It is based on the notion of *evaluation tree* for a call $C(\bar{v})$. We will represent evaluation trees using nested terms of the form $node(Call, Local\_Cost, Children)$, where *Local_Cost* is a constant in $\mathbb{R}_+$ and *Children* is a sequence of evaluation trees.

Definition 3 (evaluation tree) A tree $node(C(\bar{v}), r, \langle T_1, \ldots, T_k \rangle)$ is an *evaluation tree* for $C(\bar{v})$ in $\mathcal{S}$, if

1. there exists a renamed apart equation $\mathcal{E} = \langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi \rangle \in \mathcal{S}$ and an assignment $\sigma : vars(\mathcal{E}) \mapsto \mathbb{Z}$ such that $\sigma \models \bar{x} = \bar{v} \wedge \varphi$; and

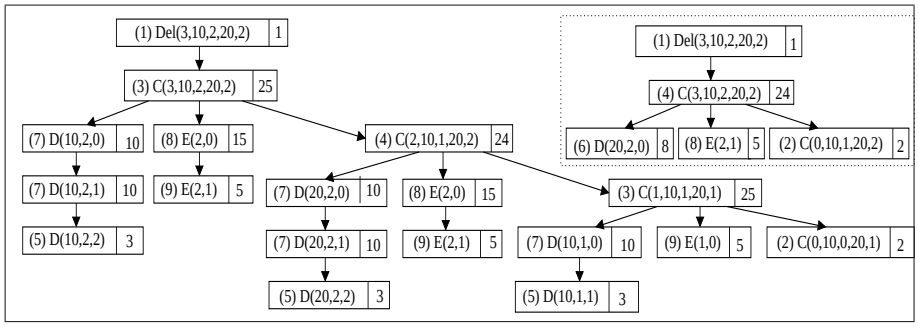


Fig. 4 Two evaluation trees for $Del(3, 10, 2, 20, 2)$

2. $r = [\![\texttt{exp}]\!]_\sigma$; and
3. $\forall 1 \leq i \leq k$ we have that T_i is an evaluation tree for $D_i(\bar{v}_i)$ and $\sigma \models \bar{y}_i = \bar{v}_i$.

Intuitively, we first look for an equation $\mathcal{E}$ such that it is applicable for solving $C(\bar{v})$ and for an assignment σ which is consistent with $\mathcal{E}$'s guard (1). Then (2) tells us that r is obtained by evaluating `exp`. Finally, (3) requires that each T_i be an evaluation tree of the corresponding call $D_i(\bar{v}_i)$. Note that step 1 is non-deterministic as there might be several equations for C and (infinitely many) different assignments σ that satisfy $\bar{x} = \bar{v} \wedge \varphi$. Therefore, we write $\mathit{Trees}(C(\bar{v}), \mathcal{S})$ to denote the set of all evaluation trees for $C(\bar{v})$. Now, we define

$$\mathit{Answers}(C(\bar{v}), \mathcal{S}) = \{\mathsf{Sum}(T) \mid T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})\}$$

where $\mathsf{Sum}(T)$ is the sum of all cost expressions in T, i.e., $\mathsf{Sum}(\mathit{node}(C(\bar{v}), r, \langle T_1, \ldots, T_k\rangle)) = r + \sum_{i=0}^{k} \mathsf{Sum}(T_i)$. A cost-bound function $C_+(\bar{x})$ can be defined as $C_+(\bar{v}) = \max(\mathit{Answers}(C(\bar{v}), \mathcal{S}))$. Clearly, it is not always computable as we might have infinite trees. Note that the branching in each tree is conjunctive and corresponds to the different calls in the body, an that the disjunction comes in the form of multiple trees for the same query.

Example 2 Figure 4 shows two possible evaluation trees for $Del(3, 10, 2, 20, 2)$ in $\mathcal{S}$, where $\mathcal{S}$ is the *CR* in Figure 3. The tree on the left has maximal cost, whereas the one on the right has minimal cost. Nodes are represented using boxes split in two parts. The part on the left contains a call, e.g., $Del(3, 10, 2, 20, 2)$ in the root nodes of both trees, annotated with a number in parenthesis, e.g., (1) in such nodes, which indicates the equation which was selected for evaluating such call. The part on the right contains the local cost associated to the call, 1 in both root nodes. Nodes are linked by arrows to their children, if any.

The two trees differ in that, for solving $C(3, 10, 2, 20, 2)$, in the one on the left we pick Equation (3) and in the one on the right we pick Equation (4). Furthermore, in the recursive call to C in Equations (3) and (4) we always assign $l' = l - 1$ in the tree on the left and we assign $l' = l - 3$ in the tree on the right. Note that both possibilities are valid w.r.t. $\mathcal{S}$, since we are allowed to pick any value l' such that $l' < l$. The tree on the left corresponds to a possible execution of the program. However, the tree on the right does not correspond to any actual execution. This is a side effect of using safe approximations in static analysis for computing size abstractions: information is correct

in the sense that given a concrete program execution, at least one of the evaluation trees must correspond to such execution, but there may be other trees which do not correspond to any valid execution. Therefore, *CRSs* provide information which is sound but possibly imprecise.

As this example shows, there may be multiple evaluation trees for a call. In fact, there may even be infinitely many of them. The latter happens in our example call, as step 1 in Definition 3 can provide an infinite number of assignments to variable j which are compatible with the constraint $j \geq 0$ in Equations (3) and (4). This shows that approaches like [26] based on evaluation of *RRs* may not be of general applicability in *CRSs*, as size relations can be inexact and multiple, or even infinitely many evaluation trees may exist. Fortunately, since we are not interested in executing *CRSs* but rather on finding closed-form (i.e., static) upper bounds for them, whether there are infinitely many evaluation trees for a call is not directly an issue, as long as there are not infinitely many *different* answers. In our example, $\mathit{Trees}(\mathit{Del}(3,10,2,20,2),\mathcal{S})$ is an infinite set, but infinitely many of the trees in this set produce equivalent results and $\mathit{Answers}(\mathit{Del}(3,10,2,20,2)),\mathcal{S})$ is finite. Thus, it is in principle possible to find an upper bound for it. □

4 Closed-Form Upper-Bounds for Cost Relations

After providing a suitable semantics for *CRs*, we now study how to obtain closed-form upper bounds for them. In what follows, we are only interested in upper bound functions which are in closed-form. Therefore, for brevity, we often just write 'upper bound' instead of 'closed-form upper-bound'.

A function $f : \mathbb{Z}^n \mapsto \mathbb{R}_+$ is in *closed-form* if it is defined as $f(\bar{x}) = \mathtt{exp}$, where $\mathtt{exp}$ is a basic cost expression and $\mathit{vars}(\mathtt{exp}) \subseteq \bar{x}$. Let C be a relation over $\mathbb{Z}^n \times \mathbb{R}_+$. A closed-form function $U : \mathbb{Z}^n \mapsto \mathbb{R}_+$ is an *upper bound* of C if $\forall \bar{v} \in \mathbb{Z}^n$ and $\forall r \in \mathit{Answers}(C(\bar{v}),\mathcal{S})$ we have $U(\bar{v}) \geq r$. Given a relation (or function) C, we use C_+ to refer to an upper bound of C.

4.1 Standalone Cost Relations

An important feature of *CRSs*, also present in *RRs*, is their *compositionality*. This allows computing upper bounds of *CRSs* composed of multiple relations by concentrating on one relation at a time. Let us consider an equation $\mathcal{E}$ for a cost relation $C(\bar{x})$ where a call of the form $D(\bar{y})$, with $D \neq C$ appears on the rhs of $\mathcal{E}$. In order to compute an upper bound of $C(\bar{x})$, we can replace $\mathcal{E}$ by another equation $\mathcal{E}'$ where the call to $D(\bar{y})$ is replaced by a call to an upper bound $D_+(\bar{y})$, already in closed-form. The resulting cost relation is trivially an upper bound of the original one. E.g., suppose that we have the following upper bounds:

$$\begin{aligned} E_+(la,j) &= 5 + 15 * \mathsf{nat}(la - j - 1) \\ D_+(a,la,i) &= 8 + 10 * \mathsf{nat}(la - i) \end{aligned}$$

Replacing the calls to D and E in Equations (3) and (4) by D_+ and E_+ results in the *CR* shown in Figure 5.

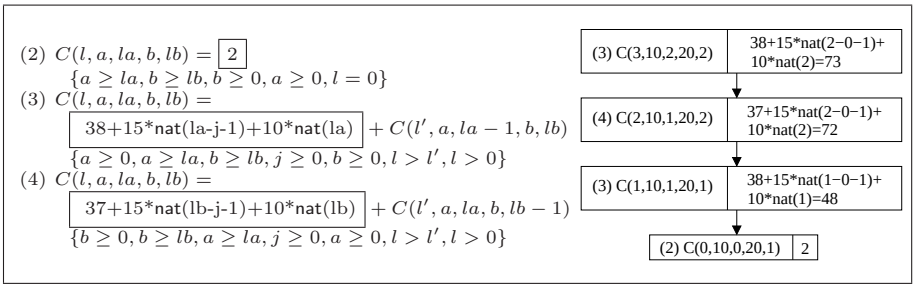


Fig. 5 Standalone *CR* for relation C and a corresponding evaluation tree

The compositionality principle only results in an effective mechanism if all recursions are *direct* (i.e., all cycles are of length one). In that case we can start by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as *standalone cost relations* and continue by replacing the computed upper bounds on the equations which call such relations. In the following, we formalize our method by assuming standalone cost relations and, in Section 8, we provide a mechanism for obtaining direct recursion automatically.

4.2 Approximating Evaluation Trees

Existing approaches to compute upper bounds and asymptotic complexity of *RRs*, usually applied by hand, are based on reasoning about evaluation trees in terms of their size, depth, number of nodes, etc. They typically consider two categories of nodes: (1) *internal* nodes, which correspond to applying recursive equations, and (2) *leaves* of the tree(s), which correspond to the application of a base (non-recursive) case. The central idea then is to count (or obtain an upper bound on) the number of leaves and the number of internal nodes in the tree separately and then multiply each of these by an upper bound on the cost of the base case and of a recursive step, respectively. For instance, in the evaluation tree in Figure 5 for the standalone cost relation C, there are three internal nodes and one leaf. The values in the internal nodes, once performed the evaluation of the expressions are 73, 72, and 48, therefore 73 is the worst case. In the case of leaves, the only value is 2. Therefore, the tightest upper bound we can find using this approximation is $3 \times 73 + 1 * 2 = 221 \ \geq \ 73 + 72 + 48 + 2 = 193$.

We now extend the approximation scheme mentioned above in order to consider all possible evaluation trees which may exist for a call. In the following, we use $|S|$ to denote the cardinality of a set S. Also, given an evaluation tree T, $\mathit{leaf}(T)$ denotes the set of leaves of T (i.e., those without children) and $\mathit{internal}(T)$ denotes the set of internal nodes (all nodes but the leaves) of T.

Proposition 2 (node-count upper-bound) *Let C be a cost relation. We define:*

$$C_+(\bar{x}) = \mathit{internal}_+(\bar{x}) * \mathit{costr}_+(\bar{x}) + \mathit{leaf}_+(\bar{x}) * \mathit{costnr}_+(\bar{x})$$

where $\mathit{internal}_+(\bar{x})$, $\mathit{costr}_+(\bar{x})$, $\mathit{leaf}_+(\bar{x})$ and $\mathit{costnr}_+(\bar{x})$ are closed-form functions defined on $\mathbb{Z}^n \mapsto \mathbb{R}_+$. Then, C_+ is an upper bound of C if for all $\bar{v} \in \mathbb{Z}^n$ and for all $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$, the following properties hold:

1. $internal_+(\bar{v}) \geq |internal(T)|$ *and* $leaf_+(\bar{v}) \geq |leaf(T)|$;
2. $costr_+(\bar{v})$ *is an upper bound of* $\{r \mid node(\_, r, \_) \in internal(T)\}$ *and*
3. $costnr_+(\bar{v})$ *is an upper bound of* $\{r \mid node(\_, r, \_) \in leaf(T)\}$.

Proof The proposition is trivially correct by the definition of upper bound and *Answers*. □

This proposition presents the main approximation approach which we use for computing upper bounds. Our main contribution is to come up with mechanisms to infer the four functions appearing above.

5 Upper Bounds on the Number of Nodes

In this section, we present an automatic mechanism for obtaining correct $internal_+(\bar{x})$ and $leaf_+(\bar{x})$ functions which *statically* provides upper bounds of the number of internal nodes and leaves in evaluation trees. The basic idea is to first obtain upper bounds on the *branching factor* (denoted b) and *height* (the distance from the root to the deepest leaf) of all corresponding evaluation trees (denoted $h_+(\bar{x})$) and, then, use the number of internal nodes and leaves of a *complete* tree with such branching factor and height as an upper bound. Well-known formulas exist which, given the branching factor and the height of the tree, compute the number of nodes of the *complete* tree. As usual, a tree is complete when all internal nodes have as many children as indicated by the branching factor and leaves are at the same depth. Clearly, complete trees provide an upper bound of the number of nodes of any tree with such height and branching factor. Therefore, we define $internal_+(\bar{x})$ and $leaf_+(\bar{x})$ as follows:

$$leaf_+(\bar{x}) = b^{h_+(\bar{x})} \qquad internal_+(\bar{x}) = \begin{cases} h_+(\bar{x}) & b = 1 \\ \frac{b^{h_+(\bar{x})}-1}{b-1} & b \geq 2 \end{cases}$$

For a cost relation C, the branching factor b in any evaluation tree for a call $C(\bar{v})$ is limited by the maximum number of recursive calls which occur in a single equation for C, which obviously can be computed statically. Note that we mean the actual occurrences of recursive calls in the right hand side of the equations which determines the complexity scheme (exponential, polynomial, etc.) not how many calls will actually be performed in a concrete execution. This is not related to how the arguments increase or decrease.

We now propose a way to compute an upper bound for the height, h_+. Given an evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$ for a cost relation C, consecutive nodes in any branch of T represent consecutive recursive calls which occur during the evaluation of $C(\bar{v})$. Therefore, bounding the height of a tree may be reduced to bounding consecutive recursive calls during the evaluation of $C(\bar{v})$. The notion of *loop* in a cost relation, which we introduce below, is used to model consecutive calls.

Definition 4 (loops) Let $\mathcal{E} = \langle C(\bar{x}) = \texttt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$ be an equation for a cost relation C. The set of *loops* induced by $\mathcal{E}$ is defined as:

$$Loops(\mathcal{E}) = \{\langle C(\bar{x}) \rightarrow C(\bar{y}_i), \varphi' \rangle \mid \varphi' = \bar{\exists}\bar{x} \cup \bar{y}_i.\varphi, 1 \leq i \leq k\}$$

Similarly, we define $Loops(C) = \cup_{\mathcal{E} \in def(\mathcal{S}, C)} Loops(\mathcal{E})$.

Intuitively, a loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi' \rangle$ over-approximates that evaluating $C(\bar{v}_1)$ such that $\bar{x} = \bar{v}_1 \models \varphi'$, may eventually be followed by an evaluation for $C(\bar{v}_2)$ such that $\bar{x} = \bar{v}_1 \wedge \bar{y} = \bar{v}_2 \models \varphi'$. In terms of evaluation trees, this means that the node corresponding to $C(\bar{v}_1)$ will have a child with $C(\bar{v}_2)$.

Example 3 The cost relation in Figure 5 induces the following two loops which correspond to Equations (3) and (4).

(3) $\langle C(l, a, la, b, lb) \rightarrow C(l', a, la', b, lb), \varphi'_1 \rangle$
where $\varphi'_1 = \{a \geq 0, a \geq la, b \geq lb, b \geq 0, l > l', l > 0, la' = la - 1\}$
(4) $\langle C(l, a, la, b, lb) \rightarrow C(l', a, la, b, lb'), \varphi'_2 \rangle$
where $\varphi'_2 = \{b \geq 0, b \geq lb, a \geq la, a \geq 0, l > l', l > 0, lb' = lb - 1\}$

□

The problem of bounding the number of consecutive recursive calls has been extensively studied in the context of termination analysis. Automatic termination analyzers usually prove that an upper bound of the number of iterations of the loop exists by proving that there exists a function f from the loop's arguments to a *well-founded* partial order, such that f decreases in any two consecutive calls. This in turn guarantees the absence of infinite traces, and therefore termination. These functions are usually called *ranking functions*. A difference w.r.t. termination analysis is that we aim at determining a concrete ranking function f, rather than just proving that it exists, which is usually enough for termination proofs. The following definition characterizes the kind of ranking functions we are interested in since, as we will see later, they are adequate for bounding the number of iterations of a loop.

Definition 5 (ranking function for a loop) A function $f : \mathbb{Z}^n \mapsto \mathbb{Z}$ is a *ranking function* for a loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi \rangle$ if the two conditions below are satisfied:

1. $\varphi \models f(\bar{x}) > f(\bar{y})$
2. $\varphi \models f(\bar{x}) > 0$.

An important thing to note is that the constraint φ is computed using static analysis and, therefore, part of the information may be lost. Thus, in order to find a ranking function it is crucial that φ keeps enough information. Condition 1 requires that f be decreasing in every iteration of the loop. In order to satisfy this condition it is required that the constraint φ captures information about the way in which the value of variables change from one iteration to another. Condition 2 requires that f be well-founded. In order to satisfy this condition it is required that φ captures sufficient information about the applicability conditions (guards) of the loop so as to identify cases where the loop does not apply.

In addition, since a cost relation may induce several loops (i.e., several possibilities for generating calls), we require the *ranking function* to decrease for all loops.

Definition 6 (ranking function for a cost relation) A function $f_C : \mathbb{Z}^n \mapsto \mathbb{Z}$ is a *ranking function* for C if it is a ranking function for all loops in $Loops(C)$.

Example 4 The function $f_C(l, a, la, b, lb) = l$ is a ranking function for C in the cost relation in Figure 5. Note that φ'_1 and φ'_2 in the loops of C in Example 3 contain the constraints $\{l > l', l > 0\}$ which is enough to guarantee that f_C is decreasing and well-founded. □

The following example illustrates that sometimes the ranking function involves several arguments.

Example 5 Consider the loop which originates from Equation (7) depicted in Figure 3. $\langle D(a, la, i) \to D(a, la, i'), \{i' = i + 1, i < la, a \geq la, i \geq 0\}\rangle$. The function $f_D(a, la, i) = la - i$ is a ranking function for the above loop. Any ranking function for D must involve both la and i. □

We propose to use ranking functions for cost relations as an upper bound on the number of consecutive calls (and therefore on the height of the corresponding evaluation trees). This is justified by the following two facts: (1) the ranking function decreases at least by one unit in each iteration when applying it on two consecutive calls (since its range is $\mathbb{Z}$); and (2) it is always greater than zero in the recursive equations (i.e., when applied to a call in an internal node).

In order to be able to define h_+ in terms of a ranking function, one thing to fix is that the ranking function is only guaranteed to return meaningful (positive) values for input values which are consistent with the loops of guards (Condition 2). Thus, if we apply the ranking function to input values which correspond to base cases (leaves of the tree) we may obtain negative values. Therefore, we define $h_+(\bar{x}) = \mathsf{nat}(f_C(\bar{x}))$. Function nat, introduced in Section 3, guarantees that negative values are lifted to 0 and, therefore, they provide a correct approximation for the height of evaluation trees with a single node.

Lemma 1 *Let $f_C(\bar{x})$ be a ranking function for a cost relation C. Then, $\forall \bar{v} \in \mathbb{Z}^n$ and $\forall\ T \in Trees(C(\bar{v}), \mathcal{S})$ it holds $\mathsf{nat}(f_C(\bar{v})) \geq h(T)$.*

Proof For $h(T) = 0$, the proof is straightforward as $\mathsf{nat}(f_C(\bar{v}))$ is non-negative. For $h(T) > 0$, assume the contrary, i.e., there exists an evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$ such that $h(T) = n > \mathsf{nat}(f_C(\bar{v}))$. This means there exists a path (starting from the root) which consists of $n + 1$ nodes. Let $C(\bar{v}_0), \ldots, C(\bar{v}_n)$ be the calls that correspond to the nodes in that path, where $\bar{v}_0 = \bar{v}$. By definition of ranking function for a cost relation, for all $i < n$, we have $f_C(\bar{v}_i) - f_C(\bar{v}_{i+1}) \geq 1$ and $f_C(\bar{v}_i) > 0$. Then, it holds that $f_C(\bar{v}) \geq n+1 > n = h(T)$, which contradicts the assumption that $f_C(\bar{v}) < h(T)$. □

Even though the ranking function provides an upper bound for the height of the corresponding trees, in some cases we can further refine it and obtain a tighter upper bound. For example, if the difference between the value of the ranking function in each two consecutive calls is guaranteed to be larger than a constant $\delta > 1$, then $\lceil \mathsf{nat}(\frac{f_C(\bar{x})}{\delta}) \rceil$ is a tighter upper bound. A more interesting case, if each loop $\langle C(\bar{x}) \to C(\bar{y}), \varphi\rangle \in Loops(C)$ satisfies $\varphi \models f_C(\bar{x}) \geq k * f_C(\bar{y})$ where $k > 1$ is a constant, then the height of the tree is bounded by $\lceil \log_k(\mathsf{nat}(f_C(\bar{v})) + 1) \rceil$, as each time the value of the ranking function decreases by k. For instance, given a loop the form: $\langle C(l) \to C(l'), \{l' = l/3, l > 0\}\rangle$, we find the bound "$\lceil \log_3(l) + 1 \rceil$" for the height of the tree. These cases are handled in our system (see Section 7).

6 Bounding the Cost per Node

After studying how to obtain upper bounds of the number of internal and leave nodes in evaluation trees, in this section, we present an automatic method to obtain functions

$costr_+(\bar{x})$ and $costnr_+(\bar{x})$, which are upper bounds of the local cost associated to an internal node and of a leave node, respectively. We first give an intuitive description of the technique on our running example. Consider the evaluation tree in Figure 5. There is only one leave node and its local cost is 2. Therefore, we can define $costnr_+(\bar{x}) = 2$. As regards the three internal nodes, observe that the corresponding expressions are instantiations of either:

$$\begin{aligned}\texttt{exp}_3 &= 38 + 15 * \mathsf{nat}(la - j - 1) + 10 * \mathsf{nat}(la)\\ \texttt{exp}_4 &= 37 + 15 * \mathsf{nat}(lb - j - 1) + 10 * \mathsf{nat}(lb)\end{aligned}$$

Knowing the expressions which generate the possible values in nodes is important, since if we know (or have a safe approximation of) the values of the variables which appear in such expressions, then it is possible to obtain an upper bound of the cost of nodes. Therefore, we split the construction of $costr_+(\bar{x})$ and $costnr_+(\bar{x})$ in the following two parts.

Invariants. First, it is necessary to know what are the possible values to which the different variables in $\texttt{exp}_3$ and $\texttt{exp}_4$ can be instantiated. Computing this information is usually undecidable or impractical, but it can be approximated (by means of a superset of the actual values) using static program analysis. One possible way to approximate it is to infer (linear) constraints between the values of the variable in each node and the initial values. For example, for the equations in Figure 5, we are interested in obtaining constraints between the root call $C(l_0, a_0, la_0, b_0, lb_0)$ and the call in any node $C(l, a, la, b, lb)$. Note that for a variable x we use x_0 to refer to the value of x at the root call. The following linear constraints describe a (possible) relation:

$$\psi = \{0 \leq l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}$$

In other words, ψ is a loop *invariant* that holds between the initial values $\{l_0, a_0, la_0, b_0, lb_0\}$ and the variables in any recursive call $C(l, a, la, b, lb)$ during the evaluation.

Upper Bounds of Cost Expressions. The invariant can then be used to infer upper bounds for $\texttt{exp}_3$ and $\texttt{exp}_4$. Since $\texttt{exp}_3$ and $\texttt{exp}_4$ are monotonic in their nat sub-expressions, as stated in Proposition 1, it is enough to obtain upper bounds for those sub-expressions in order to obtain upper bounds for $\texttt{exp}_3$ and $\texttt{exp}_4$. For maximizing $\texttt{exp}_3$, we need to compute an upper bound for $la - j - 1$ in the context of the invariant ψ conjoined with the local constraints φ_3, associated to Equation (3). By *maximizing* $la - j - 1$ w.r.t. $\{l_0, a_0, la_0, b_0, lb_0\}$, we infer that $la_0 - 1$ is an upper bound for $la - j - 1$ since $\psi \wedge \varphi_3 \models \{la \leq la_0, j \geq 0\}$. Similarly, we obtain the upper bounds la_0, $lb_0 - 1$ and lb_0 for la, $lb - j - 1$, and lb, respectively. By putting all pieces together we obtain that:

$$\begin{aligned}\texttt{mexp}_3 &= 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)\\ \texttt{mexp}_4 &= 37 + 15 * \mathsf{nat}(lb_0) + 10 * \mathsf{nat}(lb_0)\end{aligned}$$

are upper bounds for $\texttt{exp}_3$ and $\texttt{exp}_4$, respectively. Then, we use $\max(\{\texttt{mexp}_3, \texttt{mexp}_4\})$ as an upper bound for all possible expressions in the internal nodes of any possible evaluation tree for $C(l_0, a_0, la_0, b_0, lb_0)$. We now formalize the two steps that have been described above.

6.1 Invariants

Computing an *invariant*, in terms of linear constraints, that holds between the arguments at the initial call and at each call during the evaluation, can be done by using $Loops(C)$. Intuitively, if we know that a linear constraint ψ holds between the arguments of the initial call $C(\bar{x}_0)$ and the arguments of a specific recursive call $C(\bar{x})$ during the evaluation, denoted $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle$, and we have a loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi\rangle \in Loops(C)$, then we can apply the loop one more step and get the new *calling context* (or *context* for short) $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \exists \bar{x}.(\psi \wedge \varphi)\}\rangle$. The following definition describes how from a set of contexts I we learn more contexts by applying all loops in a relation. We denote by $\mathcal{R}$ the set of all possible contexts for C, and by $\wp(\mathcal{R})$ all subsets of C that include $I_0 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\bar{x}_0 = \bar{x}\}\rangle$.

Definition 7 (loop invariants) For a relation C, let $\mathcal{T}_C : \wp(\mathcal{R}) \mapsto \wp(\mathcal{R})$ be an operator defined:

$$\mathcal{T}_C(X) = \left\{ \langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \psi'\rangle \;\middle|\; \begin{array}{l} \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in X \\ \langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi\rangle \in Loops(C) \\ \psi' = \bar{\exists} \bar{x_0} \cup \bar{y}.(\psi \wedge \varphi) \end{array} \right\}$$

which derives a set of contexts, from a given context X, by applying all loops. The loop invariant I_C is defined as $\cup_{i \in \omega} \mathcal{T}_C^i(\{I_0\})$.

Example 6 Let us compute I_C for the loops that we have computed in Example 3. Let $\bar{x}_0 = \langle l_0, a_0, la_0, b_0, lb_0\rangle$ and $\bar{x} = \langle l, a, la, b, lb\rangle$. The initial context is

$$I_0 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l = l_0, a = a_0, la = la_0, b = b_0, lb = lb_0\}\rangle$$

In the first iteration we compute $\mathcal{T}_C^0(\{I_0\}) = \{I_0\}$. In the second iteration we compute $\mathcal{T}_C^1(\{I_0\})$, which results in the contexts

$$I_1 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, a = a_0, \underline{la = la_0 - 1}, b = b_0, lb = lb_0, \underline{l_0 > 0}\}\rangle$$
$$I_2 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, a = a_0, la = la_0, b = b_0, \underline{lb = lb_0 - 1}, \underline{l_0 > 0}\}\rangle$$

where I_1 and I_2 correspond to applying respectively the first and second loops on I_0. The underlined constraints are the modifications due to the application of the loop. Note that in I_1 (resp. I_2) the variable la_0 (resp. lb_0) decreases by one. The third iteration $\mathcal{T}_C^2(\{I_0\})$, i.e., $\mathcal{T}_C(\{I_1, I_2\})$, results in

$$I_3 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 2}, b = b_0, lb = lb_0, l_0 > 0\}\rangle$$
$$I_4 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 1}, b = b_0, \underline{lb = lb_0 - 1}, l_0 > 0\}\rangle$$
$$I_5 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, la = la_0, b = b_0, \underline{lb = lb_0 - 2}, l_0 > 0\}\rangle$$
$$I_6 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 1}, b = b_0, lb = lb_0 - 1, l_0 > 0\}\rangle$$

where I_3 and I_4 originate from applying the loops to I_1, and I_5 and I_6 from applying the loops to I_2. The modifications on the constraints reflect that, when applying a loop, either we decrease la or lb. After three iterations, the invariant I_C includes $\{I_0, \ldots, I_6\}$. More iterations will add more contexts that further modify the value of la or lb. Therefore, the invariant I_C grows indefinitely in this case. □

The following lemma guarantees that I_C, as defined in Definition 7, is a loop invariant, i.e., it holds between the initial call and any call in the corresponding evaluation tree.

Lemma 2 *Let $C(\bar{v})$ be a call, then $\forall T \in Trees(C(\bar{v}), \mathcal{S})$ and $\forall node(C(\bar{w}), \_, \_) \in T$, there exists $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in I_C$ such that $\{\bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w}\} \models \psi$.*

Proof Given an initial call $C(\bar{v})$ and an evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$, we show by induction that if $node(C(\bar{w}), \_, \_) \in T$ is at a level n (the level of the root is 0), then there exists $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in \cup_{0 \leq i \leq n} \mathcal{T}_C^i(\{I_0\})$ such that $\{\bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w}\} \models \psi$. Then, since $\mathcal{T}_C$ is continuous over the lattice $\langle \wp(\mathcal{R}), \{I_0\}, \mathcal{R}, \subseteq, \cup, \cap\rangle$, it holds for the least fixed point $I_C = \cup_{i \in \omega} \mathcal{T}_C^i(I_0)$ and any level.

Base case. If $n = 0$, it is obvious that the lemma holds using the initial context which is in $\mathcal{T}_C^0(\{I_0\})$.

Induction step. Assume the above lemma holds for any node at a level smaller than n. Consider a node $node(C(\bar{w}), \_, \_) \in T$ at level $n \geq 1$, and let its parent node be $node(C(\bar{w}'), \_, \_) \in T$. By the induction assumption, since the parent level is $n-1$, there exists $I = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in \cup_{0 \leq i < n} \mathcal{T}_C^i(\{I_0\})$ such that $\bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w}' \models \psi$. By the definition of $Loops(C)$, there exists a loop $\ell = \langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi\rangle \in Loops(C)$ such that $\bar{x} = \bar{w}' \wedge \bar{y} = \bar{w} \models \varphi$. Since the context I must have been introduced by $\mathcal{T}_C^k(\{I_0\})$ for some $k < n$, then at iteration $k+1 \leq n$ the operator $\mathcal{T}_C$ will use I and ℓ to generate $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{y}), \bar{\exists}\bar{x}_0 \cup \bar{y}.(\psi \wedge \varphi)\rangle \cup_{0 \leq i \leq n} \mathcal{T}_C^i(\{I_0\})$. Moreover, $\bar{x}_0 = \bar{v} \wedge \bar{y} = \bar{w} \models \bar{\exists}\bar{x}_0 \cup \bar{y}.(\psi \wedge \varphi)$. □

The problem with Definition 7 is that it is not computable in general since the invariant I_C possibly consists of an infinite number of calling contexts, as it happens in our example. In practice, we approximate I_C using abstract interpretation over, for instance, the domain of convex polyhedra [21]. For our example, as an approximation for I_C of Example 6 we obtain the invariant:

$$I_C^\alpha = \{\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}\rangle\}$$

In general, we usually approximate I_C by a single context $I_C^\alpha = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi'\rangle\}$ such that $\forall\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle \in I_C.\psi \models \psi'$. This is simply done by replacing $\cup$ in Definition 7 by a convex-hull operation, and applying a widening operator to guarantee termination [21]. It is clear that Lemma 2 also holds for such approximation of I_C.

6.2 Upper Bounds on Cost Expressions

At this point, we want to use the loop invariant in order to obtain upper bounds, in terms of the initial call values, for the values in all internal nodes and leaves in the corresponding evaluation trees. Since the values which appear in the nodes of evaluation trees correspond to different instantiations of the cost expressions in the cost equations, we concentrate first on finding upper bounds for those cost expressions and then combine them to build upper bounds for all internal nodes and all leaves.

Consider, for example, the expression $\mathsf{nat}(la - j - 1)$ which appears in Equation (3) of Figure 5. We want to infer an upper bound of the values that it can be evaluated to in terms of the input values $\langle l_0, a_0, la_0, b_0, lb_0\rangle$. We have inferred that $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi\rangle$ where $\psi = \{l \leq l_0, a = a_0, \underline{la \leq la_0}, b = b_0, lb \leq lb_0\}$, is a safe approximation of the loop invariant I_C, from which we can observe that the maximum value that la can take is la_0. In addition, from the local constraints φ of Equation (3) we know that $j \geq 0$. Since $la - j - 1$ takes its maximal value when la is maximal and j is minimal, the

expression $la_0 - 1$ is an upper bound for $la - j - 1$. In practice, this inference method can be done in a fully automatic way using linear constraints tools (e.g. [12]) as follow:

1. compute $\phi = \bar{\exists} l_0, a_0, la_0, b_0, lb_0, r.(\psi \wedge \varphi \wedge y = la - j - 1)$, where y is a new variable;
2. *syntactically* look in ϕ for an expression that can be rewritten to $y \leq f'$, where f' is a linear expression which (obviously) contains only variables from $\{l_0, a_0, la_0, b_0, lb_0\}$.

Given a cost equation $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$ and a safe approximation of its loop invariant $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$, the function below computes an upper bound for `exp` by maximizing its `nat` components:

```
1: function ub_exp(exp,x̄0,φ,ψ)
2:     mexp = exp
3:     for all nat(f)∈exp do
4:         φ = ∃̄x̄0, y.(φ ∧ ψ ∧ y = f)      // y is a fresh variable
5:         if ∃f' such that vars(f') ⊆ x̄0 and φ ⊨ y ≤ f' then mexp = mexp[nat(f)/nat(f')]
6:         else return ∞
7:     return mexp
```

This function computes an upper bound f' for each expression f which occurs inside a `nat` function and then replaces in `exp` all such f expressions with their corresponding upper bounds (line 5). If it cannot find an upper bound, the method returns ∞ (line 6).

Example 7 Applying *ub_exp* to the cost expressions $\mathtt{exp}_3$ and $\mathtt{exp}_4$, that appear in Equations (3) and (4) in Figure 5, w.r.t. the invariant that we have computed in Section 6.1, can be done by maximizing their `nat` sub-expressions. Similarly to what we have done above for $la - j - 1$, we can find upper bounds for $lb - j - 1$, la and lb as $lb_0 - 1$, la_0 and lb_0 respectively. Therefore, the expressions

$$\mathtt{mexp}_3 = 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)$$
$$\mathtt{mexp}_4 = 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0)$$

are upper bounds for $\mathtt{exp}_3$ and $\mathtt{exp}_4$. □

The lemma below guarantees the soundness of the function *ub_exp*.

Lemma 3 (soundness of *ub_exp***)** *Let* $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle$ *be a cost equation for* C, $\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$ *be a safe approximation of the loop invariant* I_C, *and* $\mathtt{mexp} = \mathit{ub\_exp}(\mathtt{exp}, \bar{x}_0, \varphi, \psi)$. *Then, for any call* $C(\bar{v})$ *and for all* $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$, *if* $\mathit{node}(C(\bar{w}), r, \_) \in T$ *such that* r *originates from* `exp`, *then* $[\![\mathtt{mexp}]\!]_\sigma \geq r$ *where* σ *is a substitution that maps* $\bar{x}_0$ *to* $\bar{v}$.

Proof The Lemma is trivially correct when $\mathtt{mexp} = \infty$. For $\mathtt{mexp} \neq \infty$, given $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$ and $\mathit{node}(C(\bar{w}), r, \_) \in T$, by Lemma 2, there exists a substitution σ, over $\bar{x}_0$ and the variables of the equation, such that $\sigma \models \bar{x}_0 = \bar{v} \wedge \bar{x} = \bar{w} \wedge \psi \wedge \varphi$ and $r = [\![\mathtt{exp}]\!]_\sigma$. Let $\mathtt{exp}'$ be a cost expression obtained from `exp` by replacing only one $\mathsf{nat}(f)$ by $\mathsf{nat}(f')$ (lines 4 and 5 in function *ub_exp*). Proposition 1 and the fact that $\psi \wedge \varphi \models f \leq f'$ implies $[\![\mathtt{exp}']\!]_\sigma \geq [\![\mathtt{exp}]\!]_\sigma$. Since `mexp` is obtained by repeating such replacement for all `nat` components, at the end we will have $[\![\mathtt{mexp}]\!]_\sigma \geq [\![\mathtt{exp}]\!]_\sigma = r$. □

The following lemma is a completeness lemma for function *ub_exp*, in the sense that if ψ and φ imply that there is f' which is an upper bound for f, then by syntactically looking on ϕ (line 4 of *ub_exp*) we will be able to find one, without guarantees that it will be the tightest one.

Lemma 4 (completeness of *ub_exp***)** *Consider line 5 of ub_exp, if there exists f' such that $\phi \models y \leq f'$ and $\phi = \{c_1, \ldots, c_n\}$, then there exists c_i which can be worked out to $y \leq f''$ (or $y = f''$) where $vars(f'') \subseteq \bar{x}_0$.*

Proof The lemma follows from: (1) if there exists f' such that $vars(f') \subseteq \bar{x}_0$ and $\psi \wedge \varphi \models f \leq f'$ then, $\phi \models y \leq f'$, since $y = f$ and $y \notin vars(\psi \wedge \varphi)$; (2) if $\phi \models y \leq f'$ and $vars(\phi) \subseteq \bar{x}_0 \cup \{y\}$, then y must appear in one of the c_i, which obviously can be worked out to $y \leq f'$; and (3) if there is more than one c_i where y appears, then taking one is safe as they appear in a conjunction. □

6.3 Concluding Remarks

Using Lemmata 2 and 3, the theorem below concludes by building the upper bound expression $costnr_+(\bar{x}_0)$ and $costr_+(\bar{x}_0)$.

Theorem 1 *Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ be a cost relation where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively the sets of non-recursive and recursive equations for C. Let*

- *$\langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \psi \rangle$ be a safe approximation of the loop invariant I_C;*
- *$E_i = \{ub\_exp(\texttt{exp}, \bar{x}_0, \varphi, \psi) \mid \langle C(\bar{x}) = \texttt{exp} + \sum_{j=1}^{k} C(\bar{y}_j), \varphi \rangle \in \mathcal{S}_i\}$, $1 \leq i \leq 2$; and*
- *$costnr_+(\bar{x}_0) = \max(E_1)$ and $costr_+(\bar{x}_0) = \max(E_2)$.*

Then, for any call $C(\bar{v})$ and for all $T \in Trees(C(\bar{v}), \mathcal{S})$, it holds that

- *$\forall node(\_, r, \_) \in internal(T)$. $costr_+(\bar{v}) \geq r$; and*
- *$\forall node(\_, r, \_) \in leaf(T)$. $costnr_+(\bar{v}) \geq r$.*

Proof Follows from Lemmata 2 and 3. □

Example 8 At this point we have all the pieces in order to compute an upper bound, as described in Proposition 2, for the *CR* depicted in Figure 3. We start by computing upper bounds for E and D as they are standalone cost relations:

	h_+	$costnr_+$	$costr_+$	Upper Bound
$E(la_0, j_0)$	$\texttt{nat}(la_0 - j_0 - 1)$	5	15	$5 + 15 * \texttt{nat}(la_0 - j_0 - 1)$
$D(a_0, la_0, i_0)$	$\texttt{nat}(la_0 - i_0)$	8	10	$8 + 10 * \texttt{nat}(la_0 - i_0)$

These upper bounds can then be substituted in the Equations (3) and (4) which results in the cost relation for C depicted in Figure 5. We have already computed a ranking function for C in Example 4, and $costnr_+$ and $costr_+$ in Example 7, which are then combined into:

$$C_+(l_0, a_0, la_0, b_0, lb_0) = 2 + \texttt{nat}(l_0) * max(\{\texttt{mexp}_3, \texttt{mexp}_4\})$$

By reasoning similarly, we obtain the upper bound for *Delete* shown in Table 1. □

7 Improving Accuracy in Divide and Conquer Programs

We have presented in Section 4 an approximation approach based on bounding both the number of nodes in evaluation trees and the cost per node which is able to provide upper bounds for a large class of programs, and it is not limited to any complexity class. However, there is an important class of programs known as *divide and conquer* for which the node-count upper-bound does not compute sufficiently precise upper bounds. Intuitively, the reason for this is that divide and conquer programs have a branching factor greater than one. Therefore, the number of nodes grows exponentially with the height of the evaluation tree. However, the size of the input data decreases so quickly from one level of the tree to the next one that the *sum* of the local cost expressions in the nodes at each level does not increase from one level to another.

In this section we propose an approximation mechanism, which we refer to as *level-count upper-bound* which is based on bounding both the number of levels in evaluation trees and the total cost per level. It allows obtaining accurate upper bounds for divide and conquer programs.

7.1 Level-count upper-bound

Given an evaluation tree T, we denote by $\mathsf{Sum\_Level}(T, i)$ the sum of the local cost of all nodes in T which are at depth i, i.e., at distance i from the root. As before, we write $h(T)$ to denote the height of T.

Proposition 3 (level-count upper-bound) *Let C be a cost relation. We define function C_+ as:*

$$C_+(\bar{x}) = l_+(\bar{x}) * costl_+(\bar{x})$$

where $l_+(\bar{x})$ and $costl_+(\bar{x})$ are closed-form functions defined on $\mathbb{Z}^n \mapsto \mathbb{R}_+$. Then, C_+ is an upper bound of C if for all $\bar{v} \in \mathbb{Z}^n$ and $T \in Trees(C(\bar{v}), \mathcal{S})$, it holds:

1. $l_+(\bar{v}) \geq h(T) + 1$*; and*
2. $\forall\ 0 \leq i \leq h(T)\ .\ costl_+(\bar{v}) \geq \mathsf{Sum\_Level}(T, i)$.

Proof The proposition is trivially correct by the definition of upper bound and *Answers*. □

Similarly to what we have done for $h_+(\bar{x})$ in Section 5, the function $l_+(\bar{x})$ can simply be defined as $l_+(\bar{x}) = \mathsf{nat}(f_C(\bar{x})) + 1$. Finding an accurate $costl_+$ function is not easy in general, which makes Proposition 3 not as widely applicable as Proposition 2.

7.2 Divide and Conquer Programs

We now provide a formal definition of *divide and conquer* programs and show that for all programs which fall into this class it is possible to apply the level-count upper-bound approach. Intuitively, a program belongs to the divide and conquer class when the local cost of each node in the evaluation tree is guaranteed to be greater than or equal to the sum of the local costs of its children. As we will see, this guarantees that $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k + 1)$. In that case, we can simply take the local cost of the root node as an upper bound of $costl_+(\bar{x})$.

Often we have multiple recursive and non-recursive equations for a cost relation. Checking that the local cost of a node is greater than the sum of those of its children needs to take into account all possible combinations of cost expressions produced by picking a recursive equation followed by picking any equation –be it recursive or not– for each recursive call in such equation. We now define the set of *child local-cost expressions* as a set of triplets composed by two cost expressions linked by a set of constraints which are all those achievable in the combinations explained.

Definition 8 (Child local-cost expressions) The set of child local-cost expressions of a standalone cost relation C, denoted $Child\_Exps(C)$, is defined as

$$Child\_Exps(C) = \left\{ \langle \mathtt{exp}, \mathtt{exp}', \psi \rangle \;\middle|\; \begin{array}{l} \langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle \in \mathcal{S}, \text{ where } k \geq 1 \\ \forall\ 1 \leq i \leq k.\ \langle C(\bar{y}_i) = \mathtt{exp}_i + \sum_{j=1}^{k_i} C(\bar{z}_j), \varphi_i \rangle \in \mathcal{S} \\ \mathtt{exp}' = \mathtt{exp}_1 + \cdots + \mathtt{exp}_k \\ \psi = \exists vars(\mathtt{exp}) \cup vars(\mathtt{exp}').\varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_k \end{array} \right\}$$

Example 9 Consider a *CR* in which C is defined by the two equations:

$$\langle C(x) = 0, \{x \leq 0\} \rangle$$
$$\langle C(x) = \mathsf{nat}(x) + C(x_1) + C(x_2), \varphi \rangle$$

where $\varphi = \{x > 0, x_1 + x_2 + 1 \leq x, x \geq 2 * x_1, x \geq 2 * x_2, x_1 \geq 0, x_2 \geq 0\}$. It corresponds to a divide and conquer problem such as merge-sort when the cost model used counts the number of comparison instructions executed, which is a usual criteria for comparing sorting programs and algorithms. The set $Child\_Exps(C)$ consists of:

$$Child\_Exps(C) = \left\{ \begin{array}{l} \langle \mathsf{nat}(x), 0, \varphi \wedge x_1 \leq 0 \wedge x_2 \leq 0 \rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_1), \varphi \wedge x_1 \leq 0 \wedge \varphi_2 \rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_2), \varphi \wedge \varphi_1 \wedge x_2 \leq 0 \rangle \\ \langle \mathsf{nat}(x), \mathsf{nat}(x_1) + \mathsf{nat}(x_2), \varphi \wedge \varphi_1 \wedge \varphi_2 \rangle \end{array} \right\}$$

where φ_1 (resp. φ_2) is a renaming apart of φ, except for the variable x_1 (resp. x_2). □

The following lemma provides a sufficient condition for a cost relation falling into the divide and conquer class, i.e., for Proposition 3 to be applicable. It is based on checking that each cost expression contributed by an equation is greater than or equal to the sum of the cost expressions contributed by the corresponding immediate recursive calls.

Lemma 5 (A sufficient condition for divide and conquer) *Let C be a standalone cost relation. If for any $\langle \mathtt{exp}, \mathtt{exp}', \psi \rangle \in Child\_Exps(C)$ and any $\sigma : vars(\mathtt{exp}) \cup vars(\mathtt{exp}') \mapsto \mathbb{Z}$ such that $\sigma \models \psi$ it holds that $[\![\mathtt{exp}]\!]_\sigma \geq [\![\mathtt{exp}']\!]_\sigma$, then for any call $C(\bar{v})$, a corresponding evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$, and a level k, it holds that $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$.*

Proof Assume the contrary, i.e., the condition holds but there exists a call $C(\bar{v})$, a corresponding evaluation tree $T \in Trees(C(\bar{v}), \mathcal{S})$, and a level k, such that $\mathsf{Sum\_Level}(T, k) < \mathsf{Sum\_Level}(T, k+1)$. This means that there exists a node $node(C(\bar{v}), r, \langle T_1, \ldots, T_n \rangle)$ at level k, such that for each subtree $T_i = node(C(\bar{v}_i), r_i, \_))$ it holds $r < r_1 + \cdots + r_n$. Assume this node was constructed using an equation $\mathcal{E} = \langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{m} C(\bar{y}_i), \varphi \rangle \in \mathcal{S}$ and that $\langle C(\bar{y}_i) = \mathtt{exp}_i + \sum_{j=1}^{m_i} C(\bar{z}_j), \varphi_i \rangle \in \mathcal{S}$ was used to match each call $C_i(\bar{y}_i)$ in $\mathcal{E}$. Then, there exists σ verifying $\sigma \models \varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_m \models \bar{x} = \bar{v} \wedge \bar{y}_1 = \bar{v}_1 \wedge \cdots \wedge \bar{y}_m = \bar{v}_m$, such that $[\![\mathtt{exp}]\!]_\sigma < [\![\mathtt{exp}_1 + \cdots + \mathtt{exp}_m]\!]_\sigma$, which contradicts the assumption that the condition holds. □

The intuition of the above lemma is that for each node in any evaluation tree, there exists a tuple $\langle \mathtt{exp}, \mathtt{exp}', \psi \rangle \in \mathit{Child\_Exps}(C)$ and a substitution $\sigma : \mathit{vars}(\mathtt{exp}) \cup \mathit{vars}(\mathtt{exp}') \mapsto \mathbb{Z}$ such that $\sigma \models \psi$, $[\![\mathtt{exp}]\!]_\sigma$ is equal to its local cost, and $[\![\mathtt{exp}']\!]_\sigma$ is equal to the sum of its children local costs.

Theorem 2 *Let C be a standalone cost relation which satisfies the divide and conquer condition of Lemma 5, $E = \{ub\_exp(\mathtt{exp}, \bar{x}_0, \varphi, \{\bar{x}_0 = \bar{x}\}) \mid \langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\bar{y}_i), \varphi \rangle \in \mathcal{S}\}$, and $costl_+(\bar{x}) = \max(E)$. Then, for any call $C(\bar{v})$, a corresponding evaluation tree $T \in \mathit{Trees}(C(\bar{v}), \mathcal{S})$, and a level k, it holds that $costl_+(\bar{v}) \geq \mathsf{Sum\_Level}(T, k)$.*

Proof It follows from Lemmata 3 and 5. □

Example 10 Consider again the cost relation C defined in Example 9. Computing the set E of Theorem 2 results in $\{\mathsf{nat}(x), 0\}$, and therefore $costl_+(x) = \mathsf{nat}(x)$. Using the techniques described in Section 5 we can automatically compute

$$l_+(x) = \lceil \log_2(\mathsf{nat}(x)+1) \rceil + 1$$

Thus, we obtain the upper bound $C_+(x) = \mathsf{nat}(x) * (\lceil \log_2(\mathsf{nat}(x) + 1) \rceil + 1)$. Note that this upper bound is inferred in a fully automatic way by our prototype which is described in Section 10. By using the node-count approach, we would obtain $C_+(x) = \mathsf{nat}(x) * (2^{\lceil \log_2(\mathsf{nat}(x)+1) \rceil} - 1) = \mathsf{nat}(x)^2$ as upper bound. □

8 Direct Recursion using Partial Evaluation

Our approach requires that all recursions be direct. However, automatically generated *CRSs* often contain recursions which are not direct, i.e., cycles involve more than one function.

Example 11 The cost analyzer of [4,6], in order to define the cost of the "for" loop in the program in Figure 1, instead of Equations (8) and (9) (relation E) in Figure 3, produces the following equations:

$$\begin{array}{lll}
(8') & E(la, j) = 5 + F(la, j, j', la') & \{j' = j, la' = la - 1, j' \geq 0\} \\
(9') & F(la, j, j', la') = H(j', la') & \{j' \geq la'\} \\
(10) & F(la, j, j', la') = G(la, j, j', la') & \{j' < la'\} \\
(11) & H(j', la') = 0 & \\
(12) & G(la, j, j', la') = 10 + E(la, j + 1) & \{j < la - 1, j \geq 0, la - la' = 1, j' = j\}
\end{array}$$

The new E relation captures the cost of evaluating the loop condition "$j < la - 1$" (5 cost units) plus the cost of its continuation, captured by F. In Equation (9') the relation F corresponds to the exit of the loop (it calls the auxiliary relation H, which represents the cost of exiting the loop, i.e., 0 units). Equation (10) captures the cost of one iteration, which accumulates 10 cost units and calls E recursively. □

In this section, we present an automatic transformation of *CRSs* into *directly recursive* form. The transformation is done by replacing calls to intermediate relations by their definitions using *unfolding*. For instance, given the *CRS* in Example 11, if we keep E and unfold the remaining relations in the example (F, G, and H), we obtain the equations for E shown in Figure 3.

8.1 Binding Time Classification

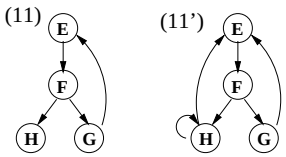
We now recall some standard terminology on graphs. A *directed graph* G is a pair $\langle N, A\rangle$ where N is the set of *nodes* and $A \subseteq N \times N$ is the set of *arcs*. Given a graph $G = \langle N, A\rangle$, a set of nodes $S \subseteq N$ is *strongly connected* if $\forall n, n' \in S$ we have that n' is reachable from n. The *strongly connected components* of $G = \langle N, A\rangle$ is a partition of N into the largest possible strongly connected sets. Given a graph G we write $SCC(G)$ to denote its strongly connected components. Given a graph $G = \langle N, A\rangle$ and a set $S \subseteq N$, the *subgraph* of G w.r.t. S, denoted $G|_S$, is defined as $G|_S = \langle S, A \cap (S \times S)\rangle$. Also, given a strongly connected component S, a node $n \in S$ is a *covering point* for $G|_S$ if $G|_{S\setminus\{n\}}$ is an acyclic graph, i.e., n is a covering point of $G|_S$ if n is part of all cycles in $G|_S$. The problem of finding a minimal set of nodes to delete from a cyclic graph in order to convert it into an acyclic graph is also known as the *feedback vertex set* problem in computational complexity theory. The *feedback vertex set* decision problem is NP-complete in general, but for reducible graphs (which is the case of most control flow graphs coming from structured programming languages) is it linear [46]. Moreover, since our interest is only in checking if there exists a feedback set of size 1, when the graph is not reducible, we can solve it in quadratic time simply by removing a node n from $G|_S$ and checking if $G|_{S\setminus\{n\}}$ is acyclic.

The notion of *unfolding* corresponds to the intuition of replacing a call to a relation by the definition of the corresponding relation. Naturally, this process in the presence of recursive relations might be non-terminating. Intuitively, the transformation proposed removes intermediate relations from the *CRS* and we achieve direct recursion if at most one relation remains per each strongly connected component in the call graph of the original *CRS*. In this section, we find a *Binding Time Classification* (or BTC for short) which ensures the termination of the unfolding process by declaring which relations are *residual*, i.e., they have to remain in the *CRS*. The remaining relations are considered *unfoldable*, i.e., they are eliminated. To define such BTC, we associate a *call graph* to each *CRS* $\mathcal{S}$ as follows. Given a *CRS* $\mathcal{S}$ with C, $D \in rel(\mathcal{S})$, we say that C *calls* D in $\mathcal{S}$, denoted $C \mapsto_{\mathcal{S}} D$, iff there is an equation $\langle C(\bar{x}) = \mathtt{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi\rangle \in \mathcal{S}$ such that $D_i = D$ for some $i \in \{1, \ldots, k\}$. The call graph associated to $\mathcal{S}$, denoted $\mathcal{G}(\mathcal{S})$, is the directed graph obtained from $\mathcal{S}$ by taking $N = rel(\mathcal{S})$ and where $(C, D) \in A$ iff $C \mapsto_{\mathcal{S}} D$. We now present sufficient conditions under which *CRSs* can be put into directly recursive form. In particular, we require that the graph associated to the *CRS* be of *minimal coverage*.

Definition 9 (minimal coverage) A graph $G = \langle N, A\rangle$ is *of minimal coverage* iff $\forall S \in SCC(G)$, there exists $n \in S$ such that n is a covering point for $G|_S$.

Intuitively, a graph is of minimal coverage if each SCC has a *covering point*. Let us see some examples.

Example 12 Given the *CRS* $\mathcal{S}$ of Example 11, its call graph $\mathcal{G}(\mathcal{S})$ is shown on the left hand side of the figure below. Also, we have that $SCC(\mathcal{G}(\mathcal{S})) = \{\{E, F, G\}, \{H\}\}$.



The strongly connected component which could be problematic as regards minimal coverage (more than one element) is $\{E, F, G\}$. Since there is just one cycle, any of the nodes is a covering point and therefore G is of minimal coverage. However, if we replace Equation (11) in Example 11 with Equation (11') below:

$$(11') \quad \langle H(j', la') \leftarrow 1 + H(j'', la') + E(j'', la'), \{j'' = j' - 1\} \rangle$$

we obtain the graph to the right of the figure. Now, $SCC(\mathcal{G}(\mathcal{S})) = \{\{E, F, G, H\}\}$, i.e., all nodes are in the same strongly connected component, and we have three cycles ($\langle E, F, G\rangle$, $\langle E, F, H\rangle$, and $\langle H\rangle$) which belong to such strongly connected component. Unfortunately, this time there is no node which belongs simultaneously to the three cycles. □

As shown in the example above, there are graphs which are not of minimal coverage. Therefore, there are *CRSs* which cannot be put into canonical form. However, structured loops (built using for, while, etc.) and the recursive patterns found in most programs naturally result in *CRSs* whose reachability graphs are of minimal coverage.

We can now define the notion of *directly recursive* BTC which ensures both the termination of our partial evaluation process and the effectiveness of the transformation (i.e., we indeed obtain direct recursion form). Formally, a relation D is *reachable* from a relation C in $\mathcal{S}$ iff there is a path from C to D in $\mathcal{G}(\mathcal{S})$. A relation C is *recursive* iff C is reachable from itself. It is *directly recursive* if $(C \mapsto_{\mathcal{S}} D \wedge D \neq C) \Rightarrow C$ is not reachable from D in $\mathcal{S}$, i.e., there cannot be cycles in the reachability relation (recursion) of length greater than one.

Definition 10 (directly recursive BTC) Given a *CRS* $\mathcal{S}$ with graph G, a BTC btc for $\mathcal{S}$ is *directly recursive* if for all $S \in SCC(G)$ the following two conditions hold:

(DR) if $s_1, s_2 \in S$ and $s_1, s_2 \in$ btc, then $s_1 = s_2$.
(TR) if S has a cycle, then there exists $s \in S$ such that $s \in$ btc.

Condition (**DR**) ensures that all recursions in the transformed *CRS* are direct, as there is only one residual relation per SCC. Condition (**TR**) guarantees that the unfolding process terminates, as there is a residual relation per cycle.

A directly recursive BTC for Example 11 is btc $= \{E\}$. In our implementation we include in BTCs only the covering point of SCCs which contain cycles, but not that of components without cycles. This way of computing BTCs, in addition to ensuring direct recursion, also eliminates all intermediate cost relations which are not part of cycles. Coming back to Example 11, our implementation computes btc $= \{E\}$. This is why the *CRS* shown in Figure 3 does not include equations for H.

8.2 Partial Evaluation of Cost Relations

We now present a *Partial Evaluation* [30] (PE for short) algorithm for transforming *CRSs*. Unfolding, in this context, in addition to taking care of combining arithmetic expressions, also has to combine the linear constraints and to consider a BTC btc to control the transformation process. The next definition of unfolding, given a call to a relation, produces a specialization for such call by unfolding all calls to relations which are marked as *unfoldable* in btc.

Definition 11 (unfolding) Given a *CRS* $\mathcal{S}$, a call $C(\bar{x}_0)$ such that $C \in rel(\mathcal{S})$, a set of linear constraints $\varphi_{\bar{x}_0}$ over the variables $\bar{x}_0$, and a BTC btc for $\mathcal{S}$, a *specialization* $\langle E, \varphi\rangle$ is obtained by unfolding $C(\bar{x}_0)$ and $\varphi_{\bar{x}_0}$ in $\mathcal{S}$ w.r.t. btc, denoted $\mathsf{Unfold}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$, if one of the following conditions hold:

(res) $(C \in \mathsf{btc} \wedge \varphi \neq \mathsf{true}) \wedge \langle E, \varphi\rangle = \langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle$.
(unf) $(C \notin \mathsf{btc} \vee \varphi = \mathsf{true}) \wedge \langle E, \varphi\rangle = \langle (\mathsf{exp} + e_1 + \ldots + e_k), \varphi' \bigwedge_{i=1..k} \varphi_i\rangle$,

where we have that:

1. $\langle C(\bar{x}) = \mathsf{exp} + \sum_{i=1}^{k} D_i(\bar{y}_i), \varphi_C\rangle$ is a renamed apart equation in $\mathcal{S}$ such that φ' is satisfiable in $\mathbb{Z}$, where $\varphi' = \varphi_{\bar{x}_0} \wedge \varphi_C[\bar{x}_0/\bar{x}]$.
2. $\mathsf{Unfold}(\langle D_i(\bar{y}_i), \varphi'\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle e_i, \varphi_i\rangle$ for all $i \in \{1, \ldots, k\}$.

The first case, (**res**), is required for termination. When we call a relation C which is marked as residual, we simply return the initial call $C(\bar{x}_0)$ and constraints $\varphi_{\bar{x}_0}$, as long as $\varphi_{\bar{x}_0}$ is not the initial one (true). The latter condition is added in order to enforce the initial unfolding step for relations marked as residual. In all subsequent calls to Unfold different from the initial one, the constraints are different from true. The second case (**unf**) corresponds to continuing the unfolding process. Step 1 is non deterministic in general, since cost relations are often defined by means of several equations. Furthermore, since expressions are transitively unfolded, step 2 may also provide multiple solutions. As a result, unfolding may produce multiple outputs. Also, note that the final constraint φ can be unsatisfiable. In such case, we simply do not regard $\langle E, \varphi\rangle$ as a valid unfolding. In the following, we denote by $\overset{unf}{=}_e$ an "unfolding step" performed by **unf** where an equation e is selected to replace a function call by its right hand side.

Example 13 Given the initial call $\langle E(la, j), true\rangle$, we obtain an unfolding by performing the following steps.

$$\begin{array}{ll}
\langle E(la, j), true\rangle & \overset{unf}{=}_{(8')} \\
\langle 5 + F(la, j, j', la'), \{j' = j, la' = la - 1, j' \geq 0\}\rangle & \overset{unf}{=}_{(10)} \\
\langle 5 + G(la, j, j', la'), \{j' = j, la' = la - 1, j' \geq 0, j' < la'\}\rangle & \overset{unf}{=}_{(12)} \\
\langle 15 + E(la, j''), \{j < la - 1, j \geq 0\}\rangle &
\end{array}$$

The last call $E(la, j'')$ cannot be further unfolded because the relation belongs to btc and $\varphi \neq true$. □

In the above definition, from each result of unfolding, we can build a *residual* equation. Given $\mathsf{Unfold}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$, its corresponding residual equation is $\langle C(\bar{x}_0) = E, \varphi\rangle$. We use $\mathsf{Residuals}(\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle, \mathcal{S}, \mathsf{btc})$ to denote the set of residual equations for $\langle C(\bar{x}_0), \varphi_{\bar{x}_0}\rangle$ in $\mathcal{S}$ w.r.t. φ. Now, we obtain a *partial evaluation* of C by collecting all residual equations for the call $\langle C(\bar{x}_0), \mathsf{true}\rangle$ where $\bar{x}_0$ are distinct variables.

Definition 12 (partial evaluation) Given a *CRS* $\mathcal{S}$, a relation C, and a BTC btc for $\mathcal{S}$, the *partial evaluation* for C in $\mathcal{S}$ w.r.t. btc is defined as:

$$\bigcup_{D \in \mathsf{btc} \cup \{C\}} \mathsf{Residuals}(\langle D(\bar{x}_0), \mathsf{true}\rangle, \mathcal{S}, \mathsf{btc})$$

The above definition provides an algorithm for partial evaluation of *CRSs*. In terms of PE [30], the algorithm we propose is an *off-line* PE which at the *global control* level is monovariant, since the initial constraint is true for all residual relations, and at the *local-control* it unfolds all calls to unfoldable relations and residualizes all calls to residual relations. Note that, in addition to the relations in btc, we also generate equations for the initial relation C.

Example 14 The partial evaluation of the equations of Example 11 w.r.t. the call of Example 13 are Equations (8) and (9) of Figure 3. Equation (9) is obtained from the unfolding steps depicted in Example 13 and Equation (8) from an unfolding derivation where the selected equations are (8'), then (9') and finally (11). As expected, the resulting *CRS* is directly recursive. □

The lemma below shows that partial evaluation is an effective way of obtaining direct recursion. It easily follows by the definition of BTC.

Lemma 6 *Let $\mathcal{S}$ be a CRS of minimal coverage and let C be a relation. Let* btc *be a directly recursive BTC for $\mathcal{S}$. Then,*

1. *partial evaluation for C in $\mathcal{S}$ w.r.t.* btc *produces a CRS $\mathcal{S}$' which is directly recursive and,*
2. *$\mathcal{S}'$ is obtained in finite time.*

Proof The proof is by contradiction. Let us first prove claim 1. Assume that we have a relation in $\mathcal{S}$' which is not directly recursive. This means that we can have equations of the form: $\langle C(\bar{x}) = \texttt{exp}{+}D(\bar{y}), \varphi_C\rangle$ and $\langle D(\bar{x}) = \texttt{exp}{+}C(\bar{y}), \varphi_D\rangle$ with $D \neq C$. As D has not been unfolded, then it must happen that $D \in$ btc. We have that C is in the same *SCC* as D. Then, by condition (**DR**) of Definition 10, it must happen that $C = D$. This contradicts the initial assumption. Claim 2 follows from the condition (**TR**) of Definition 10 by reasoning by contradiction. Let us assume that $\mathcal{S}$' is not obtained in finite time. This can only happen because Unfold does not terminate. Hence, there exists an infinite derivation $\langle E_1, \varphi_1\rangle \stackrel{unf}{=} \langle E_2, \varphi_2\rangle \stackrel{unf}{=} \ldots \stackrel{unf}{=} \langle E_n, \varphi_n\rangle \stackrel{unf}{=} \langle E_{n+1}, \varphi_{n+1}\rangle \stackrel{unf}{=} \ldots$. Since the number of cost relations in $rel(\mathcal{S})$ is finite and the sequence is infinite, there is a cycle from some E_i to an E_n for $i < n$. By condition (**TR**), this cannot happen because there must exist an E_j in the cycle with $i \leq j \leq n$ that belongs to btc. □

The following lemma guarantees that PE preserves the solutions of *CRSs*. The proof basically consists in ensuring the correctness of the basic operators in the partial evaluation algorithm of Definition 12 to, then, rely on the classical correctness results of PE proven in the context of logic programming (see e.g. [36,31,30] and more recent formulations like [35,34]).

Lemma 7 (correctness of PE) *Let $\mathcal{S}$ be a CRS, C be a relation, and let* btc *be a BTC for $\mathcal{S}$. Let $\mathcal{S}'$ be the partial evaluation of C in $\mathcal{S}$ w.r.t.* btc*. Then, $\forall \bar{v} \in \mathbb{Z}^n, \forall r \in \mathbb{R}_+$ we have that $r \in Answers(C(\bar{v}), \mathcal{S})$ iff $r \in Answers(C(\bar{v}), \mathcal{S}')$.*

Proof (sketch) The proof can be done by demonstrating that Definition 12 is a *correct* partial evaluation as defined in logic programming. Correctness results were already stated in Theorem 1 of [31] and more recent formulations appear in [35,34]. In all cases, correctness requires proving that:

1. *Soundness.* The soundness condition ensures that the all answers in the partially evaluated program are also answers in the original program. It is proven by demonstrating that each unfolding step in the partially evaluated program corresponds to a sequence of equivalent steps in the original one. In our context, it amounts to ensuring that the operator Unfold of Definition 11 preserves the answers.
2. *Completeness.* Completeness guarantees that all answers in the original program are also found in the partially evaluated one. It can be ensured when the set of terms to be partially evaluated meets the so-called *closedness* condition [36]. The role of this condition is to ensure that all possible calls that raise during the execution of a *CRS* will find a matching relation. In our context, we need to ensure that the set btc enforces the closedness condition, i.e., answers are not lost.

Point 1 requires to prove the correctness of operator Unfold of Definition 11. It indeed trivially holds as Unfold simply replaces in rule **(unf)** a function call by its right hand side, with the corresponding propagation of constraints. In terms of evaluation trees, this step basically merges a node with (some of) its successors.

The closedness of the terms to be partially evaluated, i.e., the elements in the set btc, follows from the fact that only terms in btc remain in the relation and the remaining ones are unfolded. This trivially ensures that all possible calls during execution will be covered by btc, as required by point 2 above. In standard PE, correctness requires that the partial evaluation process terminates. This is ensured by Lemma 6. □

9 Incompleteness in Cost Analysis

When we consider the whole cost analysis which comprises the two phases mentioned in Section 1, i.e., obtaining a closed-form upper-bound from a program —instead of from a *CRS*— the problem is strictly more difficult than proving termination. This is explained by the fact that obtaining a closed-form upper-bound of a program which has a non-zero cost expression associated to each recursive equation implies the termination of the program from which the *CR* has been generated. Therefore, the approach is necessarily incomplete and might fail to produce an upper-bound. Clearly, this may occur because the resource usage of the program is actually infinite w.r.t. the cost model used. For instance, a non-terminating program that can perform an infinite number of steps. When the resource consumption is finite, we can still fail to produce an upper bound because of loss of precision in one of the two phases in the cost analysis. This can occur in the first phase, i.e., when the program is transformed into the *CRS* since it applies abstract interpretation based analyses in order to approximate undecidable problems such as aliasing and size relations. However, the incompleteness in the first part of the analysis is completely outside the scope of this paper and we refer to [4] for further details.

Certainly, the second part of cost analysis is undecidable as well, i.e., if a given cost relation admits a closed-form upper-bound, so we must accept certain restrictions. In [14], it is proven that a simpler problem, namely the termination of a special case of *CRS* where all equations have at most one call in the body and constraints are of the form $x - y \leq c$, is undecidable. A detailed discussion about decidability of simple loops with integer constraints can be found in [18]. There are three sources of incompleteness in our approach, i.e., in the process of obtaining an upper bound from a *CRS* by using our techniques.

1. The first one is obtaining directly recursive *CRs*. For instance, the following *CRS* does not have a cover point:

$$\langle C(n) = C(n') + D(n'), \{n > 0, n'{=}n{-}1\}\rangle$$
$$\langle D(n) = D(n') + C(n'), \{n > 0, n'{=}n{-}1\}\rangle$$

   Importantly, this phase is complete for *CRs* extracted from structured loops and from the recursive patterns found in most programs. The use of features like `break` and `continue` that languages like Java or C do not pose any problem, since the control flow graph of the program can be constructed and turned into recursive form. As it can be seen in the example, incompleteness might occur in certain types of mutually recursive relations.
2. The second source of incompleteness in our method is on finding ranking functions. Currently, we use a complete procedure for inferring linear ranking functions [41]. However, there are *CRSs* which do not have a linear ranking function like this one (borrowed from [41]):

$$\langle C(n) = 1 + C(n'), \{n{>=}0, n'{=}{-}2{*}n{+}10\}\rangle$$

   Integrating other more sophisticated ranking functions is possible, but it is probably not required in practice.
3. The third one is finding useful invariants. Sometimes this is not possible by using linear constraints. This happens for example in this example:

$$\langle C(n,m) = m, \{n{=}0\}\rangle$$
$$\langle C(n,m) = C(n',m'), \{n'{=}n{-}1, m'{=}2{*}m, n{>}0\}\rangle$$

   The value of m in the base case will be $(2^n) * m_0$. In principle, we could use methods for inferring polynomial invariants, although we would need a different maximization procedure.

10 Experimental Evaluation

We have implemented our proposal in Prolog, and we use PPL [12] (*Parma Polyhedra Library*) for manipulating linear constraints. This implemented system, called PUBS (*Practical Upper Bounds Solver*), can be executed on-line using a web interface available at `http://www.cliplab.org/Systems/PUBS`. In order to test our system on realistic *CRs* obtained by automatic cost analysis, we have used as input a set of *CRs* automatically generated by the cost analyzer of Java bytecode described in [4] from a series of Java programs which represent classical examples in complexity analysis. The source code of such programs is also available at the above `url`. All benchmarks are presented in increasing complexity order and, as can be seen, the benchmarks range from constant to exponential complexity, ranging through polynomial and divide and conquer. The benchmark MergeSort falls into the class of divide-and-conquer programs explained in Section 7 where, by using the level-count approach, we obtain the accurate closed-form shown in the table. In addition, in the experiments, we have used three different cost models:

– the heap consumption (in bytes), in those benchmarks marked with "*", and
– the number of executed comparison instructions, in the benchmark marked with "n",

Benchmark	Properties	Upper Bound
Polynomial*	a,b,c	216
DivByTwo	a,b	$8\mathsf{log}_2(\mathsf{nat}(2x{-}1){+}1){+}14$
ArrayReverse$^{\mathsf{M}}$	a	$14\mathsf{nat}(x){+}12$
Concat$^{\mathsf{M}}$	a,c	$11\mathsf{nat}(x){+}11\mathsf{nat}(y){+}25$
Incr$^{\mathsf{M}}$	a,c	$19\mathsf{nat}(x{+}1){+}9$
ListReverse$^{\mathsf{M}}$	a,b,c	$13\mathsf{nat}(x){+}8$
MergeList	a,b,c	$29\mathsf{nat}(x+y){+}26$
Power		$10\mathsf{nat}(x){+}4$
Cons*	a,b	$22\mathsf{nat}(x{-}1){+}24$
MergeSort^n	a,b,c	$2\mathsf{nat}(-x+y+1)(\mathsf{log}_2(\mathsf{nat}(-2x+2y-1)+1)+1)$
EvenDigits	a,b,c	$\mathsf{nat}(x)(8\mathsf{log}_2(\mathsf{nat}(2x{-}3)+1)+24)+9\mathsf{nat}(x)+9$
ListInter	a,b,c	$\mathsf{nat}(x)(10\mathsf{nat}(y)+43)+21$
SelectOrd	a,c	$\mathsf{nat}(x{-}2)(17\mathsf{nat}(x{-}2)+34)+9$
FactSum	a	$\mathsf{nat}(x{+}1)(9\mathsf{nat}(x)+16)+6$
Delete	a,b,c	$3+\mathsf{nat}(l)\max(38{+}15\mathsf{nat}(la{-}1){+}10\mathsf{nat}(la),$ $37{+}15\mathsf{nat}(lb{-}1){+}10\mathsf{nat}(lb))$
MatMult$^{\mathsf{M}}$	a,c	$\mathsf{nat}(y)(\mathsf{nat}(x)(27\mathsf{nat}(x))+10)+17$
Hanoi$^{\mathsf{M}}$		$20(2^{\mathsf{nat}(x)})$-17
Fibonacci$^{\mathsf{M}}$		$18(2^{\mathsf{nat}(x-1)})$-13
BST*	a,b	$96(2^{\mathsf{nat}(x)})$-49

Table 1 Upper bounds computed automatically

– the number of executed bytecode instructions, in the rest of benchmarks.

Table 1 shows the upper bounds obtained by our system. Those benchmarks marked with M were also solved using Mathematica® in [5] but this required significant human intervention as we explain below. The column **Properties** in Table 1 shows the properties of the corresponding *CR*, in such a way that *a*, *b* and *c* indicate, respectively, that the *CR* is non-deterministic, that it has inexact size constraints, and multiple arguments (Section 2.2). We argue that the obtained upper bounds are reasonably accurate and relatively syntactically simple, especially when compared to the exact closed-form solutions produced by CAS for those marked M. The only examples that can be directly solved in Mathematica® are Fibonacci and Hanoi. For instance, for Fibonacci, Mathematica® computes the following upper bound: $-(2^{3-x}(15^{1+x}-19(1-\sqrt{5})^x+5\sqrt{5}(1-\sqrt{5})^x-19(1+\sqrt{5})^x-5\sqrt{5}(1+\sqrt{5})^x))/((-1+\sqrt{5})^2(1+\sqrt{5})^2)$ which is rather complex (even if it is more accurate) than the one computed by PUBS for the same *CRS*. The fact that it is more complex makes it more difficult to use it for the applications discussed in Section 1.1. This is clearly the case of the use of upper bounds for performance debugging, as the process of checking the complexity order from such complex expression is less intuitive to the user and sometimes hardly doable. For Hanoi, both PUBS and Mathematica® obtain the same result, which is the one shown in the table.

The remaining examples marked with M require non-trivial manual transformations in order to make them solvable in Mathematica®. Concat performs the concatenation of two lists and its complexity is a function of the length of both lists. Mathematica® has several restrictions that the *CRS* does not satisfy and which prevent us from solving it, namely (1) we cannot include guards, (2) variables cannot be repeated in the equation head, (3) all equations must have at least one variable argument and (4) variables in the equation head must appear in the body. Still, we can manipulate the *CRS*, split its equations into the two parts which correspond to the two loops which perform the

Benchmark	$\#_{eq}$	T	$\#^{c}_{eq}$	T_{pe}	T_{ub}	Rat.
Polynomial	23(3)	10	385(97)	388	1190	4.1
DivByTwo	9(3)	2	362(94)	402	1173	4.3
ArrayReverse	9(3)	2	344(88)	387	1122	4.4
Concat	14(5)	10	335(85)	386	1102	4.4
Incr	28(5)	23	321(80)	384	1046	4.5
ListReverse	9(3)	4	293(75)	374	943	4.5
MergeList	21(4)	17	284(72)	374	925	4.6
Power	8(2)	2	262(67)	366	898	4.8
Cons	22(2)	6	253(64)	376	912	5.1
MergeSort	39(12)	499	230(61)	354	805	5.0
EvenDigits	18(5)	7	191(49)	130	290	2.2
ListInter	37(9)	48	173(44)	126	246	2.2
SelectOrd	19(6)	22	136(35)	115	169	2.1
FactSum	17(5)	8	117(29)	109	143	2.2
Delete	33(9)	106	100(24)	102	130	2.3
MatMult	19(7)	17	67(15)	69	34	1.5
Hanoi	9(2)	5	48(8)	67	16	1.7
Fibonacci	8(2)	4	39(6)	63	11	1.9
BST	31(4)	36	31(4)	64	8	2.3

Table 2 Scalability of upper bounds inference

concatenation, solve them separately and then compose their results. After this, the upper bounds computed by Mathematica® and PUBS are identical for this example. Something similar occurs in MatMult, whose complexity depends on the number of rows and columns of the matrixes to be multiplied. After doing the transformation, Mathematica® and PUBS compute exactly the same upper bound. For the case of Incr, the *CRS* contains non-deterministic equations originated from a virtual invocation with three different implementations. This requires to generate three corresponding *CRS* and solve them independently and then take the maximum. As it was noted in Section 2.2, this is not always possible to ensure a sound result.

Although it does not happen in the examples shown in the table, a possible source of inaccuracy in our approach occurs when we find examples like this one:

```
for (int i = 0; i<n; i++)
  for (int j=i; j<n; j++) C
```

PUBS gives an upper bound of the form $n*n*C$ where C represents the (upper bound) cost of each iteration of the innermost loop. It is well known that the exact solution would be $C*((n*(n-1))/2)$. This is because we provide as upper bound of the inner loop the worst case, which corresponds to $i = 0$ in order to plug it in the calling context from the outer loop.

Table 2 aims at studying the efficiency of our system by showing the results of two different experiments. In the first experiment, we analyze each of the benchmarks in isolation. Column $\#_{eq}$ shows the number of equations before PE (in brackets after PE). Note that PE greatly reduces $\#_{eq}$ in all benchmarks. Column **T** shows the total runtime in milliseconds. The experiments have been performed on an Intel Core 2 Quad Q9300 at 2.50GHz with 1.95GB of RAM, running Linux 2.6.24-21. We argue that analysis times are acceptable. In the case of MergeSort analysis time is higher because its equations contain a large number of variables when compared to those of

the other examples. This affects the efficiency when computing the ranking function and also when maximizing expressions.

The second experiment aims at studying how analysis time increases when larger *CRs* are used as benchmarks, i.e., the scalability of our approach. In order to do so, we have connected together the *CRs* for the different benchmarks by introducing a call from each *CR* to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. The results of this second experiment are shown in the last four columns of the table. Column $\#^c_{eq}$ shows the number of equations we want to solve in each case (in brackets after PE). Reading this column bottom-up, we can see that when we analyze BST in the second experiment we have the same number of equations as in the first experiment. Then, for Fibonacci we have its 8 equations plus 31 which have been previously accumulated. Progressively, each benchmark adds its own number of equations to $\#^c_{eq}$. Thus, in the first row we have a *CRS* with all the equations connected, i.e., we compute a closed-form upper-bound of a *CRS* with at least 20 nested loops and 385 equations. In this experiment, the analysis time is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and $\mathbf{T}_{ub}$ is the time of all other phases. The results show that even though PE is a *global* transformation, its time efficiency is linear with the number of equations, since PE operates on strongly connected components. Our system solves 385 equations in $388 + 1190ms$.

Finally, column **Rat**. shows the total time per equation. The ratio is quite small from BST to EvenDigits, which are the simplest benchmarks and also have few equations. It increases notably when we analyze the benchmark MergeSort because, as discussed above, its equations have a large number of variables. The important point is that for larger *CRs* (from MergeSort upwards) this ratio decreases more and more as we connect new benchmarks. It should be observed that it decreases even if the size of the *CRs* increases and also the equations have to count more complex expressions. This happens because the new benchmarks which are connected are simpler than MergeSort in terms of the number of variables. We believe that this demonstrates that our approach is scalable even if the implementation is preliminary. The upper bound expressions get considerably large when the benchmarks are composed together. We are currently implementing standard techniques for simplification of arithmetic expressions.

Pubs is already integrated within the COst and Termination Analyzer for Java bytecode, Costa [6]. If one wants to obtain closed-form upper bounds from the Java program rather than from the cost relations, the Costa system can be used online at: http://pargo.ls.fi.upm.es/costa.

11 Related Work

As already mentioned in Section 1, the classical approach to automatic cost analysis, which dates back to the seminal work of [50] consists of two phases. In the first phase, given a program and a cost model, static analysis produces what we call a *cost relation* (*CR*), which is a set of recursive equations which capture the cost of our program in terms of the size of its input data. The fact that *CRs* are recursive make them not very useful for most applications of cost analysis. Therefore, a second phase is required to obtain a non-recursive representation of such *CRs*, known as *closed-form*. In most cases, it is not possible to find an exact solution and the closed-form corresponds to an upper-bound.

There are a number of cost analyses available which are based on building *CRs* and which can handle a range of programming languages, including functional [50,33, 43,49,45,16,37], logic [25,39], and imperative [4]. Such *CRs* must ensure that, for any valid input integer tuple, a value which is guaranteed to be an upper bound of the execution cost of the program for any input data in the (usually infinite) set of values which are consistent with the input sizes. There is no unified terminology in this area and such cost relations are referred to as *worst-case complexity functions* in [1], as *time-bound functions* in [43], and *recursive time-complexity functions* in [33]. Apart from syntactic differences, the main differences between such forms of functions and our cost relations are twofold: (1) our equations contain associated size constraints and (2) we consider (possibly) non-deterministic relations. Both features are necessary to perform cost analysis of realistic languages (see Section 2.2). While in all such analyses the first phase, i.e., producing *CRs* is studied in detail, the second phase, i.e., obtaining closed-form upper-bounds for them, has received comparatively less attention.

There are two main ways of viewing *CRs* which lead to different mechanisms for finding closed-form upper-bounds. We call the first view *algebraic* and the second view *transformational*. The algebraic one is based on regarding *CRs* as *recurrence relations*. This view was the first one to be proposed and it is the one which is advocated for in a larger number of works. It allows reusing the large existing body of work in solving recurrence relations. Within this view, two alternatives have been used in previous analyzers. One alternative consists in implementing restricted recurrence solvers within the analyzer based on standard mathematical techniques, as done in [50,25]. The other alternative, motivated by the availability of powerful *computer algebra systems* (CASs for short) such as Mathematica®, MAXIMA, MAPLE, etc., consists in connecting the analyzer with an external solver, as proposed in [49,45,16,4,37].

The transformational view consists in regarding *CRs* as (functional) programs. In this view, closed-form upper-bounds are produced by applying (general-purpose) program transformation techniques on the *time-bound program* [43] until a non-recursive program is obtained. Note that, as discussed in Section 2, it is straightforward to obtain time-bound programs from *CRs* by introducing a maximization operator (or disjunctive execution). The transformational view was first proposed in the ACE system [33], which contained a large number of program transformation rules aimed at obtaining non-recursive representations. It was also advocated by Rosendahl in [43], who later in [44] provided a series of program transformation techniques based on super-compilation [48] which were able to obtain closed-forms for some classes of programs.

The problem with all the approaches mentioned above is that, though they can be successfully applied for obtaining closed-forms for *CRs* generated from simple programs, they do not fulfill the initial expectations in that they are not of general applicability to *CRs* generated from real programs. The essential features which neither the algebraic nor the transformational approaches can handle are discussed in Section 2.2. The main motivation for this work was our own experience in trying to apply the algebraic approach on the *CRs* generated by [4]. We argue that automatically converting *CRs* into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. In contrast, our approach can produce correct and comparatively simple results even in the presence of non-determinism.

The need for improved mechanisms for automatically obtaining closed-form upper-bounds was already pointed out in Hickey and Cohen [28]. A significant work in this

direction is PURRS [13], which has been the first system to provide, in a fully automatic way, non-asymptotic closed-form upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires *CRs* to be deterministic. Another relevant work is that of Marion et. al. [38,17], who propose an analysis for stack frame size in first order functional programming. They use quasi-interpretations, which are different from ranking functions and the whole approach is limited to polynomial bounds.

An altogether different approach to cost analysis is based on type systems with resource annotations, which does not use *CRs* as an intermediate step. Thus, this approach does not require computing closed-form upper-bounds for *CRs*, but it is often restricted to linear bounds [29], with some notable exception like [24].

A program analysis based approach for inferring *polynomial* boundedness of computed values (as a function of the input) has been recently proposed in [15]. It infers the complexity of a given program by first obtaining a step-counting program. This work builds on similar previous works along the lines of [40,32], and the main novelty here is that it provides completeness for a simple (Turing incomplete) language. Compared to this line of research, our approach is more powerful in that it is not limited to polynomial complexity but, on the other hand, the techniques we use are inherently incomplete.

12 Conclusions and Future Work

We have proposed an approach to the automatic inference of non-asymptotic closed-form upper-bounds of *CRs* produced by automatic cost analysis. For this, we have formally defined *CRs* as a target language for cost analysis. Hence, our method for closed-form upper-bound inference can be used in static cost analysis of any programming language. In spite of the inherent incompleteness, we have experimentally shown that our approach is able to obtain useful upper bounds for a large class of common programs. In summary, the use of ranking functions and our practical method to compute upper bounds for a very general notion of cost expression (including exponential, logarithmic, etc.) allows obtaining closed-form upper-bounds for realistic *CRs* with possibly non-deterministic equations, multiple arguments, and inexact size constraints.

In recent work [8], we have applied our method to obtain closed-form upper-bounds from non-standard *CRs*, namely from *CRs* which capture the *heap space usage* of programs by taking into account the deallocations performed by garbage collection, without requiring any change to the techniques presented in this paper.The way that cost relations are generated is different from the standard approach because the live heap space is not an accumulative resource of a program's execution but, instead, it requires to reason on all possible states to obtain their maximum. As a result, cost relations include non-deterministic equations which capture the different peak heap usages reached along the execution. Importantly, the additional non-determinism does not pose any problem to applying our method.

As future work, we plan to adapt our general framework to infer closed-form lower-bounds on the cost. While the minimization can be done similarly to the maximization method used for upper bounds inference, we will need different techniques to infer lower bounds on the number of iterations of loops. As another direction of future work, *Abstraction-Carrying Code* [9] (ACC) proposes the use of static analysis as enabling technology for certification. In particular, in ACC, the safety policies are defined over

different abstract domains. The main idea is that the *abstraction* (or abstract model) of the program computed by standard static analyzers is used as a certificate. Then, the validity of the abstraction on the consumer side is checked in a single pass by a very efficient and specialized abstract-interpreter. Within the ACC framework, the present work suggests that it is possible to automatically generate non-trivial resource usage bounds for a realistic programming language. In the future, we plan to integrate our work into a proof-carrying code infrastructure where our upper bounds should be translated to *certificates* which provide cost assurances on the mobile code.

Acknowledgements We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped to improve this article. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* ,TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Relation Systems: a Language–Independent Target Language for Cost Analysis. In *Spanish Conference on Programming and Computer Languages (PROLE'08)*, volume 17615 of *ENTCS*. Elsevier, October 2008. To appear.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
5. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 67–83. Elsevier - North Holland, July 2007.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.
7. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis of Java Bytecode. In *International Symposium on Memory Management (ISMM'07)*. ACM Press, 2007.
8. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.
9. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code: A Model for Mobile Code Safety. *New Generation Computing*, 26(2):171–204, March 2008.
10. Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.
11. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 1–27. Springer, 2005.

12. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
13. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, 2005. `arXiv:cs/0512056` available from http://arxiv.org/.
14. Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.
15. Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 2008.
16. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.
17. G. Bonfante, J-Y. Marion, and J-Y. Moyen. Quasi-interpretations and small space bounds. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2005.
18. M. Braverman. Termination of Integer Linear Programs. In *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2006.
19. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.
20. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
21. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM Press, 1978.
22. Stephen-John Craig and Michael Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.
23. K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.
24. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
25. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
26. G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.
27. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
28. T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35(1), 1988.
29. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.
30. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
31. J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.
32. Lars Kristiansen and Neil D. Jones. The flow of data and the complexity of algorithms. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *CiE*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2005.
33. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.
34. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):413 – 463, May 2004.

35. M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
36. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.
37. Beatrice Luca, Stefan Andrei, Hugh Anderson, and Siau-Cheng Khoo. Program transformation by solving recurrences. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 121–129, New York, NY, USA, 2006. ACM.
38. J-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2006.
39. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.
40. K-H. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.
41. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
42. G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 261–271. ACM Press, July 2006.
43. M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.
44. M. Rosendahl. Simple driving techniques. In T. Mogensen, D. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 404–419. Springer, 2002.
45. D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.
46. Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM J. Comput.*, 8(4):645–655, 1979.
47. Fausto Spoto, Patricia M. Hill, and Etienne Payet. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
48. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
49. P. Wadler. Strictness analysis aids time analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM Press, 1988.
50. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, we first produce *recurrence relations* (RRs) which capture the cost of our program in terms of the size of its input data. Second, we convert such RRs into *closed form* (i.e., without recurrences). Whereas the first phase has received considerable attention, with a number of cost analyses available for a variety of programming languages, the second phase has received comparatively little attention. In this paper we first study the features of RRs generated by automatic cost analysis and discuss why existing computer algebra systems are not appropriate for automatically obtaining closed form solutions nor upper bounds of them. Then we present, to our knowledge, the first practical framework for the fully automatic generation of reasonably accurate upper bounds of RRs originating from cost analysis of a wide range of programs. It is based on the inference of *ranking functions* and *loop invariants* and on *partial evaluation*.

1 Introduction

The aim of *cost analysis* is to obtain *static* information about the execution cost of programs w.r.t. some cost *measure*. Cost analysis has a large application field, which includes resource certification [11,4,16,9], whereby code consumers can reject code which is not guaranteed to run within the resources available. The resources considered include processor cycles, memory usage, or *billable events*, e.g., the number of text messages or bytes sent on a mobile network.

A well-known approach to automatic cost analysis, which dates back to the seminal work of [25], consists of two phases. In the first phase, given a program and some cost measure, we produce a set of equations which captures the cost of our program in terms of the size of its input data. Such equations are generated by converting the iteration constructs of the program (loops and recursion) into recurrences and by inferring *size relations* which approximate how the size of arguments varies. This set of equations can be regarded as *recurrence relations* (RRs for short). Equivalently, it can be regarded as *time bound programs* [22]. The aim of the second phase is to obtain a non-recursive representation of the equations, known as *closed form*. In most cases, it is not possible to find an exact solution and the closed form corresponds to an upper bound.

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 221–237, 2008.

There are a number of cost analyses available which are based on this approach and which can handle a range of programming languages, including functional [7,18,22,23,24,25], logic [12,20], and imperative [1,3]. While in all such analyses the first phase is studied in detail, the second phase has received comparatively less attention. Basically, there are three different approaches for the second phase. One approach, which is conceptually linked viewing equations as time bound programs, was proposed in [18] and advocated in [22]. It is based on existing source-to-source transformations which convert recursive programs into non-recursive ones. The second approach consists in building restricted recurrence solvers using standard mathematical techniques, as in [12,25]. The third approach consists in relying on existing *computer algebra systems* (CASs for short) such as Mathematica®, MAXIMA, MAPLE, etc., as in [3,7,23,24].

The problem with the three approaches above is that they assume a rather limited form of equations which does not cover the essential features of equations actually generated by automatic cost analysis. In the rest of the paper, we will concentrate on viewing equations as recurrence relations and will use the term *Cost Relation* (CR for short) to refer to the relations produced by automatic cost analysis. In our own experience with [3], we have detected that existing CASs are, in most cases, not capable of handling CRs. We argue that automatically converting CRs into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. An altogether different approach to cost analysis is based on type systems with resource annotations which does not use equations. Thus, it does not need to obtain closed forms, but it is typically restricted to linear bounds [16]. The need for improved mechanisms for obtaining upper bounds was already pointed out in Hickey and Cohen [14]. A relevant work in this direction is PURRS [5], which has been the first system to provide, in a fully automatic way, non-asymptotic upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires CRs to be deterministic. Marion et. al. [19,8] use a kind of polynomial ranking functions, but the approach is limited to polynomial bounds and can only handle a rather restricted form of CRs.

We believe that the lack of automatic tools for the above second phase is a major reason for the diminished use of automatic cost analysis. In this paper we study the features of CRs and discuss why existing CASs are not appropriate for automatically bounding them. Furthermore, we present, to our knowledge, the first practical framework for the fully automatic inference of reasonably accurate upper bounds for CRs originating from a wide range of programs. To do this, we apply semantic-based transformation and analysis techniques, including inference of *ranking functions*, *loop invariants* and the use of *partial evaluation*.

1.1 Motivating Example

Example 1. Consider the Java code in Fig. 1. It uses a class List for (non sorted) linked lists of integers. Method del receives an input list without repetitions l,

```
void del(List l, int p, int a[], int la, int b[], int lb){
  while (l!=null) {
    if (l.data<p) {
      la=rm_vec(l.data, a, la);
    } else {
      lb=rm_vec(l.data, b, lb);
    }
    l=l.next;
  }
}

int rm_vec(int e, int a[], int la){
  int i=0;
  while (i<la && a[i]<e)  i++;
  for (int j=i; j<la−1; j++) a[j]=a[j+1];
  return la−1;
}
```

(1) $Del(l, a, la, b, lb)=1+C(l, a, la, b, lb)$ $\{b\geq lb, lb\geq 0, a\geq la, la\geq 0, l\geq 0\}$
(2) $C(l, a, la, b, lb)=2$ $\{a\geq la, b\geq lb, b\geq 0, a\geq 0, l=0\}$
(3) $C(l, a, la, b, lb)=$ $25+D(a, la, 0)+E(la, j)+C(l', a, la-1, b, lb)$ $\{a\geq 0, a\geq la, b\geq lb, j\geq 0, b\geq 0, l>l', l>0\}$
(4) $C(l, a, la, b, lb)=$ $24+D(b, lb, 0)+E(lb, j)+C(l', a, la, b, lb-1)$ $\{b\geq 0, b\geq lb, a\geq la, j\geq 0, a\geq 0, l>l', l>0\}$
(5) $D(a, la, i)=3$ $\{i\geq la, a\geq la, i\geq 0\}$
(6) $D(a, la, i)=8$ $\{i<la, a\geq la, i\geq 0\}$
(7) $D(a, la, i)=10+D(a, la, i+1)$ $\{i<la, a\geq la, i\geq 0\}$
(8) $E(la, j)=5$ $\{j\geq la-1, j\geq 0\}$
(9) $E(la, j)=15+E(la, j+1)$ $\{j<la-1, j\geq 0\}$

Fig. 1. Java Code and the Result of Cost Analysis

an integer value p (the *pivot*), two sorted arrays of integers a and b, and two integers la and lb which indicate, respectively, the number of positions occupied in a and b. The array a (resp. b) is expected to contain values which are smaller than the pivot p (resp. greater or equal). Under the assumption that all values in l are contained in either a or b, the method del removes all values in l from the corresponding arrays. The auxiliary method rm_vec removes a given value e from an array a of length la and returns its new length, la−1.

We have applied the cost analysis in [3] on this program in order to approximate the cost of executing the method del in terms of the number of executed bytecode instructions. For this, we first compile the program to bytecode and then analyze the resulting bytecode. Fig. 1 (right) presents the results of analysis, after performing *partial evaluation*, as we will explain in Sec. 6, and inlining equality constraints (e.g., inlining equality $lb'=lb-1$ is done by replacing the occurrences of lb' by $lb-1$). In the analysis results, the data structures in the program are abstracted to their sizes: l represents the maximal *path-length* [15] of the corresponding dynamic structure, which in this case corresponds to the length of the list, a and b are the lengths of the corresponding arrays, and la and lb are the integer values of the corresponding variables. There are nine equations which define the relation Del, which corresponds to the cost of the method del, and three auxiliary recursive relations, C, D, and E. Each of them corresponds to a loop (C: while loop in del; D: while loop in rm_vec; and E: for loop in rm_vec). Each equation is annotated with a set of constraints which capture *size relations* between the values of variables in the left hand side (lhs) and those in the right hand side (rhs). In addition, size relations may contain applicability conditions (i.e., *guards*) by providing constraints which only affect variables in the lhs. Let us explain the equations for D. Eqs. (5) and (6) are base cases which correspond to the exits from the loop when i≥la and a[i]≥e, respectively. Note that the condition a[i]≥e does not appear in the size relation of Eq. (6) nor (7). This is because the array a has been abstracted to its length. Thus, the value in a[i] is no longer observable. For our cost measure , we count 3 bytecode instructions in Eq. (5) and 8 in Eq. (6). The cost of executing an iteration of the loop is

captured by Eq. (7), where the condition $i<la$ must be satisfied and variable i is increased by one at each recursive call. □

1.2 Cost Relations vs. Recurrence Relations

CRs differ from standard RRs in the following ways:

(a) *Non-determinism.* In contrast to RRs, CRs are possibly non-deterministic: equations for the same relation are not required to be mutually exclusive. Even if the programming language is deterministic, *size abstractions* introduce a loss of precision: some guards which make the original program deterministic may not be observable when using the size of arguments instead of their actual values. In Ex. 1, this happens between Eqs. (3) and (4) and also between (6) and (7).

(b) *Inexact size relations.* CRs may have size relations which contain constraints (not equalities). When dealing with realistic programming languages which contain non-linear data structures, such as trees, it is often the case that size analysis does not produce exact results. E.g., analysis may infer that the size of a data structure strictly decreases from one iteration to another, but it may be unable to provide the precise reduction. This happens in Ex. 1 in Eqs. (3) and (4).

(c) *Multiple arguments.* CRs usually depend on several arguments that may increase (variable i in Eq. (7)) or decrease (variable l in Eq. (2)) at each iteration. In fact, the number of times that a relation is executed can be a combination of several of its arguments. E.g., relation E is executed $la-j-1$ times.

Point (a) was detected already in [25], where an explicit `when` operator is added to the RR language to introduce non-determinism, but no complete method for handling it is provided. Point (b) is another source of non-determinism. As a result, CRs do not define functions, but rather relations. Given a relation C and input values $\overline{v}$, there may exist multiple results for $C(\overline{v})$. Sometimes it is possible to automatically convert relations with several arguments into relations with only one. However, in contrast to our approach, it is restricted to very simple cases such as when the CR only count constant cost expressions.

Existing methods for solving RRs are insufficient to bound CRs since they do not cover points (a), (b), and (c) above. On the other hand, CASs can solve complex recurrences (e.g., coefficients to function calls can be polynomials) which our framework cannot handle. However, this additional power is not needed in cost analysis, since such recurrences do not occur as the result of cost analysis.

An obvious way of obtaining upper bounds in non-deterministic CRs would be to introduce a maximization operator. Unfortunately, such operator is not supported by existing CAS. Adding it is far from trivial, since computing the maximum when the equations are not mutually exclusive requires taking into account multiple possibilities, which results in a highly combinatorial problem. Another possibility is to convert CRs into RRs. For this, we need to remove equations from CRs as well as sometimes to replace inexact size relations by exact ones while preserving the worst-case solution. However, this is not possible in general. E.g., in Fig. 1, the maximum cost is obtained when the execution interleaves Eqs. (3) and (4), and therefore we cannot remove either of them.

2 Cost Relations: Evaluation and Upper Bounds

Let us introduce some notation. We use x, y, z, possibly subscripted, to denote variables which range over integers ($\mathbb{Z}$), v, w denote integer values, a, b natural numbers ($\mathbb{N}$) and q rational numbers ($\mathbb{Q}$). We denote by $\mathbb{Q}^+$ (resp. $\mathbb{R}^+$) the set of non-negative rational (resp. real) numbers. We use $\overline{t}$ to denote a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We sometimes apply set operations on sequences. Given $\overline{x}$, an *assignment* for $\overline{x}$ is a sequence $\overline{v}$ (denoted by $[\overline{x}/\overline{v}]$). Given any entity t, $t[\overline{x}/\overline{v}]$ stands for the result of replacing in t each occurrence of x_i by v_i. We use $vars(t)$ to refer to the set of variables occurring in t. A *linear expression* has the form $q_0{+}q_1x_1{+}\cdots{+}q_nx_n$. A *linear constraint* has the form l_1 *op* l_2 where l_1 and l_2 are linear expressions and $op \in \{=, \leq, <, >, \geq\}$. A *size relation* φ is a set of linear constraints (interpreted as a conjunction). The operator $\bar{\exists}\overline{x}.\varphi$ eliminates from φ all variables except for $\overline{x}$. We write $\varphi_1 \models \varphi_2$ to indicate that φ_1 implies φ_2. The following definition presents our notion of *basic cost expression*.

Definition 1 (basic cost expression). Basic cost expressions *are of the form:* $\mathtt{exp}{::=}a|\mathsf{nat}(l)|\mathtt{exp}{+}\mathtt{exp}|\mathtt{exp}{*}\mathtt{exp}|\mathtt{exp}^a|\log_{\mathtt{a}}(\mathtt{exp})|a^{\mathtt{exp}}|\max(S)|\frac{\mathtt{exp}}{a}|\mathtt{exp}{-}a$*, where* $a{\geq}1$*,* l *is a linear expression,* S *is a non empty set of cost expressions,* $\mathsf{nat}{:}\mathbb{Z}{\rightarrow}\mathbb{Q}^+$ *is defined as* $\mathsf{nat}(v){=}\max(\{v, 0\})$*, and* $\mathtt{exp}$ *satisfies that for any assignment* $\overline{v}$ *for* $vars(\mathtt{exp})$ *we have that* $\mathtt{exp}[vars(\mathtt{exp})/\overline{v}] \in \mathbb{R}^+$*.*

Basic cost expressions are symbolic expressions which indicate the resources we accumulate and are the non-recursive building blocks for defining cost relations. They enjoy two crucial properties: (1) by definition, they are always evaluated to non negative values; (2) replacing a sub-expression $\mathsf{nat}(l)$ by $\mathsf{nat}(l')$ such that $l'{\geq}l$, results in an upper bound of the original expression.

A *cost relation* C of arity n is a subset of $\mathbb{Z}^n * \mathbb{R}^+$. This means that for a single tuple $\overline{v}$ of integers there can be multiple solutions in $C(\overline{v})$. We use C and D to refer to cost relations. Cost analysis of a program usually produces multiple, interconnected, cost relations. We refer to such sets of cost relations as *cost relation systems* (CRSs for short), which we formally define below.

Definition 2 (Cost Relation System). *A* cost relation system $\mathcal{S}$ *is a set of equations of the form* $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle$ *with* $k{\geq}0$*, where* C *and all* D_i *are cost relations, all variables* $\overline{x}$ *and* $\overline{y}_i$ *are distinct variables;* $\mathtt{exp}$ *is a basic cost expression; and* φ *is a size relation between* $\bar{x}$ *and* $\bar{x}{\cup}vars(\mathtt{exp}){\cup}\bar{y_i}$*.*

In contrast to standard definitions of RRs, the variables which occur in the rhs of the equations in CRSs do not need to be related to those in the lhs by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by $rel(\mathcal{S})$ the set of cost relations which are defined in $\mathcal{S}$. Also, $def(\mathcal{S}, C)$ denotes the subset of the equations in $\mathcal{S}$ whose lhs is of the form $C(\overline{x})$. W.l.o.g. we assume that all equations in $def(\mathcal{S}, C)$ have the same variable names in the lhs. We assume that any CRS $\mathcal{S}$ is self-contained in the sense that all cost relations which appear in the rhs of an equation in $\mathcal{S}$ must be in $rel(\mathcal{S})$.

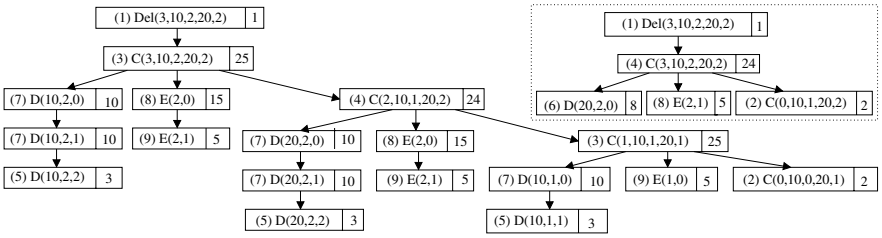


Fig. 2. Two Evaluation Trees for $Del(3, 10, 2, 20, 2)$

We now provide a semantics for CRSs. Given a CRS $\mathcal{S}$, a *call* is of the form $C(\overline{v})$, where $C{\in}rel(\mathcal{S})$ and $\overline{v}$ are integer values. Calls are evaluated in two phases. In the first phase, we build an *evaluation tree* for the call. In the second phase we obtain a *value* in $\mathbb{R}^+$ by adding up the constants which appear in the nodes of the evaluation tree. We make evaluation trees explicit since, as discussed below, our approximation techniques are based on reasoning about the number of nodes and the values in the nodes in such evaluation trees. Evaluation trees are obtained by repeatedly expanding nodes which contain calls to relations. Each expansion is performed w.r.t an appropriate instantiation of a rhs of an applicable equation. If all leaves in the tree contain basic cost expressions then there is no node left to expand and the process terminates. We will represent evaluation trees using nested terms of the form $node(Call,Local\_Cost,Children)$, where $Local\_Cost$ is a constant in $\mathbb{R}^+$ and $Children$ is a sequence of evaluation trees.

Definition 3 (evaluation tree). *Given a CRS $\mathcal{S}$ and a call $C(\overline{v})$, a tree node $(C(\overline{v}), e, \langle T_1, \ldots, T_k\rangle)$ is an* evaluation tree *for $C(\overline{v})$ in $\mathcal{S}$, denoted $\mathsf{Tree}(C(\overline{v}), \mathcal{S})$ if: 1) there is a renamed apart equation $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle \in \mathcal{S}$ s.t. φ' is satisfiable in $\mathbb{Z}$, with $\varphi'{=}\varphi[\overline{x}/\overline{v}]$, and 2) there exist assignments $\overline{w}, \overline{v}_i$ for $vars(\mathtt{exp}), \overline{y}_i$ respectively s.t. $\varphi'[vars(\mathtt{exp})/\overline{w},\ \overline{y}_i/\overline{v}_i]$ is satisfiable in $\mathbb{Z}$, and 3) $e{=}\mathtt{exp}[vars(\mathtt{exp})/\overline{w}]$, T_i is an evaluation tree $\mathsf{Tree}(D_i(\overline{v}_i), \mathcal{S})$ with $i = 0, \ldots, k$.*

In step 1 we look for an equation $\mathcal{E}$ which is applicable for solving $C(\overline{v})$. Note that there may be several equations which are applicable. In step 2 we look for assignments for the variables in the rhs of $\mathcal{E}$ which satisfy the size relations associated to $\mathcal{E}$. This a non-deterministic step as there may be (infinitely many) different assignments which satisfy all size relations. Finally, in step 3 we apply the assignment to `exp` and continue recursively evaluating the calls. We use $Trees(C(\overline{v}), \mathcal{S})$ to denote the set of all evaluation trees for $C(\overline{v})$. We define $Answers(C(\overline{v}), \mathcal{S}){=}\{\mathsf{Sum}(T) \mid T{\in}Trees(C(\overline{v}), \mathcal{S})\}$, where $\mathsf{Sum}(T)$ traverses all nodes in T and computes the sum of the cost expressions in them.

Example 2. Fig. 2 shows two possible evaluation trees for $Del(3, 10, 2, 20, 2)$. The tree on the left has maximal cost, whereas the one on the right has minimal cost. A node in either tree contains a call (left box) and its local cost (right box) and it is linked by arrows to its children. We annotate calls with a number in

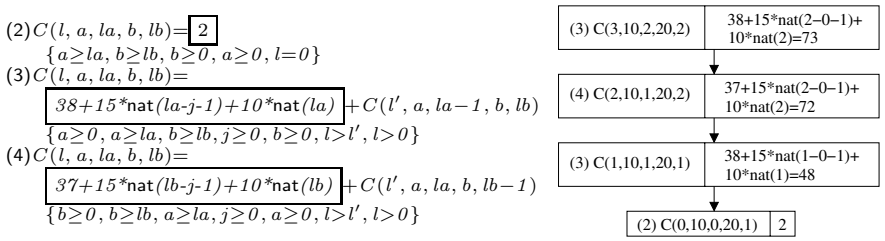


Fig. 3. Self-Contained CR for relation C and a corresponding evaluation tree

parenthesis to indicate the equation which was selected for evaluating such call. Note that, in the recursive call to C in Eqs. (3) and (4), we are allowed to pick any value l' s.t. $l'<l$. In the tree on the left we always assign $l'=l-1$. This is what happens in actual executions of the program. In the tree on the right we assign $l'=l-3$ in the recursive call to C. The latter results in a minimal approximation, however, it does not correspond to any actual execution. This is a side effect of using safe approximations in static analysis: information is correct in the sense that at least one of the evaluation trees must correspond to the actual cost, but there may be other trees with different cost. In fact, there are an infinite number of evaluation trees for our example call, as step 2 can provide an infinite number of assignments to variable j which are compatible with the constraint $j\geq 0$ in Eqs. (3) and (4). This shows that approaches like [13] based on evaluation of CRSs are not of general applicability. Nevertheless, it is possible to find an upper bound for this call since though the number of trees is infinite, infinitely many of them produce equivalent results. □

2.1 Closed Form Upper Bounds for Cost Relations

Let C be a relation over $\mathbb{Z}^n * \mathbb{R}^+$. A function $U{:}\mathbb{Z}^n \to \mathbb{R}^+$ is an *upper bound* of C iff $\forall \overline{v} \in \mathbb{Z}^n, \forall a \in Answers(C(\overline{v}), \mathcal{S}), U(\overline{v}) \geq a$. We use C^+ to refer to an upper bound of C. A function $f{:}\mathbb{Z}^n \to \mathbb{R}^+$ is in *closed form* if it is defined as $f(\overline{x})=\mathtt{exp}$, with $\mathtt{exp}$ a basic cost expression s.t. $vars(\mathtt{exp}) \subseteq \overline{x}$. An important feature of CRSs, inherited from RRs, is their *compositionality*, which allows computing upper bounds of CRSs by concentrating on one relation at a time. I.e., given a cost equation for $C(\overline{x})$ which calls $D(\overline{y})$, we can replace the call to $D(\overline{y})$ by $D^+(\overline{y})$. The resulting relation is trivially an upper bound of the original one. E.g., suppose that we have the following upper bounds: $E^+(la, j)=5+15*\mathsf{nat}(la-j-1)$ and $D^+(a, la, i)=8+10*\mathsf{nat}(la-i)$. Replacing the calls to D and E in equations (3) and (4) by D^+ and E^+ results in the CRS shown in Fig. 3.

The compositionality principle only results in an effective mechanism if all recursions are *direct* (i.e., all cycles are of length one). In that case we can start by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as *standalone cost relations* and continue by replacing

the computed upper bounds on the equations which call such relations. In the following, we formalize our method by assuming standalone cost relations and in Sec. 6 we provide a mechanism for obtaining direct recursion automatically.

Existing approaches to compute upper bounds and asymptotic complexity of RRs, usually applied by hand, are based on reasoning about evaluation trees in terms of their size, depth, number of nodes, etc. They typically consider two categories of nodes: (1) *internal* nodes, which correspond to applying recursive equations, and (2) *leaves* of the tree(s), which correspond to the application of a base (non-recursive) case. The central idea then is to count (or obtain an upper bound on) the number of leaves and the number of internal nodes in the tree separately and then multiply each of these by an upper bound on the cost of the base case and of a recursive step, respectively. For instance, in the evaluation tree in Fig. 3 for the standalone cost relation C, there are three internal nodes and one leaf. The values in the internal nodes, once performed the evaluation of the expressions are 73, 72, and 48, therefore 73 is the worst case. In the case of leaves, the only value is 2. Therefore, the tightest upper bound we can find using this approximation is $3{\times}73{+}1{*}2{=}221 \geq 73{+}72{+}48{+}2{=}193$.

We now extend the approximation scheme mentioned above in order to consider all possible evaluation trees which may exist for a call. In the following, we use $|S|$ to denote the cardinality of a set S. Also, given an evaluation tree T, $\mathit{leaf}(T)$ denotes the set of leaves of T (i.e., those without children) and $\mathit{internal}(T)$ denotes the set of internal nodes (all nodes but the leaves) of T.

Proposition 1 (node-count upper bound). *Let C be a cost relation and let $C^+(\overline{x}) = \mathit{internal}^+(\overline{x}) * \mathit{costr}^+(\overline{x}) + \mathit{leaf}^+(\overline{x}) * \mathit{costnr}^+(\overline{x})$, where $\mathit{internal}^+(\overline{x})$, $\mathit{costr}^+(\overline{x})$, $\mathit{leaf}^+(\overline{x})$ and $\mathit{costnr}^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \to \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and for all $T \in \mathit{Trees}(C(\overline{v}), \mathcal{S})$, it holds: (1) $\mathit{internal}^+(\overline{v}) \geq |\mathit{internal}(T)|$ and $\mathit{leaf}^+(\overline{v}) \geq |\mathit{leaf}(T)|$; (2) $\mathit{costr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{internal}(T)\}$ and (3) $\mathit{costnr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{leaf}(T)\}$.*

3 Upper Bounds on the Number of Nodes

In this section we present an automatic mechanism to obtain safe $\mathit{internal}^+(\overline{x})$ and $\mathit{leaf}^+(\overline{x})$ functions which are valid for any assignment for $\overline{x}$. The basic idea is to first obtain upper bounds b and $h^+(\overline{x})$ on, respectively, the *branching factor* and *height* (the distance from the root to the deepest leaf) of all corresponding evaluation trees, and then use the number of internal nodes and leaves of a *complete* tree with such branching factor and height as an upper bound. Then,

$$\mathit{leaf}^+(\overline{x}) = b^{h^+(\overline{x})} \qquad \mathit{internal}^+(\overline{x}) = \begin{cases} h^+(\overline{x}) & b{=}1 \\ \frac{b^{h^+(\overline{x})}-1}{b-1} & b{\geq}2 \end{cases}$$

For a cost relation C, the branching factor b in any evaluation tree for a call $C(\overline{v})$ is limited by the maximum number of recursive calls which occur in a single equation for C. We now propose a way to compute an upper bound for

the height, h^+. Given an evaluation tree $T \in Trees(C(\overline{v}), \mathcal{S})$ for a cost relation C, consecutive nodes in any branch of T represent consecutive recursive calls which occur during the evaluation of $C(\overline{v})$. Therefore, bounding the height of a tree may be reduced to bounding consecutive recursive calls. The notion of *loop* in a cost relation, which we introduce below, is used to model consecutive calls.

Definition 4. *Let* $\mathcal{E}=\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k} C(\overline{y_i}), \varphi\rangle$ *be an equation for a cost relation* C*. Then,* $Loops(\mathcal{E})=\{\langle C(\overline{x})\rightarrow C(\bar{y}_i), \varphi'\rangle \mid \varphi'=\bar{\exists}\bar{x}\cup\bar{y}_i.\varphi, i=1\cdots k\}$ *is the set of* loops *induced by* $\mathcal{E}$*. Similarly,* $Loops(C) = \cup_{\mathcal{E}\in def(\mathcal{S},C)} Loops(\mathcal{E})$*.*

Example 3. Eqs. (3) and (4) in Fig. 3 induce the following two loops:

(3)$\langle C(l, a, la, b, lb)\rightarrow C(l', a, la', b, lb), \varphi'_1=\{a\geq 0, a\geq la, b\geq lb, b\geq 0, l>l', l>0, la'=la-1\}\rangle$
(4)$\langle C(l, a, la, b, lb)\rightarrow C(l', a, la, b, lb'), \varphi'_2=\{b\geq 0, b\geq lb, a\geq la, a\geq 0, l>l', l>0, lb'=lb-1\}\rangle$

Bounding the number of consecutive recursive calls is extensively used in the context of termination analysis. It is usually done by proving that there is a function f from the loop's arguments to a *well-founded* partial order which decreases in any two consecutive calls and which guarantees the absence of infinite traces, and thus termination. These functions are usually called *ranking functions*. We propose to use the ranking function to generate a h^+ function. In practice, we use [21] to generate functions which are defined as follows: a function $f:\mathbb{Z}^n\mapsto\mathbb{Z}$ is a *ranking function* for a loop $\langle C(\bar{x})\rightarrow C(\bar{y}), \varphi\rangle$ if $\varphi\models f(\bar{x})>f(\bar{y})$ and $\varphi\models f(\bar{x})\geq 0$.

Example 4. The function $f_C(l, a, la, b, lb)=l$ is a ranking function for C in the cost relation in Fig. 3. Note that φ'_1 and φ'_2 in the above loops of C contain the constraints $\{l>l', l>0\}$ which is enough to guarantee that f_C is decreasing and well-founded. The height of the evaluation tree for $C(3, 10, 2, 20, 2)$ is precisely predicted by $f_C(3, 10, 2, 20, 2)=3$. Ranking functions may involve several arguments, e.g., $f_D(a, la, i)=la-i$ is a ranking function for $\langle D(a, la, i) \rightarrow D(a, la, i'), \{i'=i+1, i<la, a\geq la, i\geq 0\}\rangle$ which comes from Eq. (7). □

Observe that the use of global ranking functions allows bounding the number of iterations of possibly non-deterministic CRSs with multiple arguments (see Sec. 1.2). In order to be able to define h^+ in terms of the ranking function, one thing to fix is that the ranking function might return a negative value when is applied to values which correspond to base cases (leaves of the tree). Therefore, we define $h^+(\overline{x})=\mathsf{nat}(f_C(\overline{x}))$. Function nat guarantees that negative values are lifted to 0 and, therefore, they provide a correct approximation for the height of evaluation trees with a single node. Even though the ranking function provides an upper bound for the height of the corresponding trees, in some cases we can further refine it and obtain a tighter upper bound. For example, if the difference between the value of the ranking function in each two consecutive calls is larger than a constant $\delta>1$, then $\lceil \mathsf{nat}(\frac{f_C(\bar{x})}{\delta})\rceil$ is a tighter upper bound. A more interesting case, if each loop $\langle C(\overline{x})\rightarrow C(\overline{y}), \varphi\rangle \in Loops(C)$ satisfies $\varphi\models f_C(\overline{x})\geq k*f_C(\overline{y})$ where $k>1$, then the height of the tree is bounded by $\lceil \log_k(\mathsf{nat}(f_C(\overline{v})+1))\rceil$.

4 Estimating the Cost Per Node

Consider the evaluation tree in Fig. 3. Note that all expressions in the nodes are instances of the expressions which appear in the corresponding equations. Thus, computing $costr^+(\overline{x})$ and $costnr^+(\overline{x})$ can be done by first finding an upper bound of such expressions and then combining them through a *max* operator. We first compute *invariants* for the values that the expression variables can take w.r.t. the initial values, and use them to derive upper bounds for such expressions.

4.1 Invariants

Computing an *invariant* (in terms of linear constraints) that holds in all *calling contexts* (*contexts* for short) to a relation C between the arguments at the initial call and at each call during the evaluation can be done by using $Loops(C)$. Intuitively, if we know that a linear constraint ψ holds between the arguments of the initial call $C(\overline{x}_0)$ and those of a recursive call $C(\overline{x})$, denoted $\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}), \psi\rangle$, and we have a loop $\langle C(\overline{x}){\rightarrow}C(\overline{y}), \varphi\rangle{\in}Loops(C)$, then we can apply the loop one more step and get the new *calling context* $\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{y}), \overline{\exists\overline{x_0}\cup\overline{y}.\psi\wedge\varphi}\rangle$.

Definition 5 (loop invariants). *For a relation C, let $\mathcal{T}$ be an operator defined:*

$$\mathcal{T}(X) = \left\{ \langle C(\overline{x}_0){\rightsquigarrow}C(\overline{y}), \psi'\rangle \;\middle|\; \begin{array}{l}\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}), \psi\rangle{\in}X, \langle C(\overline{x}){\rightarrow}C(\overline{y}), \varphi\rangle{\in}Loops(C),\\ \psi'=\overline{\exists\overline{x_0}\cup\overline{y}.\psi\wedge\varphi}\end{array} \right\}$$

which derives a set of contexts, from a given context X, by applying all loops, then the loop invariants I is $lfp{\cup_{i\geq 0}}\mathcal{T}^i(I_0)$ where $I_0 = \{\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}), \{\overline{x}_0{=}\overline{x}\}\rangle\}$.

Example 5. Let us compute I for the loops in Sec. 3. The initial context is $I_1{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{=}l_0, a{=}a_0, la{=}la_0, b{=}b_0, lb{=}lb_0\}\rangle$ where $\bar{x}_0{=}\langle l_0, a_0, la_0, b_0, lb_0\rangle$ and $\bar{x}{=}\langle l, a, la, b, lb\rangle$. In the first iteration we compute $\mathcal{T}^0(\{I_1\})$ which by definition is $\{I_1\}$. In the second iteration we compute $\mathcal{T}^1(\{I_1\})$ which results in

$$I_2{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{\underline{l{<}l_0}, a{=}a_0, \underline{la{=}la_0{-}1}, b{=}b_0, lb{=}lb_0, \underline{l_0{>}0}\}\rangle$$
$$I_3{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{\underline{l{<}l_0}, a{=}a_0, la{=}la_0, b{=}b_0, \underline{lb{=}lb_0{-}1}, \underline{l_0{>}0}\}\rangle$$

where I_2 and I_3 correspond to applying respectively the first loop and second loops on I_1. The underlined constraints are the modifications due to the application of the loop. Note that in I_2 the variable la_0 decreases by one, and in I_3 lb_0 decreases by one. The third iteration $\mathcal{T}^2(\{I_1\})$, i.e. $\mathcal{T}(\{I_2, I_3\})$, results in

$$I_4{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, \underline{la{=}la_0{-}2}, b{=}b_0, lb{=}lb_0, l_0{>}0\}\rangle$$
$$I_5{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, \underline{la{=}la_0{-}1}, b{=}b_0, \underline{lb{=}lb_0{-}1}, l_0{>}0\}\rangle$$
$$I_6{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, la{=}la_0, b{=}b_0, \underline{lb{=}lb_0{-}2}, l_0{>}0\}\rangle$$
$$I_7{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, \underline{la{=}la_0{-}1}, b{=}b_0, lb{=}lb_0{-}1, l_0{>}0\}\rangle$$

where I_4 and I_5 originate from applying the loops to I_2, and I_6 and I_7 from applying the loops to I_3. The modifications on the constraints reflect that, when applying a loop, either we decrease *la* or *lb*. After three iterations, the invariant I includes $I_1 \cdots I_7$. More iterations will add more contexts that further modify the value of *la* or *lb*. Therefore, the invariant I grows indefinitely in this case. □

In practice, we approximate I using abstract interpretation over, for instance, the domain of convex polyhedra [10], whereby we obtain the invariant $\Psi = \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \{l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}\rangle$.

4.2 Upper Bounds on Cost Expressions

Once invariants are available, finding upper bounds of cost expressions can be done by maximizing their nat parts independently. This is possible due to the monotonicity property of cost expressions. Consider, for example, the expression $\mathsf{nat}(la-j-1)$ which appears in equation (3) of Fig. 3. We want to infer an upper bound of the values that it can be evaluated to in terms of the input values $\langle l_0, a_0, la_0, b_0, lb_0\rangle$. We have inferred, in Sec. 4.1, that whenever we call C the invariant Ψ holds, from which we can see that the maximum value that la can take is la_0. In addition, from the local size relations φ of equation (3) we know that $j \geq 0$. Since $la-j-1$ takes its maximal value when la is maximal and j is minimal, the expression la_0-1 is an upper bound for $la-j-1$. This can be done automatically using linear constraints tools [6]. Given a cost equation $\langle C(\overline{x}) = \mathtt{exp} + \sum_{i=0}^{k} C(\overline{y}_i),\ \varphi\rangle$ and an invariant $\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$, the function below computes an upper bound for exp by maximizing its nat components.

```
1: function ub_exp(exp,x̄0,φ,Ψ)
2:     mexp=exp
3:     for all nat(f)∈exp do
4:         Ψ'=∃x̄0, r.(φ∧Ψ∧(r=f))      // r is a fresh variable
5:         if ∃f' s.t. vars(f')⊆x̄0 and Ψ'⊨r≤f' then mexp=mexp[nat(f)/nat(f')]
6:         else return ∞
7:     return mexp
```

This function computes an upper bound f' for each expression f which occurs inside a nat operator and then replaces in exp all such f expressions with their corresponding upper bounds (line 5). If it cannot find an upper bound, the method returns ∞ (line 6). The *ub_exp* function is complete in the sense that if Ψ and φ imply that there is an upper bound for a given $\mathsf{nat}(f)$, then we can find one by *syntactically* looking on Ψ' (line 4).

Example 6. Applying *ub_exp* to $\mathtt{exp}_3$ and $\mathtt{exp}_4$ of Eqs. (3) and (4) in Fig. 3 w.r.t. the invariant we have computed in Sec. 4.1 results in $\mathtt{mexp}_3 = 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)$ and $\mathtt{mexp}_4 = 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0)$. □

Theorem 1. *Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ be a cost relation where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively the sets of non-recursive and recursive equations for C, and let $\mathcal{I} = \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$ be a loop invariant for C; $E_i = \{ub\_exp(\mathtt{exp}, \overline{x}_0, \varphi, \Psi) \mid \langle C(\overline{x}) = \mathtt{exp} + \sum_{j=0}^{k} C(\overline{y}_j), \varphi\rangle \in \mathcal{S}_i\}$; $costnr^+(\overline{x}_0) = \max(E_1)$ and $costr^+(\overline{x}_0) = \max(E_2)$. Then for any call $C(\overline{v})$ and for all $T \in Trees(C(\overline{v}), \mathcal{S})$: (1) $\forall node(\_, e, \_) \in internal(T)$ we have $costr^+(\overline{v}) \geq e$; and (2) $\forall node(\_, e, \_) \in leaf(T)$ we have $costnr^+(\overline{v}) \geq e$.*

Example 7. At this point we have all the pieces in order to compute an upper bound for the CRS depicted in Fig. 1 as described in Prop. 1. We start by computing upper bounds for E and D as they are cost relations:

	Ranking Function	$costnr^+$	$costr^+$	*Upper Bound*
$E(la_0, j_0)$	$\mathsf{nat}(la_0 - j_0 - 1)$	5	15	$5+15*\mathsf{nat}(la_0 - j_0 - 1)$
$D(a_0, la_0, i_0)$	$\mathsf{nat}(la_0 - i_0)$	8	10	$8+10*\mathsf{nat}(la_0 - i_0)$

These upper bounds can then be substituted in the equations (3) and (4) which results in the cost relation for C depicted in Fig. 3. We have already computed a ranking function for C in Ex. 4 and $costnr^+$ and $costr^+$ in Ex. 6, which are then combined into $C^+(l_0, a_0, la_0, b_0, lb_0) = 2+\mathsf{nat}(l_0)*max(\{\mathtt{mexp}_3, \mathtt{mexp}_4\})$. Reasoning similarly, for *Del* we get the upper bound shown in Table 1. □

5 Improving Accuracy in Divide and Conquer Programs

For some CRSs, we can obtain a more accurate upper bound by approximating the cost of *levels* instead of approximating the cost of nodes, as indicated by Prop. 1. Given an evaluation tree T, we denote by $\mathsf{Sum\_Level}(T, i)$ the sum of the values of all nodes in T which are at depth i, i.e., at distance i from the root.

Proposition 2 (level-count upper bound). *Let C be a cost relation and let C^+ be a function defined as: $C^+(\overline{x}) = l^+(\overline{x}) * costl^+(\overline{x})$, where $l^+(\overline{x})$ and $costl^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \to \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and $T \in Trees(C(\overline{v}), \mathcal{S})$, it holds: (1) $l^+(\overline{v}) \geq depth(T) + 1$; and (2) $\forall i \in \{0, \ldots, depth(T)\}$ we have that $costl^+(\overline{v}) \geq \mathsf{Sum\_Level}(T, i)$.*

The function l^+ can simply be defined as $l^+(\overline{x}) = \mathsf{nat}(f_C(\overline{x}))+1$ (see Sec. 3). Finding an accurate $costl^+$ function is not easy in general, which makes Prop. 2 not as widely applicable as Prop. 1. However, evaluation trees for *divide and conquer* programs satisfy that $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$, i.e., the cost per level does not increase from one level to another. In that case, we can take the cost of the root node as an upper bound of $costl^+(\overline{x})$. A sufficient condition for a cost relation falling into the divide and conquer class is that each cost expression that is contributed by an equation is greater than or equal to the sum of the cost expressions contributed by the corresponding immediate recursive calls. This check is implemented in our prototype using [6].

Consider a CRS with the two equations $\langle C(n) = 0, \{n \leq 0\}\rangle$ and $\langle C(n) = \mathsf{nat}(n)+C(n_1)+C(n_2), \varphi\rangle$ where $\varphi = \{n>0, n_1+n_2+1 \leq n, n \geq 2*n_1, n \geq 2*n_2, n_1 \geq 0, n_2 \geq 0\}$. It corresponds to a divide and conquer problem such as merge-sort. In order to prove that $\mathsf{Sum\_Level}$ does not increase, it is enough to check that, in the second equation, n is greater than or equal to the sum of the expressions that immediately result from the calls $C(n_1)$ and $C(n_2)$, which are n_1 and n_2 respectively. This can be done by simply checking that $\varphi \models n \geq n_1+n_2$. Then, $costl^+(\overline{x}) = max\{0, \mathsf{nat}(x)\} = \mathsf{nat}(x)$. Thus, given that $l^+(x) = \lceil \log_2(\mathsf{nat}(x)+1) \rceil + 1$, we obtain the upper bound $\mathsf{nat}(x) * (\lceil \log_2(\mathsf{nat}(x)+1) \rceil + 1)$. Note that by using the node-count approach we would obtain $\mathsf{nat}(x) * (2^{\mathsf{nat}(x)} - 1)$ as upper bound.

6 Direct Recursion Using Partial Evaluation

Automatically generated CRSs often contain recursions which are not direct, i.e., cycles involve more than one function. E.g., the actual CRS obtained for

the program in Fig. 1 by the analysis in [3] differs from that shown in the right hand side of Fig. 1 in that, instead of Eqs. (8) and (9), the "for" loop results in:

(8') $E(la,j)=5+F(la,j,j',la')$ $\{j'=j, la'=la-1, j'\geq 0\}$
(9') $F(la,j,j',la')=H(j',la')$ $\{j'\geq la'\}$
(10) $F(la,j,j',la')=G(la,j,j',la')$ $\{j'<la'\}$
(11) $H(j',la')=0$ $\{\}$
(12) $G(la,j,j',la')=10+E(la,j+1)$ $\{j<la-1, j\geq 0, la-la'=1, j'=j\}$

Now, E captures the cost of the loop condition "$j<la-1$" (5 cost units) plus the cost of its continuation, captured by F. Eq. (9') corresponds to the exit of the loop (it calls H, Eq. (11), which has 0 cost). Eq. (10) captures the cost of one iteration by calling G, Eq. (12), which accumulates 10 units and returns to E.

In this section we present an automatic transformation of CRSs into *directly recursive* form. The transformation is based on *partial evaluation* (PE) [17] and it is performed by replacing calls to intermediate relations by their definitions using *unfolding*. The first step in the transformation is to find a *binding time classification* (or BTC for short) which declares which relations are *residual*, i.e., they have to remain in the CRS. The remaining relations are considered *unfoldable*, i.e., they are eliminated. For computing BTCs, we associate to each CRS $\mathcal{S}$ a *call graph*, denoted $\mathcal{G}(\mathcal{S})$, which is the directed graph obtained from $\mathcal{S}$ by taking $rel(\mathcal{S})$ as the set of nodes and by including an arc (C, D) iff D appears in the rhs of an equation for C. The following definition provides sufficient conditions on a BTC which guarantee that we obtain a directly recursive CRS.

Definition 6. *Let $\mathcal{G}(\mathcal{S})$ be the call graph of $\mathcal{S}$ and let SCC be its strongly connected components. A BTC* btc *for $\mathcal{S}$ is* directly recursive *if for all $S\in SCC$ the following conditions hold: (1) if $s_1, s_2\in S$ and $s_1, s_2\in$ btc, then $s_1=s_2$; and (2) if S has a cycle, then there exists $s\in S$ such that $s\in$ btc.*

Condition 1 ensures that all recursions in the transformed CRS are direct, as there is only one residual relation per SCC. Condition 2 guarantees that the unfolding process terminates, as there is a residual relation per cycle. A directly recursive BTC for the above example is btc$=\{E\}$. In our implementation we only include in the BTC the *covering point* (i.e., a node which is part of all cycles) of SCCs which contain cycles, but no node is included for SCCs without cycles. This way of computing BTCs, in addition to ensuring direct recursion, also eliminates all relations which are not part of cycles (such as H in our example).

We now define unfolding in the context of CRSs. Such unfolding is guided by a BTC and at each step it combines both cost expressions and size relations.

Definition 7 (unfolding). *Given a CRS $\mathcal{S}$, a call $C(\overline{x}_0)$ s.t. $C\in rel(\mathcal{S})$, a size relation $\varphi_{\overline{x}_0}$ over $\overline{x}_0$, and a BTC* btc *for $\mathcal{S}$, a pair $\langle E, \varphi\rangle$ is an* unfolding *for $C(\overline{x}_0)$ and $\varphi_{\overline{x}_0}$ in $\mathcal{S}$ w.r.t.* btc*, denoted* $\mathsf{Unfold}(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$*, if either of the following conditions hold:*

(res) $C\in\mathsf{btc}\wedge\varphi\neq\mathsf{true}\wedge\langle E, \varphi\rangle=\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle$;
(unf) $(C\notin\mathsf{btc}\vee\varphi=\mathsf{true})\wedge\langle E, \varphi\rangle=\langle(\mathsf{exp}+e_1+\ldots+e_k), \varphi' \bigwedge_{i=1..k} \varphi_i\rangle$

where $\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k} D_i(\overline{y}_i), \varphi_C\rangle$ is a renamed apart equation in $\mathcal{S}$ s.t. $\varphi' = \varphi_{\overline{x}_0}\wedge\varphi_C[\overline{x}/\overline{x}_0]$ is satisfiable in $\mathbb{Z}$ and $\forall 1\leq i\leq k$ $\mathsf{Unfold}(\langle D_i(\overline{y_i}), \varphi'\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle e_i, \varphi_i\rangle$.

The first case, (**res**), is required for termination. When we call a relation C which is marked as residual, we simply return the initial call $C(\overline{x}_0)$ and size relation $\varphi_{\overline{x}_0}$, as long as the current size relation $\varphi_{\overline{x}_0}$ is not the initial one (true). The latter condition is added in order to force the initial unfolding step for relations marked as residual. In all subsequent calls to Unfold different from the initial one, the size relation is different from true. The second case (**unf**) corresponds to continuing the unfolding process. Note that step 1 is non-deterministic, since often cost relations contain several equations. Since expressions are transitively unfolded, step 2 may also provide multiple solutions. Also, if the final size relation φ is unsatisfiable, we simply do not regard $\langle E, \varphi\rangle$ as a valid unfolding.

Example 8. Given the initial call $\langle E(la, j), true\rangle$, we obtain an unfolding by performing the following steps, denoted by $\overset{e}{\rightsquigarrow}$ where e is the selected equation:
$\langle E(la,j), true\rangle \overset{(8')}{\rightsquigarrow} \langle 5{+}F(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0\}\rangle \overset{(10)}{\rightsquigarrow}$
$\langle 5{+}G(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0, j'{<}la'\}\rangle \overset{(12)}{\rightsquigarrow} \langle 15{+}E(la,j''), \{j{<}la{-}1, j{\geq}0\}\rangle$
The call $E(la, j'')$ is not further unfolded as E belongs to btc and $\varphi{\neq}true$. □

From each result of unfolding we can build a *residual equation*. Given the unfolding $\mathsf{Unfold}(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$ its corresponding residual equation is $\langle C(\overline{x}_0){=}E, \varphi\rangle$. As customary in PE, a *partial evaluation* of C is obtained by collecting all residual equations for the call $\langle C(\overline{x}_0), \mathsf{true}\rangle$. The PE of $\langle E(la, j), true\rangle$ results in Eqs. (8) and (9) of Fig. 1. Eq. (9) is obtained from the unfolding steps depicted in Ex. 8 and Eq. (8) from unfolding w.r.t. Eqs. (8'), (9'), and (11).

Correctness of PE ensures that the solutions of CRSs are preserved. Regarding completeness, we can obtain direct recursion if all SCCs in the call graph have covering point(s). Importantly, structured loops (for, while, etc.) and recursive patterns found in most programs result in CRSs that satisfy this property. In addition, before applying PE, we check that the CRS terminates [2] with respect to the initial query, otherwise we might compromise non-termination and thus lead to incorrect upper bounds. We believe this check is not required when CRSs are generated from imperative programs.

7 Experiments in Cost Analysis of Java Bytecode

A prototype implementation in Ciao Prolog, which uses PPL [6] for manipulating linear constraints, is available at `http://www.cliplab.org/Systems/PUBS`. We have performed a series of experiments which are shown in Table 1. We have used CRSs automatically generated by the cost analyzer of Java bytecode described in [3] using two cost measures: heap consumption for those marked with "*", and the number of executed bytecode instructions for the rest. The benchmarks are presented in increasing complexity order and grouped by asymptotic class. Those marked with M were solved using Mathematica® by [3] but after significant human intervention. The marks a, b and c after the name indicate, respectively, if the CRS is non-deterministic, has inexact size relations and multiple arguments (Sec. 1.2). Column $\#_{eq}$ shows the number of equations before PE (in brackets

Table 1. Experiments on Cost Analysis of Java Bytecode

Benchmark	$\#_{eq}$	T	$\#^c_{eq}$	$\mathbf{T}_{pe}$	$\mathbf{T}_{ub}$	Rat.	Upper Bound
Polynomial* abc	23 (3)	13	346 (70)	174	649	2.4	216
DivByTwo ab	9 (3)	3	323 (68)	166	596	2.4	$8\log_2(\mathsf{nat}(2x{-}1){+}1){+}14$
Factorial$^{\mathsf{M}}$	8 (2)	4	314 (66)	165	590	2.4	$9\mathsf{nat}(x){+}4$
ArrayRev$^{\mathsf{M}}$ a	9 (3)	4	305 (64)	165	579	2.4	$14\mathsf{nat}(x){+}12$
Concat$^{\mathsf{M}}$ ac	14 (5)	13	296 (62)	158	538	2.4	$11\mathsf{nat}(x){+}11\mathsf{nat}(y){+}25$
Incr$^{\mathsf{M}}$ ac	28 (5)	29	282 (58)	155	490	2.3	$19\mathsf{nat}(x{+}1){+}9$
ListRev$^{\mathsf{M}}$ abc	9 (3)	4	254 (54)	144	415	2.2	$13\mathsf{nat}(x){+}8$
MergeList abc	21 (4)	18	245 (52)	138	406	2.2	$29\mathsf{nat}(x{+}y){+}26$
Power	8 (2)	3	223 (48)	125	371	2.2	$10\mathsf{nat}(x){+}4$
Cons* ab	22 (2)	6	214 (46)	123	359	2.3	$22\mathsf{nat}(x{-}1){+}24$
EvenDigits abc	18 (5)	9	191 (44)	115	322	2.3	$\mathsf{nat}(x)(8\log_2(\mathsf{nat}(2x{-}3){+}1){+}24){+}9\mathsf{nat}(x){+}9$
ListInter abc	37 (9)	59	173 (40)	110	298	2.4	$\mathsf{nat}(x)(10\mathsf{nat}(y){+}43){+}21$
SelectOrd ac	19 (6)	27	136 (32)	86	198	2.1	$\mathsf{nat}(x{-}2)(17\mathsf{nat}(x{-}2){+}34){+}9$
FactSum a	17 (5)	8	117 (27)	76	173	2.1	$\mathsf{nat}(x{+}1)(9\mathsf{nat}(x){+}16){+}6$
Delete abc	33 (9)	125	100 (23)	71	165	2.4	$3{+}\mathsf{nat}(l)\max(38{+}15\mathsf{nat}(la{-}1){+}10\mathsf{nat}(la),$ $37{+}15\mathsf{nat}(lb{-}1){+}10\mathsf{nat}(lb))$
MatMult$^{\mathsf{M}}$ ac	19 (7)	23	67 (15)	27	40	1.0	$\mathsf{nat}(y)(\mathsf{nat}(x)(27\mathsf{nat}(x)){+}10){+}17$
Hanoi	9 (2)	4	48 (8)	23	17	0.8	$20(2^{\mathsf{nat}(x)})$-17
Fibonacci$^{\mathsf{M}}$	8 (2)	5	39 (6)	20	13	0.8	$18(2^{\mathsf{nat}(x-1)})$-13
BST* ab	31 (4)	26	31 (4)	19	7	0.9	$96(2^{\mathsf{nat}(x)})$-49

after PE). Note that PE greatly reduces $\#_{eq}$ in all benchmarks. Column **T** shows the total runtime in milliseconds. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running Linux.

The next four columns aim at demonstrating the scalability of our approach. To do so, we connect the CRSs for the different benchmarks by introducing a call from each CRS to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. Column $\#^c_{eq}$ shows the number of equations we want to solve in each case (in brackets after PE). Reading this column bottom-up, we can see that BST has the same number of equations as the original one and that, progressively, each benchmark adds its own number of equations to $\#^c_{eq}$. Thus, in the first row we have a CRS with all the equations connected, i.e., we compute an upper bound of CRS with at least 19 nested loops and 346 equations. The total runtime is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and it shows that even though PE is a *global* transformation, its time efficiency is linear with the number of equations. Our system solves 346 equations in $823ms$. Column **Rat**. shows the total time per equation. The ratio is small for benchmarks with few equations, and for reasonably large CRSs (from Delete upwards) it almost has no variation (2.1–2.4 ms/eq). The small increase is due to the fact that the equations count more complex expressions as we connect more benchmarks. This demonstrates that our approach is totally scalable, even if the implementation is preliminary. The upper bound expressions get considerably large when the benchmarks are composed together. We are currently implementing standard techniques for simplification of arithmetic expressions.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

1. Adachi, A., Kasai, T., Moriya, E.: A theoretical study of the time analysis of programs. In: Becvar, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 201–207. Springer, Heidelberg (1979)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Proc. of FMOODS. LNCS, Springer, Heidelberg (to appear, 2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
4. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile Resource Guarantees for Smart Devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
5. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, arXiv:cs/0512056 (2005), `http://arxiv.org/`
6. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
7. Benzinger, R.: Automated higher-order complexity analysis. In: TCS, vol. 318(1-2) (2004)
8. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations and small space bounds. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, Springer, Heidelberg (2005)
9. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.: Enforcing resource bounds via static verification of dynamic checks. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
11. Crary, K., Weirich, S.: Resource bound certification. In: POPL (2000)
12. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS 15(5) (1993)
13. Gómez, G., Liu, Y.A.: Automatic time-bound analysis for a higher-order language. In: PEPM, ACM Press, New York (2002)
14. Hickey, T., Cohen, J.: Automating program analysis. J. ACM 35(1) (1988)
15. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI, Elsevier, Amsterdam (2006)
16. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL (2003)
17. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
18. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. TOPLAS 10(2) (1988)

19. Marion, J.-Y., Péchoux, R.: Resource analysis by sup-interpretation. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, Springer, Heidelberg (2006)
20. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-definable resource bounds analysis for logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, Springer, Heidelberg (2007)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, Springer, Heidelberg (2004)
22. Rosendahl, M.: Automatic Complexity Analysis. In: FPCA, ACM Press, New York (1989)
23. Sands, D.: A naïve time analysis and its theory of cost equivalence. Journal of Logic and Computation 5(4) (1995)
24. Wadler, P.: Strictness analysis aids time analysis. In: POPL (1988)
25. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM 18(9) (1975)

Cost Analysis of Object-Oriented Bytecode Programs

ELVIRA ALBERT, PURI ARENAS and SAMIR GENAIM
Complutense University of Madrid
GERMAN PUEBLA and DAMIANO ZANARDINI
Technical University of Madrid

Cost analysis statically approximates the cost of programs in terms of their input data sizes. This paper presents, to the best of our knowledge, the first approach to the automatic cost analysis of object-oriented bytecode programs. In languages such as Java and C#, analyzing bytecode has a much wider application area than analyzing source code since the latter is often not available. Cost analysis in this context has to consider, among others, dynamic dispatch, jumps, the operand stack and the heap. The proposed method takes a bytecode program and a *cost model* specifying the resource of interest and generates *cost relations* which approximate the execution cost of the program w.r.t. such resource. Our basic techniques can be directly applied to infer cost relations for other object-oriented imperative languages, not necessarily in bytecode form. Finally, we report on COSTA, an implementation for Java bytecode which can obtain upper bounds on cost for a large class of programs and complexity classes, getting meaningful results.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Automatic analysis of algorithms*

General Terms: Complexity, Languages, Theory.

Additional Key Words and Phrases: Upper bounds, cost analysis, program complexity, bytecode.

1. INTRODUCTION

The *computational complexity theory* is a fundamental research area in computer science, which aims at determining the amount of resources required to run a given algorithm or to solve a given problem in terms of the input values. Computational complexity has received considerable attention since the early days of computer science. The most common metrics studied are *time-complexity* and *space-complexity*, which measure, respectively, the time and memory required for running an algorithm or solving a problem. In complexity theory, algorithms and problems are

Author's address: Elvira Albert / Puri Arenas / Samir Genaim. Facultad de Informática, Universidad Complutense de Madrid, E-28.040, Madrid (Spain).
Germán Puebla / Damiano Zanardini. Facultad de Informática, Universidad Politécnica de Madrid, E-28.660, Boadilla del Monte (Spain).

Submitted to ACM Transactions on Programming Languages

often categorized into *complexity classes*, according to the amount of resources required for executing the algorithm or solving the problem by using the best possible algorithm. Though complexity theory has produced a wealth of research results, especially in the last decades, assigning a complexity class to an algorithm is still far from being automatic, and requires significant human intervention.

In this work, rather than on the complexity of problems or algorithms, we concentrate on analyzing the complexity of *programs*, i.e., a particular implementation of a given algorithm. The first proposal for *automatically* computing the complexity of programs was the seminal work of Wegbreit [1975], whereby the *Metric* system is described, together with a number of applications of automatic cost analysis. This system was able to automatically compute *closed-form cost functions* which capture the non-asymptotic cost of simple Lisp programs as functions of the size of the input arguments (using different *measures*). Since then, a number of cost analysis frameworks have been proposed, mostly in the context of *declarative* programming languages (functional programming [Le Metayer 1988; Rosendahl 1989; Wadler 1988; Sands 1995; Benzinger 2004] and logic programming [Debray and Lin 1993; Navas et al. 2007]). Cost analysis of imperative programming languages has received significantly less attention. It is worth mentioning the pioneering work of Adachi et al. [1979] and, for Java-like languages, recent work focused on estimating memory consumption [Braberman et al. 2008; Chin et al. 2008].

Traditionally, cost analysis has been formulated at the *source code* level. However, it can be the case that the analysis must consider the *compiled code* instead. This may happen, in particular, when the *code consumer* is interested in verifying some properties of 3rd party programs, but has no direct access to the source code, as usual for commercial software and in mobile code. This is the general picture where the idea of *Proof-Carrying code* [Necula 1997] was born: in order for the code to be verifiable by the user, security properties (including resource usage limitations) must refer to the (compiled) code available to the user, so that it is possible to check the provided proof and verify that the program satisfies the requirements (e.g., that the code does not require more than a certain amount of memory, or that it executes in less than a certain amount of time).

In the context of mobile code, programming languages which are compiled to *bytecode* and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET. The execution model based on virtual machines has two important benefits when compared to classical machine code. First, bytecode is *platform-independent*. Therefore, the same compiled code can be run on multiple platforms. Second, since the virtual machine is not directly executed on the hardware, it is possible to apply a *sand-box* model which guarantees that the bytecode does not have access to certain assets of the platform unless the code is explicitly granted access to them.

1.1 Applications of Cost Analysis of Object-Oriented Bytecode Programs

We now summarize some of the applications of cost analysis in general and of cost analysis of object-oriented bytecode in particular:

—*Resource Bound Certification.* This research area deals with security properties involving resource usage requirements; i.e., the (untrusted) code must adhere to

specific bounds on its resource consumption. *Abstraction-Carrying Code* [Albert et al. 2008] (ACC) proposes the use of static analysis as enabling technology for mobile code certification. In particular, in ACC, the safety policies are defined over different abstract domains. The main idea is that the *abstraction* (or abstract model) of the program computed by standard static analyzers is used as a certificate. Then, the validity of the abstraction on the consumer side is checked in a single pass by a very efficient and specialized abstract-interpreter. Within the ACC framework, the present work suggests that it is possible to automatically generate non-trivial resource usage bounds for a realistic programming language. Such bounds could then be translated to *certificates*, in the proof-carrying code style. Previous work in this direction was restricted to *linear bounds* [Crary and Weirich 2000; Aspinall et al. 2005; Hofmann and Jost 2003], to *semi-automatic techniques* [Chander et al. 2005], or to source code [Albert et al. 2008]. It is important to note that mobile code (where bytecode is widely used) is one of the settings where estimating the cost of programs and guaranteeing upper bounds it is most important, because of the limited computing power typically available on mobile devices. Nevertheless, an actual use of resource bounds certificates will require the overall design of the framework and of the certificate checkers. This remains as a challenge for the future.

—*Performance Debugging and Validation.* This is a direct application of resource usage analysis, where the analyzer tries to verify or falsify *assertions* about the efficiency of the program which are written by the programmer. This application was already mentioned as future work in [Wegbreit 1975] and is available in the CiaoPP system for Prolog programs [Hermenegildo et al. 2005]. In our context, and when the compiled code is not obfuscated, assertions can possibly refer to the source code level, but it is straightforward to translate them to be understandable by our compiled code analyzer. In the other direction, results obtained by our analysis performed on the compiled code can be easily related to the source program.

—*Granularity Control.* Parallel computers have recently become mainstream with the massive use of *multicore* processors. In parallel systems, knowledge about the cost of different procedures in the code can be used in order to guide the partitioning, allocation and scheduling of parallel processes [Debray and Lin 1993; Hermenegildo et al. 2005].

—*Program Synthesis and Optimization.* This application was already mentioned as one of the motivations for [Wegbreit 1975]. Both in program synthesis and in semantic-preserving optimizations, such as partial evaluation (see e.g. [Craig and Leuschel 2005; Puebla and Ochoa 2006]), there are multiple programs which may be produced in the process, with possibly different efficiency levels. Here, automatic cost analysis can be used for guiding the selection process among a set of candidates.

1.2 Summary of Contributions

As its main contribution, the present work formulates an automatic approach to cost analysis of real-life, object-oriented bytecode programs (from now on, we use *bytecode* for short), whose features include the most important difficulties encoun-

tered when analyzing both (source) object-oriented languages and low-level code: (1) from low-level code, bytecode inherits its *unstructured control flow*: execution flow is modified using conditional and unconditional *jumps*; (2) from object-oriented languages, bytecode inherits features such as *virtual method invocation*, extensive usage of exceptions, and the use of a *heap*; and (3) an additional difficulty present in bytecode (but not necessarily in other low-level languages) is the use of an *operand stack* for storing the intermediate results of computations.

As will be presented in detail in the rest of the paper, our analysis takes as input the bytecode corresponding to a program, the cost model of interest, and yields a set of recursive equations which capture the cost of the program by performing the following steps:

1. A *control flow graph* (CFG for short) is generated for each method in the original bytecode program by using techniques from *compiler theory* [Aho et al. 1974; Aho et al. 1986]. Advanced features like virtual invocation and exceptions are simply handled as additional nodes and arcs in the graph.

2. Each CFG is represented as a set of procedures (composed of one or more rules) by using a *rule-based*, recursive *representation* (RBR for short), where the arguments to the bytecode instructions are made explicit, and the operand stack is *flattened* by converting its contents into additional local variables.

3. Static analysis infers, for each rule, *size relations* among the input variables to the rule and the variables in all calls in the rule. Size relations are, in the case of integer variables, constraints on the value of variables, and, in the case of references, constraints on their *path length*, i.e., the length of the longest reference chain reachable from the given reference [Spoto et al. 2006a].

4. A parametric notion of *cost model* is introduced, which allows describing how the resource consumption associated to a program execution should be computed. A cost model defines how cost is assigned to each execution step and, by extension, to an entire execution trace. We consider a range of non-trivial cost models for measuring different aspects of computations.

5. From the RBR, the size relations, and a given cost model, a *cost relation system* (CRS for short) is automatically obtained. Cost relations express the cost of any block in the CFG (or rule in the RBR) in terms of the cost of the block itself plus the cost of its successors.

6. If possible, an exact solution or an upper bound in non-recursive, i.e., *closed-form* is found for the cost relation system. This step requires the use of a solver for such systems, whose detailed description is not in the scope of this paper. See e.g. [Albert et al. 2008a] for more details.

As another contribution, the present work reports on the COSTA system: an implementation of our proposed framework for *Java bytecode* (JBC), which is one of the most widely used languages in *mobile code* architectures, and one of the candidates for building a realistic *proof-carrying code* framework for software verification. The implementation considers *exceptions*, though it is restricted to sequential JBC.

COSTA [Albert et al. 2008a; 2008b] computes an upper bound for the cost of JBC programs, by relying on a series of static analyses, such as *class*, *sign* and *nullity* analysis, *slicing* of irrelevant variables, *sharing* and *cyclicity* analysis, etc.

Moreover, closed-form solutions are found by means of a dedicated solver [Albert et al. 2008a]. By means of such techniques, COSTA can deal with quite a large class of JBC programs.

Finally, though our method is formalized at the level of bytecode, the basic techniques developed in the paper could be directly applied to cost analysis of other object-oriented imperative languages, not necessarily in low-level form. Besides, since Java is compiled to bytecode, then our work also constitutes the first approach to the automatic cost analysis of Java programs.

1.3 Organization of the Article

This article is organized as follows. The next section briefly describes the bytecode language we consider and its semantics. Section 3 is devoted to present the transformation from bytecode programs to rule-based programs. First, the syntax of the RBR language is described. The definition of CFG for a bytecode method and its RBR transformation are then formally presented, together with the semantics of the RBR. Its equivalence to the original bytecode program is proven. The resulting RBR can be useful in itself for developing various static analyses (not necessarily cost) on the bytecode in a much simpler way, as it will become apparent later from the simplicity of the size and cost analyses that we define on the RBR.

Section 4 formalizes the notion of cost that the proposed analysis aims at approximating. It also presents examples of cost models. Section 5 introduces the proposed cost analysis in two main steps. First, Sections 5.1, 5.2 and 5.3 present the components of the size analysis which allow inferring precise size relations between the execution states at different program points. Then, the generation of CRSs for bytecode programs is introduced in Section 5.4 and the correctness of the resulting analysis is proven.

Section 6 discusses a series of practical issues related to the simplification and solving of the CRS. In particular, Section 6.1 discusses the notion of cost-relevant variables, and sketches a static analysis to infer them. Section 6.2 states the differences between CRS and traditional recurrence relation systems (RRs for short). Then, Section 6.3 overviews the existing technology to compute *closed form* (i.e., non recursive) solutions or upper bounds both from CRSs and RRs.

Section 7 describes our implementation, the COSTA system, and demonstrates its practicality on a series of experiments. With this aim, COSTA infers the cost of a set of selected benchmarks illustrating object-oriented features and using Java libraries. Section 8 reviews some related work, and Section 9 concludes the article.

2. OBJECT-ORIENTED BYTECODE

In order to simplify the formalization of our analysis, a simple sequential object-oriented bytecode language is considered, which roughly corresponds to a representative subset of Java bytecode. We refer to it as *simple bytecode*. For short, unless we explicitly mention *Java* bytecode, all references to bytecode in the rest of the paper correspond to our simple bytecode. Simple bytecode is able to handle integers and object creation and manipulation. For simplicity, simple bytecode does not include advanced features of Java bytecode, such as exceptions, interfaces, static methods and fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and references. Anyway, such

```
class A {                  1: load 1
  int incr(int i) {        2: push 1
    return i+1;            3: add
  }                        4: return
}

class B extends A {        1: load 1
  int incr(int i) {        2: push 2
    return i+2;            3: add
  }                        4: return
}

class C extends B {        1: load 1
  int incr(int i) {        2: push 3
    return i+3;            3: add
  }                        4: return
}

class Main {               1: push 0       12: load 2
  int add(int n,A o) {     2: store 3      13: load 4
    int res=0;             3: push 0       14: invokevirtual A.incr:(I)I
    int i=0;               4: store 4      15: store 4
    while (i<=n) {         5: load 4       16: goto 5
      res=res+i;           6: load 1       17: load 3
      i=o.incr(i);         7: ifgt 17      18: return
    }                      8: load 3
    return res;            9: load 4
  }                       10: add
}                         11: store 3
```

Fig. 1. A Java source (left) with its corresponding bytecode (right)

features can be easily handled in this framework, as shown by the implementation presented in Section 7, based on actual Java bytecode.

2.1 The Syntax

A bytecode program consists of a set of *classes* $\mathcal{C}$, partially ordered with respect to the *subclass* relation $\preceq$. Each class $c \in \mathcal{C}$ contains information about the class $c' = \mathit{extends}(c) \in \mathcal{C} \cup \{\mathsf{none}\}$ it extends, the fields $\mathit{fields}(c)$ and the methods $\mathit{methods}(c)$ it declares. Subclasses inherit all the fields and methods of the class they extend. The special class name none is used as in Java bytecode when there is no super class (e.g., for the class Object). Each method comes with a *signature* m which consists of the class $\mathit{class}(m) \in \mathcal{C}$ where it is defined, its name $\mathit{name}(m)$ and its type $\mathit{type}(m) = \tau_1, \ldots, \tau_k \to \tau$, where $\tau_i \in \mathcal{C} \cup \{\mathit{int}\}$ is the type of its i-th argument, and $\tau \in \mathcal{C} \cup \{\mathit{int}\}$ is the type of its return value (for simplicity, all methods are supposed to return a value). There cannot be two methods with the same signature. In addition, the name init is reserved for the class instance *initialization* method. The bytecode associated to a method m is denoted $\mathit{bc}(m)$, and it is a sequence $\langle b_1, \ldots, b_n \rangle$ where each b_i is a bytecode instruction. We use $m[i]$ to denote the i-th instruction in $\mathit{bc}(m)$. Local variables of a k-ary method are denoted by $\langle l_0, \ldots, l_n \rangle$ with $n \geq k$, from which $\langle l_1, \ldots, l_k \rangle$, are the formal parameters, l_0 corresponds to the *this* reference and the remaining $\langle l_{k+1}, \ldots, l_n \rangle$ are the local variables declared in the method. Similarly, each field f has a unique signature which consists of the class $\mathit{class}(f) \in \mathcal{C}$ where it is declared, its name $\mathit{name}(f)$ and its type $\mathit{type}(f) \in \mathcal{C} \cup \{\mathit{int}\}$.

A class cannot declare two fields with the same name. The following instructions are included:

$$\begin{aligned} \mathit{bcInstr} ::= \quad & \mathsf{load}\ i \mid \mathsf{store}\ i \mid \mathsf{push}\ n \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{add} \mid \mathsf{sub} \\ \mid\ & \mathsf{iflt}\ j \mid \mathsf{ifgt}\ j \mid \mathsf{ifeq}\ j \mid \mathsf{ifnull}\ j \mid \mathsf{goto}\ j \\ \mid\ & \mathsf{new}\ c \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f \\ \mid\ & \mathsf{newarray}\ d \mid \mathsf{aload} \mid \mathsf{astore} \mid \mathsf{arraylength} \\ \mid\ & \mathsf{invokevirtual}\ m \mid \mathsf{invokenonvirtual}\ m \mid \mathsf{return} \end{aligned}$$

Similarly to Java bytecode, simple bytecode is a stack-based language. The instructions in the first row manipulate the *operand stack* (see the semantics below). The second row contains *jump* instructions. Instructions in the third row manipulate *objects*, while the fourth row works on *arrays*, where $d \in \mathcal{C} \cup \{int\}$. The last row contains instructions dealing with *method invocation*. As regards notation, i is an integer which corresponds to a local variable index, n is an integer or null, j is an integer which corresponds to an index in the bytecode sequence, $c \in \mathcal{C}$, m is a method signature, and f is a field signature.

EXAMPLE 2.1. *(running example) Figure 1 depicts the bytecode and a possible Java source (only shown for clarity of the presentation) of our running example. The program consists of four classes.* `A`, `B` *and* `C` *are three related classes which provide different implementations for the* `incr` *method, which returns the result of increasing an integer by a different amount. The* `Main` *class defines a method named* `add` *which receives an integer value* `n` *and an object* `o`*, and computes (1)* $\Sigma_{i=0}^{n} i$ *if the runtime class of* `o` *is* `A`*; (2)* $\Sigma_{i=0}^{\lfloor n/2 \rfloor} 2i$ *if the runtime class is* `B`*; or (3)* $\Sigma_{i=0}^{\lfloor n/3 \rfloor} 3i$ *if the runtime class is* `C`*. The first four instructions of* `add` *initialize local variables* `res` *and* `i` *to the value* 0*, corresponding, respectively, to indexes 3 and 4 in the table of local variables. Instructions 5, 6 and 7 check the loop condition. If the test fails, then control goes to the end of the loop (instruction 17). Otherwise, the value of* `i` *is accumulated in* `res` *in the next four instructions. Afterwards, before invoking* `incr`*, the reference to* `o` *(local variable 2) and the variable* `i` *are pushed on the stack (instructions 12-13). Depending on the runtime instance in* `o`*, the corresponding method is invoked at instruction 14, the return value is stored in* `i` *at 15, and control goes back to the loop entry (at 16). The last two bytecodes return the result.* □

2.2 The Operational Semantics

This section presents the operational semantics of simple bytecode. A *heap* h is a partial map from an infinite set of *memory locations* to *objects*. We use *Heaps*, *Locations*, and *Objects* to denote the sets of all heaps, memory locations, and objects, respectively. For simplicity, an array of length n is modeled as an object o with a special (read-only) field *length* initialized to n, and fields $1, \ldots, n$ of type d which correspond to the array elements. Moreover, we assume that the class tag of an object contains information from which it is possible to distinguish array objects from class instance objects. Given $h \in \mathit{Heaps}$ and $r \in \mathit{Locations}$, we use $h(r)$ to denote the object referred to by r in h. We use $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$ while h stays the same for all locations different from r. Note that for any location r and heap h, $r \in \mathit{dom}(h)$ iff there is an object associated to r in h. A *value* v belongs to the set $\mathbb{Z} \cup \{\mathsf{null}\} \cup \mathit{Locations}$. An

$$(1)_{bc} \quad \frac{m[pc] \equiv \mathsf{load}\ i}{\langle m, pc, lv, stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, lv(i){\cdot}stk\rangle{\cdot}ar; h}$$

$$(2)_{bc} \quad \frac{m[pc] \equiv \mathsf{store}\ i}{\langle m, pc, lv, v{\cdot}stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv[i{\mapsto}v], stk\rangle{\cdot}ar; h}$$

$$(3)_{bc} \quad \frac{m[pc] \equiv \mathsf{push}\ n}{\langle m, pc, lv, stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, n{\cdot}stk\rangle{\cdot}ar; h}$$

$$(4)_{bc} \quad \frac{m[pc] \equiv \mathsf{pop}}{\langle m, pc, lv, v{\cdot}stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, stk\rangle{\cdot}ar; h}$$

$$(5)_{bc} \quad \frac{m[pc] \equiv \mathsf{dup}}{\langle m, pc, lv, v{\cdot}stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, v{\cdot}v{\cdot}stk\rangle{\cdot}ar; h}$$

$$(6)_{bc} \quad \frac{(m[pc] \equiv \mathsf{add} \wedge v = v_1 + v_2) \vee (m[pc] = \mathsf{sub} \wedge v = v_2 - v_1)}{\langle m, pc, lv, v_1{\cdot}v_2{\cdot}stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, v{\cdot}stk\rangle{\cdot}ar; h}$$

$$(7)_{bc} \quad \frac{m[pc] \equiv \mathsf{new}\ c, \quad o = \mathsf{newobject}(c), \quad r \notin dom(h)}{\langle m, pc, lv, stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, r{\cdot}stk\rangle{\cdot}ar; h[r{\mapsto}o]}$$

$$(8)_{bc} \quad \frac{m[pc] \equiv \mathsf{getfield}\ f, \quad r \neq \mathsf{null}, \quad v = h(r).f,}{\langle m, pc, lv, r{\cdot}stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, v{\cdot}stk\rangle{\cdot}ar; h}$$

$$(9)_{bc} \quad \frac{m[pc] \equiv \mathsf{putfield}\ f, \quad r \neq \mathsf{null}, \quad o = h(r),}{\langle m, pc, lv, v{\cdot}r{\cdot}stk\rangle{\cdot}ar; h \leadsto_{\mathsf{bc}} \langle m, pc{+}1, lv, stk\rangle{\cdot}ar; h[o.f{\mapsto}v]}$$

Fig. 2. Operational Semantics of the Bytecode Language (1)

object o is a pair consisting of the object class tag and a mapping from field names to values which is consistent with the signatures of the fields. The class tag of o is denoted by $class(o)$, $o.f$ refers to the value of the field f in o, and $o[f{\mapsto}v]$ sets the value of $o.f$ to v. Finally, $h[o.f{\mapsto}v]$ is a shortcut for $h(r)[f{\mapsto}v]$ with $o = h(r)$.

An *activation record* a is a tuple of the form $\langle m, pc, lv, stk\rangle$, where: m is a method *signature*; pc is the *program counter*, i.e., the address of the instruction to be executed; stk is an *operand stack* of values (ϵ denotes the empty stack); and lv is a mapping from local variable indexes to values. Given an index i, $lv(i)$ refers to the value of l_i, and $lv[i{\mapsto}v]$ updates lv by making $lv(i) = v$ while lv remains the same for all indexes different from i. Different activation records do not share information, but may contain references to the same objects in the heap.

An execution state or *configuration* C takes the form $ar; h$, where ar is a stack of activation records, and h is a global heap. Given a configuration C, the operational semantics (see Figures 2 and 3) uniquely determines how to perform a step and produce the next configuration C'. The transition is denoted by $C \leadsto_{\mathsf{bc}} C'$.

Executions start from an initial configuration $C_0 = \langle start, 1, lv, v_n \cdots v_1\rangle; h$, where: *start* is an auxiliary method (not part of the program) that has only one instruction (at *start*[1]) which is a non-virtual call to the actual method that we want to start the execution from (e.g., *main* in Java and Java bytecode); lv is an empty variable mapping (since *start* is just an auxiliary method); $v_n, \ldots, v_1$ are the initial stack elements where v_1 corresponds to the object whose method

$$(10)_{bc} \quad \frac{m[pc] \equiv \mathsf{newarray}\ d, \quad n \geq 0, \quad o = \mathsf{newarray}(d,n), \quad r \notin dom(h)}{\langle m, pc, lv, n{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc{+}1, lv, r{\cdot}stk\rangle{\cdot}ar; h[r{\mapsto}o]}$$

$$(11)_{bc} \quad \frac{m[pc] \equiv \mathsf{aload}, \quad r \neq \mathsf{null}, \quad o = h(r), \quad 1 \leq i \leq o.length, \quad v = o.i}{\langle m, pc, lv, i{\cdot}r{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc{+}1, lv, v{\cdot}stk\rangle{\cdot}ar; h}$$

$$(12)_{bc} \quad \frac{m[pc] \equiv \mathsf{astore}, \quad r \neq \mathsf{null}, \quad o = h(r), \quad 1 \leq i \leq o.length}{\langle m, pc, lv, v{\cdot}i{\cdot}r{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc{+}1, lv, stk\rangle{\cdot}ar; h[o.i \mapsto v]}$$

$$(13)_{bc} \quad \frac{m[pc] \equiv \mathsf{arraylength}, \quad r \neq \mathsf{null}, \quad o = h(r), \quad n = o.length}{\langle m, pc, lv, r{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc{+}1, lv, n{\cdot}stk\rangle{\cdot}ar; h}$$

$$(14)_{bc} \quad \frac{\begin{array}{c}((m[pc]{\equiv}\mathsf{invokevirtual}\ m' \ \wedge\ m''{=}lookup(m', class(h(v_0)))) \vee \\ (m[pc]{\equiv}\mathsf{invokenonvirtual}\ m'')), \\ v_0 \neq \mathsf{null}, \quad lv' = newenv(m''), \quad \forall i \in [0 \ldots k].\ lv'[i{\mapsto}v_i]\end{array}}{\langle m, pc, lv, v_k \cdots v_1{\cdot}v_0{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m'', 1, lv', \epsilon\rangle{\cdot}\langle m, pc+1, lv, stk\rangle{\cdot}ar; h}$$

$$(15)_{bc} \quad \frac{m[pc] \equiv \mathsf{return}}{\langle m, pc, lv, v{\cdot}stk\rangle{\cdot}\langle m', pc', lv', stk'\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m', pc', lv', v{\cdot}stk'\rangle{\cdot}ar; h}$$

$$(16)_{bc} \quad \frac{\begin{array}{c}(m[pc] \equiv \mathsf{ifgt}\ pc' \wedge (v_2 > v_1)) \vee \\ (m[pc] \equiv \mathsf{iflt}\ pc' \wedge (v_2 < v_1)) \vee (m[pc] \equiv \mathsf{ifeq}\ pc' \wedge (v_2 = v_1))\end{array}}{\langle m, pc, lv, v_1{\cdot}v_2{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc', lv, stk\rangle{\cdot}ar; h}$$

$$(17)_{bc} \quad \frac{\begin{array}{c}(m[pc] \equiv \mathsf{ifgt}\ pc' \wedge (v_2{\leq}v_1)) \vee \\ (m[pc] \equiv \mathsf{iflt}\ pc' \wedge (v_2{\geq}v_1)) \vee (m[pc] \equiv \mathsf{ifeq}\ pc' \wedge (v_2 \neq v_1))\end{array}}{\langle m, pc, lv, v_1{\cdot}v_2{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc{+}1, lv, stk\rangle{\cdot}ar; h}$$

$$(18)_{bc} \quad \frac{(m[pc] \equiv \mathsf{ifnull}\ pc', \quad ((r = \mathsf{null} \wedge pc'' = pc') \vee (r \neq \mathsf{null} \wedge pc'' = pc{+}1)))}{\langle m, pc, lv, r{\cdot}stk\rangle{\cdot}ar; h \rightsquigarrow_{\mathsf{bc}} \langle m, pc'', lv, stk\rangle{\cdot}ar; h}$$

Fig. 3. Operational Semantics of the Bytecode Language (2)

we want to call, and $v_2, \ldots, v_n$ are the actual parameters to that method (v_2 the first parameter); and h is an initial heap. Note that each v_i might be an integer, null or a reference to an object in the initial heap h. A configuration is *final* iff it takes the form $\langle start, 2, lv, v\rangle; h$ where v is the return value of the call at $start[1]$ and h is the final heap (note that the return instruction increases the program counter by 1). Execution proceeds deterministically by applying the execution step corresponding to the instruction in $m[pc]$, and ends when a final configuration is reached. When there is no rule to apply (e.g., dereferencing a null pointer), we assume that the execution stops. Thus, executions can be regarded as *traces* of the form $C_0 \rightsquigarrow_{\mathsf{bc}} C_1 \rightsquigarrow_{\mathsf{bc}} \cdots \rightsquigarrow_{\mathsf{bc}} C_\omega$, where C_ω is a final configuration. Non-terminating executions have infinite traces.

As usual in Java bytecode [Lindholm and Yellin 1996], we assume that code is verified prior to execution. This guarantees that the following conditions are satisfied by any program: (1) addresses in conditional and unconditional jumps are within the bounds of the method code; (2) execution cannot fall off the end of the code; (3) if the code refers to a field signature f, then a field with such signature must be defined either in $class(f)$ or in one of its super classes; (4) if the code refers to a method signature m in invokenonvirtual then a method with the corresponding signature must be defined in $class(m)$ (not in its super classes); and (5) the height

of the operand stack and the types of its elements (reference or integer type) at each program point are fixed, regardless of the path execution comes from and (6) the instructions operate on variables of the required type (for instance, add requires that both operands be integers). Thus, after the verification, the types of the elements are statically known.

The operational semantics shown in Figures 2 and 3 is quite standard. As an example, consider rule $(8)_{bc}$ for reading fields: when the instruction at pc is a getfield f, then the value v corresponding to the field f of the object referred to by r in h is pushed on the stack (thus resulting in $v{\cdot}stk$), and the program counter is increased by 1 ($pc{+}1$).

For the rest of the rules, we just point out that, in the execution step corresponding to method invocation (rule $(14)_{bc}$), it is assumed that: (1) *lookup* is defined in a standard object-oriented fashion, i.e., it looks up for an implementation of the method in the signature m' (i.e., removing the name of the class from the signature), starting from the class of the corresponding object and going iteratively into its super classes; it fails if it does not find such a method; (2) *newenv* creates a new mapping of local variables for the corresponding method, where each variable is initialized to either 0 or null, depending on its type. Similarly, in rule $(7)_{bc}$, newobject(c) creates a new object of class c by initializing its fields as *newenv* does, while, in rule $(10)_{bc}$, newarray(d, n) creates an array of n elements initialized to 0 or null depending on d.

3. FROM BYTECODE TO THE RULE-BASED REPRESENTATION

The *control flow* of a program is an essential piece of information in order to reason about cost. It allows reasoning about all possible paths which might be taken during the execution. Also, it allows one to group together sequences of instructions which are always executed as a *block*, i.e., in a non-dividable fashion: either all or none of the instructions in the sequence are executed and the instructions are processed in exactly the order in which they appear in the sequence.

Unlike programs written in high-level, structured programming languages as Java, bytecode programs feature *unstructured* control flows, since conditional and unconditional *jumps* are allowed instead of the if-then-else, switch and loop constructs available in structured programming languages. Reasoning about unstructured programs is more complicated for both human and automatic analysis. Moreover, in a realistic object-oriented programming language, *virtual method invocation* and *exception handling* make the control flow even more difficult to deal with.

In Section 3.1 we introduce a rule-based *structured* language which is rich enough to allow the (de-)compilation of bytecode programs to it (and preserve the information about cost), while staying simple enough to develop a precise cost analysis, since object-oriented features are compiled away and recursion is the only iterative construct. The compilation of bytecode programs into this language is described in Section 3.2. The equivalence with the original bytecode is formalized in Section 3.3. It is important to note that our aim here is not to decompile bytecode back to Java, but rather to a representation which is as amenable as possible to static analysis.

3.1 The Rule-Based Representation

This section introduces the syntax of the RBR as a simple, *structured* procedural language which will be used to develop the cost analysis framework in the rest of the article. The following key features of the RBR will make the development of the analysis easier, as they allow the control flow to be simple and explicit:

(1) recursion becomes a kind of iteration;
(2) there is only one form of conditional construct: the use of *guarded rules*;
(3) there is only one kind of variables, the *local variables*, and there is no stack;
(4) object-oriented features are gone:
—objects can be simply regarded as records which include an additional field which contains their type;
—the behaviour induced by dynamic dispatch is compiled into *dispatch blocks*;
(5) there is no distinction between executing a method and executing a block.

As will be clear later, these choices are not arbitrary. Indeed, they are designed in order to make the generation of cost relation systems –whose solution is classically the goal of cost analysis– feasible, and consistent with the program structure. In order to cover object-oriented features, the language also supports object creation, field manipulation and arrays.

A *rule-based representation* consists of a set of (global) *procedure*s. A procedure p with k input arguments $\bar{x}$ and a single output argument y is defined by a set of *guarded rules*, where each rule is defined according to the following grammar:

$$
\begin{array}{rcl}
\mathit{rule} & ::= & p(\bar{x}, y) \ \leftarrow g, \mathit{body}. \\
g & ::= & \mathit{true} \mid \mathit{exp}_1 \ \mathit{op} \ \mathit{exp}_2 \mid \mathsf{type}(x, c) \\
\mathit{body} & ::= & \epsilon \mid b, \mathit{body} \\
b & ::= & x := \mathit{exp} \mid x := \mathsf{new}\ c \mid x := y.f \mid x.f := y \mid x := \mathsf{newarray}(d, y) \mid \\
 & & x[i] := y \mid x := y[i] \mid x := \mathsf{arraylength}(y) \mid \mathsf{nop}(\mathit{any}) \mid q(\bar{x}, y) \\
\mathit{exp} & ::= & x \mid \mathsf{null} \mid n \mid x{-}y \mid x{+}y \\
\mathit{op} & ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{array}
$$

where $p(\bar{x}, y)$ is the *head* of the rule, and $\bar{x} = x_1, \ldots, x_k$. Note that the last argument is always the output argument. The identifiers $x_1, \ldots, x_k$ and y (possibly subscripted or primed) are taken from a valid set of variable names; n is an integer; i is a non-negative integer; $d \in \mathcal{C} \cup \{\mathit{int}\}$, $q(\bar{x}, y)$ is a procedure call (by value); and $\mathsf{nop}(\mathit{any})$ is an auxiliary instruction which takes *any* instruction in *bcInst* as argument, and has no effect on the semantics (but is useful for preserving some information about the original bytecode program). In the following, we use *rrInstr* to denote the set of guards and instructions which can appear in the body of the rules. Note that though more readable than bytecode, all guards and instructions correspond to three-address code, as in bytecode, except for calls to procedures (i.e., of the form $q(\bar{x}, y)$).

As it will be described in Section 3.2 below, the class hierarchy of the bytecode program is used, together with class analysis, in order to generate the required dispatch blocks. Then in the RBR the class hierarchy is no longer needed for execution. For simplicity, it is assumed that there are no two procedures with the

same name and different number of arguments, and that rules that correspond to the same procedure uses the same names for their input and output variables. Furthermore, we restrict ourselves to RBR programs which are *strictly deterministic*, i.e., the guards for all rules for the same procedure are pairwise mutually exclusive and the disjunction of all guards is always true (the guards cover all possible cases). This will always be the case for RBR programs compiled from bytecode programs as described in Section 3.2 below.

EXAMPLE 3.1. *The following procedure, defined by two guarded rules, implements the* fib *method, which computes the n-th number of the Fibonacci series.*

$$\begin{array}{lll} \mathsf{fib}(n,y) & \leftarrow & n{>}1,\ n_1 := n{-}1,\ \mathsf{fib}(n_1,y'),\\ & & \qquad n_2 := n{-}2,\ \mathsf{fib}(n_2,y''),\ y := y'{+}y''.\\ \mathsf{fib}(n,y) & \leftarrow & n{\leq}1,\ y := 1. \end{array}$$

Note that the variable y stands for the return value of the method. □

The next section shows that bytecode programs can be translated into the above syntax, thus enjoying the five features mentioned previously. Executing an RBR program still needs a *heap* and a *stack* of activation records, similarly to the bytecode.

3.2 Compiling Bytecode Programs to the Rule-Based Representation

The translation of a bytecode program to a RBR is performed in two steps. First, a *control flow graph* is built from the bytecode program, which recovers the structure of all possible explicit and implicit branching and loops. In the second step, a *procedure* is defined for each basic block in the graph, and the operand stack is *flattened* by considering its elements as additional local variables.

The notion of control flow graph (CFG) is well-known, and makes it easier to reason about programs in unstructured languages [Aho et al. 1986]. CFGs are similar to *flow charts*, except that they include the notions of *calls to* and *returns from*. A method in the bytecode program is represented as a CFG[1], and *calls* from a method to another one correspond to calls between graphs.

In order to build the CFG of a method m, with its corresponding instruction sequence $\langle b_1,\ldots,b_n\rangle$, the first step is to partition $\langle b_1,\ldots,b_n\rangle$ into a set of *basic blocks*. An instruction b_j is said to be a *predecessor* of b_i if one of the following conditions holds: (1) $b_j{=}\mathsf{goto}\ i$; (2) $b_j{=}\mathsf{if}\phi\ i$; (3) $i{=}j{+}1$ and $b_j \neq \mathsf{goto}\ i'$. The following definition introduces the notion of partition into basic blocks.

DEFINITION 3.2. *(partition into basic blocks) Given a method m and its instruction sequence* $bc(m) = \langle b_1,\ldots,b_n\rangle$, *a* partition *into basic blocks* $m_{i_1},\ldots,m_{i_k}$ *takes the form*

$$\underbrace{b_{i_1},\ldots,b_{f_1}}_{m_{i_1}},\underbrace{b_{i_2},\ldots,b_{f_2}}_{m_{i_2}},\ldots\underbrace{b_{i_k},\ldots,b_{f_k}}_{m_{i_k}}$$

[1]In the implementation, a method can be represented by means of a set of graphs, due to the *loop extraction* transformation described later in Section 7.1.

where i_1=1, f_k=n, $i_j = f_{j-1}+1$ and (1) in each basic block m_{i_j}, only the instruction b_{f_j} can be a jump or an invocation; and (2) in each m_{i_j}, only the first instruction b_{i_j} can have more than one predecessor. □

Clearly, if we consider each instruction as a separate block, both conditions above are satisfied. However, we are interested in obtaining basic blocks which are as large as possible, resulting in an optimal partition. We can obtain an optimal partition to basic blocks as follows: the first sequence m_{i_1} starts at address 1 and ends at f_1=$min(e, s)$, where e is the address of the first non-sequential instruction (i.e., (un)conditional jump or method invocation), and s is the first address such that the instruction at address s+1 has one or more predecessors different from b_s. The sequence m_{i_2} is computed similarly starting at $i_2 = f_1+1$, and so on. This partition can be computed by going twice through the bytecode: the first pass computes the predecessors of each bytecode, and the second one defines the beginning and end of each sub-sequence.

Due to dynamic method resolution, in the case of invokevirtual m, the actual method to be called is only known at runtime. The compilation to RBR is made easier by approximating this information and introducing it explicitly in the CFG. This is done by adding new blocks, which are called *dispatch blocks*, containing calls to the *actual methods* which might be called at runtime. In addition, the access to these blocks is guarded by mutually exclusive conditions on the runtime class of the object whose method is called.

DEFINITION 3.3. *(dispatch block) Let m be a method containing a bytecode b_i =* invokevirtual *m' and let s be the stack element that contains the receiver. Let C be a superset of the runtime classes of the objects that s points to. The* dispatch block *for $c \in C$ is $m_{i:c}$, and it contains the single instruction* invoke(m'') *where $m'' = lookup(m', c)$.* □

Note that the set C in the above definition can be approximated statically by $C = \{c \mid c \preceq class(m')\}$, which is clearly a safe approximation of the set of the actual classes of the objects that s might point to. In some cases, this might result in a larger set than the actual one, which in turn affects the corresponding static analysis precision and performance. Class analysis [Spoto and Jensen 2003] is usually applied in such cases to reduce this set, as it gives information about the possible runtime classes of the object whose method is being called. Although the invokenonvirtual bytecode instruction is different from invokevirtual as it always corresponds to only one possible method call, in order to simplify the presentation, it is treated as invokevirtual, and associated to a single *dispatch block* with *true* guard. Note that invoke(m'') does not appear in the original bytecode; yet, it will be used to define the compilation to the RBR. Similarly, some instructions are wrapped with nop. These instructions will not be considered for building execution paths in the CFG, but only for inferring the cost of the original program.

The notion of CFG is defined below. As regards notation, $j = BlockId(i, m)$ indicates that the instruction b_i in m belongs to the block m_j. In addition, for a given invokevirtual instruction b_i, M_i^m denotes the set of its *dispatch blocks*.

DEFINITION 3.4. *(control flow graph) The* control flow graph *for a method m is a graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$. The nodes $\mathcal{N}$ consist of:*

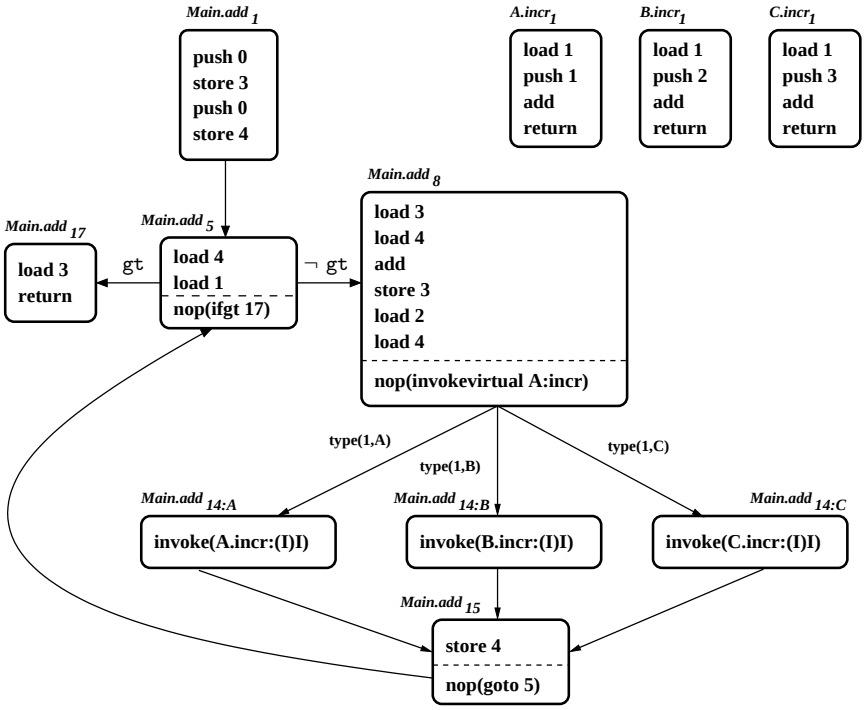


Fig. 4. The control flow graph of the `add` example

(1) *the basic blocks* $m_{i_1}, \ldots, m_{i_k}$ *of* m *(represented by their names); and*

(2) *the dispatch blocks* M_i^m *corresponding to all invocation instructions* b_i *in* m*.*

Edges in $\mathcal{E}$ *take the form* $\langle m_i \to m_j, \varphi_{ij} \rangle$*, where* m_i *and* m_j *are the source and destination nodes, and* φ_{ij} *is the Boolean condition labeling this transition. The set of edges is defined by considering each node* $m_i \in \mathcal{N}$ *which corresponds to a (non-dispatch) basic block, whose last instruction is* b_{f_i}*, as follows:*

(1) *if* b_{f_i}=goto j*, then* $\langle m_i \to m_j, true \rangle$ *is in* $\mathcal{E}$*, and* b_{f_i} *is annotated as* $\mathsf{nop}(b_{f_i})$ *in* m_i*;*

(2) *if* b_{f_i}=ifϕ j *where* $\phi \in \{\mathsf{lt}, \mathsf{gt}, \mathsf{eq}, \mathsf{null}\}$ *and* i'=$f_i + 1$ *then* $\langle m_i \to m_j, \phi \rangle$ *and* $\langle m_i \to m_{i'}, \neg\phi \rangle$ *are in* $\mathcal{E}$*, and* b_{f_i} *is annotated as* $\mathsf{nop}(b_{f_i})$ *in* m_i*;*

(3) *if* $b_{f_i} \in \{\mathsf{invokevirtual}\ m', \mathsf{invokenonvirtual}\ m'\}$ *and* $i' = f_i + 1$*, then, for all* $m_{f_i:c} \in M_{f_i}^m$*, the edges* $\langle m_i \to m_{f_i:c}, \mathsf{type}(n, c) \rangle$ *and* $\langle m_{f_i:c} \to m_{i'}, true \rangle$ *are in* $\mathcal{E}$*, and* b_{f_i} *is annotated as* $\mathsf{nop}(b_{f_i})$ *in* m_i*. The guard* $\mathsf{type}(n, c)$ *means that the runtime class of the object located on the* $(n+1)$*-th stack element from the top is* c *where* n *is the number of arguments of* m'*;*

(4) *otherwise, if* $b_{f_i} \neq$ return *and* j=$BlockId(f_i + 1, m)$*, then* $\langle m_i \to m_j, true \rangle$ *is in* $\mathcal{E}$*.* □

Example 3.5. *The CFGs for methods Main.add, A.incr, B.incr and C.incr are depicted in Figure 4. The CFG for Main.add consists of eight nodes, where*

b_j	comp(b_j)
load i	$s_{t+1} := l_i$
store i	$l_i := s_t$
push n	$s_{t+1} := n$
pop	nop(pop)
dup	$s_{t+1} := s_t$
add	$s_{t-1} := s_{t-1} + s_t$
sub	$s_{t-1} := s_{t-1} - s_t$
lt	$s_{t-1} < s_t$
gt	$s_{t-1} > s_t$
eq	$s_{t-1} = s_t$
null	$s_t =$ null
¬ lt	$s_{t-1} \geq s_t$
¬ gt	$s_{t-1} \leq s_t$

b_j	comp(b_j)
¬ eq	$s_{t-1} \neq s_t$
¬ null	$s_t \neq$ null
type(n, c)	type(s_{t-n}, c)
new c	$s_{t+1} :=$ new c
getfield f	$s_t := s_t.f$
putfield f	$s_{t-1}.f := s_t$
newarray c	$s_t :=$ newarray(c, s_t)
aload	$s_{t-1} := s_{t-1}[s_t]$
astore	$s_{t-2}[s_{t-1}] := s_t$
arraylength	$s_t :=$ arraylength(s_t)
invoke m	$m(s_{t-n}, \ldots, s_t, s_{t-n})$
return	$out := s_t$
nop(b)	nop(b)

Fig. 5. Compiling bytecode instructions (as they appear in the CFG) to rule-based instructions (t stands for the height of the stack before executing the bytecode instruction).

Main.add$_{14:A}$, Main.add$_{14:B}$ and Main.add$_{14:C}$ are dispatch blocks, and the remaining ones are basic blocks. Edges indicate that control may flow from the last instruction in the source node to the first instruction in the destination node. An edge can be labeled with a guard stating the conditions under which it can be traversed at runtime. For instance, the fact that the successor of ifgt 17 *can be either the instruction at address* 8 *or* 17 *is expressed by two guarded edges from Main.add$_5$, one to Main.add$_8$ and the other to Main.add$_{17}$. The invocation* 14: invokevirtual A.incr : (I)I *is split into three possible runtime instances, captured by the dispatch blocks Main.add$_{14:A}$, Main.add$_{14:B}$ and Main.add$_{14:C}$. Depending on the runtime type of the object* o*, whose memory location is stored on the second stack element (from top) as indicated by the guards in the edges, only one of these blocks will be executed. Each one transfers the control to the corresponding definition of* incr. □

Once the CFGs for all methods are generated, they can be translated into the rule-based language. A key point in this translation is that the top of the stack at each program point is known statically, so that stack elements can be treated as additional local variables. The bytecode instructions in the CFG are compiled to rule-based expressions comp(b_i) as described in Figure 5. In the following definition, given a block m_p, comp(m_p) is the (comma separated) compilation of its instructions.

Definition 3.6. *(rule-based representation) Let m be a method with local variables $\bar{l} = l_0, \ldots, l_n$, from which $\bar{l}' = l_0, l_1, \ldots, l_k$ with $k \leq n$ are the formal parameters, where l_0 is the* this *reference. Let $CFG_m = \langle \mathcal{N}, \mathcal{E} \rangle$ be its control flow graph. The* rule-based representation *of m consists of a set of procedures, each containing one or more rules, which are obtained as follows:*

$$\{m(\bar{l}', out) \leftarrow true, m_1(\bar{l}, out)\} \bigcup_{m_p \in \mathcal{N}} \text{translate}(m_p)$$

where $\mathsf{translate}(m_p)$ *is defined as:*

$$\begin{cases} \{(m_p(\bar{l}, s_1, \ldots, s_{beg}, out) \leftarrow true, \mathsf{comp}(m_p))\} & \nexists \langle m_p \rightarrow, \_ \rangle \in \mathcal{E} \\ \{(m_p(\bar{l}, s_1, \ldots, s_{beg}, out) \leftarrow true, \mathsf{comp}(m_p), m_q(\bar{l}, s_1, \ldots, s_{end}, out))\} & \exists! \langle m_p \rightarrow m_q, true \rangle \in \mathcal{E} \\ \{(m_p(\bar{l}, s_1, \ldots, s_{beg}, out) \leftarrow true, \mathsf{comp}(m_p), m_p^c(\bar{l}, s_1, \ldots, s_{end}, out))\} & \\ \quad \bigcup \{(m_p^c(\bar{l}, s_1, \ldots, s_{end}, out) \leftarrow \mathsf{comp}(\phi), m_q(\bar{l}, s_1, \ldots, s_{begq}, out)) & otherwise \\ \qquad \mid \langle m_p \rightarrow m_q, \phi \rangle \in \mathcal{E}\} & \end{cases}$$

where "beg" and "end" are, respectively, the height of the stack at the beginning and at the end of block m_p *and "begq" is the height of the stack at the beginning of the block* m_q*.* □

The RBR for each method m has an *entry procedure* (containing a unique rule) which simply calls the procedure m_1 of the first block in its CFG. Sink nodes in the CFG, represented by the first case of function translate, simply contain compiled bytecode instructions but do not have a *continuation* call. Rules with the superscript c, introduced in the last case of translate, correspond to continuation procedures (consisting of the possible *continuation rules*). They are used to choose one execution branch when the corresponding node in the CFG has more than one successor. If there is a single successor (guarded by *true*), then continuation procedures are avoided by the second case of translate. Also, in the following, m_p stands for a rule name if it is relevant to know that it comes from block p in the CFG. Otherwise, rules will be simply denoted by p and q.

Example 3.7. *Figure 6 shows the RBR obtained from the CFGs in Figure 4. For readability, local variables have the same name as in the source code, and guards which are true are omitted. The first rule is the entry procedure of Main.add, which receives the method arguments as input parameters. The call to* $Main.add_1$ *from this rule adds, as parameters, the local variables of the method. Non-continuation procedures are named as the corresponding blocks in the CFGs. The loop entry corresponds to procedure* $Main.add_5$*, where the bytecodes loading* i *and* n *on the stack are compiled to the assignments* $s_1 := i$ *and* $s_2 := n$*, respectively. This procedure calls* $Main.add_5^c$ *to check the condition of the loop. If* $s_1 > s_2$ *(i.e.,* $i > n$ *in the program source), then the control exits the loop and calls* $Main.add_{17}$*, which assigns* s_1 *to the return value out and terminates. Otherwise, the loop continues by calling* $Main.add_8$*, which first accumulates* i *on res and then prepares the call to A.incr by assigning its parameters to the stack variables. Finally, it calls* $Main.add_8^c$ *to continue the execution depending on the runtime type of* o*. The continuation procedure calls the corresponding dispatch block, named* $Main.add_{14:Class}$*, which invokes the* incr *instance which matches the guard. Calls to* incr *receive* s_1, s_2 *(which correspond to variables* o *and* i*, respectively) as input arguments, and return* s_1 *as the output argument to store the (incremented value) of* i*. The computation continues by calling* $Main.add_{15}$*, which stores the top of the stack (i.e., the incremented* i *returned by the call to* incr*) in* i*, and calls* $Main.add_5$ *(the loop entry) to proceed with the next iteration.* □

$Main.add(this, n, o, out) \leftarrow$	$Main.add_1(this, n, o, res, i, out).$
$Main.add_1(this, n, o, res, i, out) \leftarrow$	$s_1 := 0, res := s_1, s_1 := 0, i := s_1,$
	$Main.add_5(this, n, o, res, i, out).$
$Main.add_5(this, n, o, res, i, out) \leftarrow$	$s_1 := i, s_2 := n,$ nop(ifgt 17),
	$Main.add_5^c(this, n, o, res, i, s_1, s_2, out).$
$Main.add_5^c(this, n, o, res, i, s_1, s_2, out) \leftarrow$	$s_1 > s_2, Main.add_{17}(this, n, o, res, i, out).$
$Main.add_5^c(this, n, o, res, i, s_1, s_2, out) \leftarrow$	$s_1 \leq s_2, Main.add_8(this, n, o, res, i, out).$
$Main.add_{17}(this, n, o, res, i, out) \leftarrow$	$s_1 := res, out := s_1$
$Main.add_8(this, n, o, res, i, out) \leftarrow$	$s_1 := res, s_2 := i, s_1 := s_1 + s_2, res := s_1,$
	$s_1 := o, s_2 := i,$ nop(invokevirtual `A.incr(I)I`),
	$Main.add_8^c(this, n, o, res, i, s_1, s_2, out).$
$Main.add_8^c(this, n, o, res, i, s_1, s_2, out) \leftarrow$	type(s_1, A),
	$Main.add_{14:A}(this, n, o, res, i, s_1, s_2, out).$
$Main.add_8^c(this, n, o, res, i, s_1, s_2, out) \leftarrow$	type(s_1, B),
	$Main.add_{14:B}(this, n, o, res, i, s_1, s_2, out).$
$Main.add_8^c(this, n, o, res, i, s_1, s_2, out) \leftarrow$	type(s_1, C),
	$Main.add_{14:C}(this, n, o, res, i, s_1, s_2, out).$
$Main.add_{14:A}(this, n, o, res, i, s_1, s_2, out) \leftarrow$	$A.incr(\langle s_1, s_2\rangle, \langle s_1\rangle),$
	$Main.add_{15}(this, n, o, res, i, s_1, out).$
$Main.add_{14:B}(this, n, o, res, i, s_1, s_2, out) \leftarrow$	$B.incr(\langle s_1, s_2\rangle, \langle s_1\rangle),$
	$Main.add_{15}(this, n, o, res, i, s_1, out).$
$Main.add_{14:C}(this, n, o, res, i, s_1, s_2, out) \leftarrow$	$C.incr(\langle s_1, s_2\rangle, \langle s_1\rangle),$
	$Main.add_{15}(this, n, o, res, i, s_1, out).$
$Main.add_{15}(this, n, o, res, i, s_1, out) \leftarrow$	$i := s_1,$ nop(goto 5),
	$Main.add_5(this, n, o, res, i, out).$
$A.incr(this, i, out) \leftarrow$	$A.incr_1(this, i, out).$
$A.incr_1(this, i, out) \leftarrow$	$s_1 := i, s_2 := 1, s_1 := s_1 + s_2, out := s_1.$
$B.incr(this, i, out) \leftarrow$	$B.incr_1(this, i, out).$
$B.incr_1(this, i, out) \leftarrow$	$s_1 := i, s_2 := 2, s_1 := s_1 + s_2, out := s_1.$
$C.incr(this, i, out) \leftarrow$	$C.incr_1(this, i, out).$
$C.incr_1(this, i, out) \leftarrow$	$s_1 := i, s_2 := 3, s_1 := s_1 + s_2, out := s_1.$

Fig. 6. Rule-based Representation of `add`. Guards which are true are omitted.

3.3 Equivalence between a Bytecode Program and its corresponding RBR

This section states the equivalence between a bytecode program P and its corresponding rule-based representation P_{rr} obtained as described above. An operational semantics for RBR is presented, and it is shown that any *bytecode trace* in P has a corresponding *rule-based trace* in P_{rr}. The execution of P_{rr} requires a heap h_{rr} to allocate and access objects in the same way than P does. The stack ar_{rr} of *activation records* stores information about the point to which each active procedure should return the control when its execution terminates. When there is no confusion, we will omit the subscript rr. An important feature of P_{rr} which facilitates the design of the subsequent analysis is that, as already mentioned, there is no distinction anymore between calls to blocks and calls to methods, as they are both now procedure definitions in P_{rr}. As a consequence, ar_{rr} operates at the level of procedures (while ar works at the level of methods on the bytecode). This means that a new activation record is created whenever a new procedure is called.

Rules in Figure 7 define an *operational semantics* for the RBR. The notation for the local variables mapping lv and the heap h is inherited from the bytecode se-

$$(1)_{rr}\quad \frac{b \equiv x := exp,\quad v = eval(exp, lv)}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv[x \mapsto v]\rangle{\cdot}ar; h}$$

$$(2)_{rr}\quad \frac{b \equiv x := \mathsf{new}\ c,\quad o{=}newobject(c),\quad r{\notin}dom(h)}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv[x \mapsto r]\rangle{\cdot}ar; h[r \mapsto o]}$$

$$(3)_{rr}\quad \frac{b \equiv x := y.f,\quad lv(y) \neq \mathsf{null}}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv[x \mapsto h(lv(y)).f]\rangle{\cdot}ar; h}$$

$$(4)_{rr}\quad \frac{b \equiv x.f := y,\quad lv(x) \neq \mathsf{null},}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv\rangle{\cdot}ar; h[lv(x).f \mapsto lv(y)]}$$

$$(5)_{rr}\quad \frac{\begin{array}{c} b \equiv x[y] := z,\quad lv(x) \neq \mathsf{null},\quad o = h(lv(x)),\quad v_1 = lv(y), \\ 1 \leq v_1 \leq o.length,\quad v_2 = lv(z) \end{array}}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv\rangle{\cdot}ar; h[o.v_1 \mapsto v_2]}$$

$$(6)_{rr}\quad \frac{b \equiv x := z[y],\quad lv(z) \neq \mathsf{null},\quad o = h(lv(z)),\quad v = lv(y),\quad 1 \leq v \leq o.length}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv[x \mapsto o.v]\rangle{\cdot}ar; h}$$

$$(7)_{rr}\quad \frac{b \equiv x := \mathsf{arraylength}(y),\quad lv(y) \neq \mathsf{null},\quad o = h(lv(y))}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv[x \mapsto o.length]\rangle{\cdot}ar; h}$$

$$(8)_{rr}\quad \frac{b \equiv x := \mathsf{newarray}(c, y)\quad v = lv(y) \geq 0\quad o{=}newarray(c, v)\quad r{\notin}dom(h)}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv[x \mapsto r]\rangle{\cdot}ar; h[r \mapsto o]}$$

$$(9)_{rr}\quad \frac{b \equiv \mathsf{nop}(any)}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv\rangle{\cdot}ar; h}$$

$$(10)_{rr}\quad \frac{\begin{array}{c} b \equiv q(\bar{x}, y),\quad \text{there is a rule } q(\bar{x}', y'){:=}g, b_1, \cdots, b_k \in RBR, \\ lv'{=}newenv(q),\ \forall i.lv'(x_i') = lv(x_i),\ eval(g, lv') = true \end{array}}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle q, b_1 \cdots b_k, lv'\rangle{\cdot}\langle p[y', y], bc, lv\rangle{\cdot}ar; h}$$

$$(11)_{rr}\quad \frac{}{\langle q, \epsilon, lv\rangle{\cdot}\langle p[y', y], bc, lv'\rangle{\cdot}ar; h \leadsto_{\mathsf{rr}} \langle p, bc, lv'[y \mapsto lv(y')]\rangle{\cdot}ar; h}$$

Fig. 7. Operational semantics of bytecode programs in rule-based form

mantics. The table lv is indexed by variable names instead of integers, as in Figures 2 and 3. In addition, there is no operand stack, as stack positions have become local variables. An *activation record* is of the form $\langle p, bc, lv\rangle$, where p is a procedure name, bc is a sequence of instructions and lv the variable mapping. The first rule $(1)_{rr}$ accounts for all rules in the bytecode semantics which perform operations on variables (both local and stack variables). The evaluation $eval(exp, lv)$ returns the evaluation of the arithmetic or Boolean expression exp for the values of the corresponding variables from lv in the standard way and, for reference variables, it returns the reference. For well typed programs, the operational semantics always produces a successor configuration for any configuration which is not final. Essentially, we restrict ourselves to well typed programs since the bytecode programs we consider are well typed as they are verified (see Section 2.2). Since the program is well typed, the types of the variables are known statically. One can thus annotate the variables types at each program point. Instead, for simplicity, we assume that, given a variable x, $\mathsf{static\_type}(x)$ denotes its static type. Rules $(2)_{rr}$, $(3)_{rr}$ and $(4)_{rr}$ deal with objects as expected. Rules $(5)_{rr}$, $(6)_{rr}$, $(7)_{rr}$ and $(8)_{rr}$ account for arrays. The rule $(9)_{rr}$ takes care of ignoring nop-wrapped instructions. The rule $(10)_{rr}$ (resp., $(11)_{rr}$) corresponds to calling (resp., returning from) a procedure. The notation $p[y', y]$ records the association between the formal and actual return variables.

An execution in the RBR starts from an initial configuration $\langle start, p(\bar{x}, y), lv\rangle; h$ and ends in the configuration $\langle start, \epsilon, lv'\rangle; h'$ where:

(1) *start* is an auxiliary name to indicate an initial configuration, for clarity we use the same name as in the initial configuration of the bytecode;
(2) $p(\bar{x}, y)$ is a call to the procedure from which we want to start the execution;
(3) h is an initial heap; and
(4) lv is a variable mapping such that $dom(lv) = \bar{x} \cup \{y\}$ and all variables are initialized to an integer value, null or a reference to an object in h.

The following definition introduces the notion of *equivalence* between activation records and configurations of the bytecode, and those of the RBR.

Intuitively, the equivalence between activation records requires that they correspond to the same basic block in the CFG, have equivalent instructions and variables with exactly the same values. Equivalence of configurations requires configurations to have equivalent activations records at their top positions and that both heaps are identical.

Definition 3.8. *(activation record and configuration equivalence) A bytecode activation record a is (cost) equivalent to a rule-based activation record a_{rr}, denoted $a \approx a_{rr}$, if one of the following conditions holds:*

(1) $a = \langle start, 1, lv, v_t \cdots v_1\rangle$ *and* $a_{rr} = \langle start, m(\bar{s}, s_1), lv'\rangle$ *are initial activation records such that* $\bar{s} = \langle s_1, \ldots, s_t\rangle$, $lv'(s_i) = v_i$ *for all* $1 \leq i \leq t$, *and the instruction* $start[1]$ *corresponds to a (non-virtual) call to the method* m *in the corresponding bytecode program;*
(2) $a = \langle start, 2, lv, v\rangle$ *and* $a_{rr} = \langle start, \epsilon, lv'\rangle$ *are final activation records such that* $lv'(s_1) = v$*;*
(3) $a = \langle m, pc, lv, v_t \cdots v_1\rangle$ *and* $a_{rr} = \langle m, bc, lv'\rangle$ *such that:*
  (a) bc is the compilation of the bytecode instructions in the block to which $m[pc]$ *belongs, starting from* $m[pc]$ *until the end of the block (except for the last instruction in bc when it is a continuation call);*
  (b) for each local variable l_i *in* m, $lv(i) = lv'(l_i)$, *and, for each stack element* v_i *where* $1 \leq i \leq t$, $lv'(s_i) = v_i$.

A bytecode configuration $C{=}ar; h$ and a rule-based configuration $RC{=}ar_{rr}; h_{rr}$ are cost equivalent (written $C \approx RC$) if the top activation records are equivalent and h and h_{rr} are "identical". □

As we will see later, the cost of an execution step depends *only* on the values available in the top activation record, and therefore the above notion of equivalence is the simplest one that we need to define when two execution steps from a bytecode and a RBR configuration will cost the same. However, if we start an execution from equivalent initial configurations, we will show later that at any point during the execution we will have configurations which are (cost) equivalent. This in turn implies that the corresponding traces will cost the same.

Definition 3.9. *(redundant rule-based configurations) A rule-based configuration $\langle id, bc, lv\rangle \cdot ar; h$ is* redundant *if $id \neq start$, and bc is either an empty sequence or includes only procedure calls.* □

The following theorem states the soundness of the RBR transformation, from the point of view of the equivalence that we have defined above. It amounts to saying that, given a trace in the bytecode, we can always generate a RBR trace such that, when removing the redundant configurations, we obtain a sequence of RBR configurations which are (cost) equivalent to those of the bytecode trace.

THEOREM 3.10. *(trace equivalence) Consider a bytecode program P and its rule-based representation P_{rr}. Given a trace $t \equiv C_0 \leadsto^n_{\mathsf{bc}} C_n$ of length $n \geq 0$ in P, where $C_0 = \langle start, 1, lv_0, v_t \cdots v_1 \rangle; h_0$ is an initial configuration, then there exists a rule-based trace $t_{rr} \equiv RC_0 \leadsto^k_{\mathsf{rr}} RC_k$ of length $k \geq n$ in P_{rr}, such that:*

(1) $RC_0 = \langle start, m'(\bar{s}, s_1), lv'_0 \rangle; h_0$ where $s = \langle s_1, \ldots, s_t \rangle$ and $lv'_0(s_i) = v_i$ for all $1 \leq i \leq t$, where m' is the method invoked by start[1];

(2) removing redundant configurations from t_{rr} results in the sub-trace that consists of configurations $RC'_{i_0}, \ldots, RC'_{i_n}$ where $i_0 = 0 < i_1 < \cdots < i_n = k$ (i.e., of length n) such that $C_j \approx RC'_{i_j}$ for all $0 \leq j \leq n$.

We say that t_{rr} is equivalent to t, denoted $t \approx t_{rr}$. □

PROOF. see Appendix A. □

The other direction of the equivalence, namely that every rule-based trace has a corresponding bytecode trace, also holds. It is omitted here because it is not required for the soundness of cost analysis.

4. THE NOTION OF COST AND COST MODEL

For some purposes, such as when reasoning about program correctness, given a program P, it suffices to know *what* P computes, i.e., its *input–output behaviour*. However, other analyses may also require to know *how* P performs its computation. Reasoning about the *cost* of a program is a clear example of this. It focuses not only on the result of the computation, but also on some aspects of its *history*. Such history is captured to a great extent by its corresponding execution trace. It is thus natural that our notion of cost model is tightly related to execution traces. In practical terms, often only some specific features of traces are needed to reason about cost. Such features are clearly dependent on the resource of interest. For example, the classical notions of cost used in complexity analysis, such as the number of execution steps performed or the number of times certain instructions are executed, only consider a subset of the information provided by traces. In the case of bytecode programs, the natural choice for measuring execution steps amounts to counting the number of bytecode instructions executed (i.e., the *length* of the trace).

This section formally introduces the notion of *cost model* of bytecode programs, which characterizes how the resource consumption due to executing a program can be computed. To this end, a cost model defines how cost has to be assigned to an execution step and, by extension, to an entire trace. In general, a cost model can be viewed as an instrumentation of the program with *cost counting* instructions that accumulate the cost in a *ghost* variable. For example, counting the number of execution steps can be done by adding an instruction "$cost = cost + 1$" after each instruction in the program. Measuring the memory consumption can be done by

adding "$cost = cost + size(c)$" after each instruction "new c". This view poses an important feature of such cost models: they can be defined in terms of the values of local variables only (in addition to static information like $size(c)$ above) since they are written in the same language as the program. Therefore, as in the case of Java and Java bytecode programs, a cost model cannot access a variable in other activation records (unless it is aliased with a local one). All realistic cost models fall into this category. Considering the above restrictions, the cost model can be viewed as a function from bytecode instructions, and dynamic information (local variables, stack, and heap) to the real numbers. In the following, *Local* and *Stacks* denote, resp., the sets of all local variable tables and operand stacks which meet the above restrictions. The following definition formalizes the notion of such cost models.

DEFINITION 4.1. *(cost model) A* cost model $\mathcal{M}$ *is a function from*

$$\mathcal{M} : bcInstr \times Local \times Stacks \times Heaps \mapsto \mathbb{R}$$

□

Essentially, given an activation record and a corresponding heap, the cost model can be used to compute a number which represents the amount of resources which will be consumed by the corresponding execution step. Note that, for simplicity, we ignore the context in which the bytecode appears (the method), but taking this information into account is straightforward. Let us see some examples of cost models, which are indeed implemented in our system (see Section 7).

EXAMPLE 4.2. *The first model, denoted $\mathcal{M}_{inst}$, counts the number of instructions by giving cost 1 to every configuration, and it is defined as $\mathcal{M}_{inst}(b, lv, stk, h) = 1$ for any b, lv, stk and h. The next model $\mathcal{M}_{heap}$ is used for estimating the heap consumption [Albert et al. 2007]. It assigns the number of bytes which are allocated by the next instruction. For instance, the bytecode instruction "*newarray *int" (res. "*newarray *c") allocates $v * size(int)$ (res. $v * size(ref)$) bytes in the heap, where v denotes the length of the array stored on the top of the stack and $size(int)$ (res. $size(ref)$) denotes the number of bytes allocated for an integer value (res. reference) by the corresponding virtual machine, e.g., $size(int) = size(ref) = 4$:*

$$\mathcal{M}_{heap}(b, lv, stk, h) = \begin{cases} size(c) & m[pc] = \text{new } c \\ v * size(int) & b = \text{"newarray } int\text{"} \text{ and } stk{=}v{\cdot}stk' \\ v * size(ref) & b = \text{"newarray } c\text{"} \text{ and } stk{=}v{\cdot}stk' \\ 0 & otherwise \end{cases}$$

The $\mathcal{M}_{calls(m')}$ model is used to count the number of calls to a certain method m'. It gives cost 1 to a configuration where the next instruction is any non-virtual call to m' or a virtual call which is resolved to a call to m'.

$$\mathcal{M}_{calls(m')}(b, lv, stk, h) = \begin{cases} 1 & b = \mathsf{invokevirtual}\ m'' \\ & m''\ has\ k\ arguments \\ & stk = v_n \cdots v_{n-k} \cdot stk' \\ & m' = lookup(m'', class(h(v_{n-k}))) \\ 1 & b = \mathsf{invokenonvirtual}\ m' \\ 0 & otherwise \end{cases}$$

Note that the heap is required in this model to infer the cost. □

For simplicity, given a configuration $C \equiv a \cdot ar; h$, where $a = \langle m, pc, lv, stk \rangle$ we write $\mathcal{M}(C)$ for $\mathcal{M}(m[pc], lv, stk, h)$. Then, the *cost of an execution step* $C_1 \leadsto_{\mathsf{bc}} C_2$ with respect to $\mathcal{M}$ is $\mathcal{M}(C_1)$, and the cost of a trace t, denoted $\mathcal{M}(t)$, is the sum of the costs of its execution steps. Note that, in order to simplify the representation, we use $\mathcal{M}$ to refer to the cost of a configuration and a trace.

In order to reason about the cost on the RBR, it is also necessary to define cost models on RBR executions. This is done similarly to Definition 4.1, because of the tight relation between the bytecode and the RBR language. We use $Local_{rr}$ to denote the set of all local variable mappings in the RBR.

Definition 4.3. *(RBR cost model) A* cost model $\mathcal{M}^{rr}$ *on the RBR is a function*

$$\mathcal{M}^{rr} : rrInstr \times Local_{rr} \times Heaps \mapsto \mathbb{R}.$$

□

Example 4.4. *As an example, the RBR version of* $\mathcal{M}_{heap}$ *is defined as follows:*

$$\mathcal{M}^{rr}_{heap}(b, lv, h) = \begin{cases} size(c) & b \equiv x := \mathsf{new}\ c \\ lv(y) * size(int) & b \equiv x := \mathsf{newarray}(int, y) \\ lv(y) * size(c) & b \equiv x := \mathsf{newarray}(c, y) \\ 0 & otherwise \end{cases}$$

It can be seen that, in addition to the type of the array elements c, the size y of the array is an explicit argument of the $\mathsf{newarray}(c, y)$ *instruction of the RBR.* □

The cost of the execution with respect to a cost model $\mathcal{M}^{rr}$ can be defined similarly to the bytecode version: given an RBR configuration $RC \equiv a \cdot ar; h$ where $a = \langle p, b_1 \cdots b_n, lv \rangle$, we write $\mathcal{M}^{rr}(RC)$ for $\mathcal{M}^{rr}(b_1 \cdots b_n, lv, h)$. For simplicity, if $n = 0$ we assume cost 0. Now the cost of an execution step $RC_1 \leadsto_{\mathsf{rr}} RC_2$ is $\mathcal{M}^{rr}(RC_1)$, and the cost $\mathcal{M}^{rr}(t_{rr})$ of a trace t_{rr} is the sum of the costs of all execution steps.

In general, given a cost model on the bytecode it is straightforward to define a cost model on the RBR which is equivalent. For this, the cost model should (1) assign cost zero to configurations which correspond to procedure calls and return from such calls, since they do not correspond to bytecode instructions and (2) assign to instructions marked as nop the cost corresponding to the instruction they contain, since they do not affect execution but should be taken into account by the cost model. In addition, it should be observed that different bytecode instructions are

compiled to similar variable assignments in the RBR (see, e.g., the compilations of load and store in Figure 5). If the cost model requires one to assign them different resource consumptions, we can simply use the nop construct available in the RBR syntax to keep the original bytecode instructions in the RBR and then assign a corresponding cost to each of them. In order to simplify the presentation, notation will be abused by denoting both models in the bytecode and the RBR by the same name $\mathcal{M}$.

THEOREM 4.5. *(cost equivalence) Consider a bytecode program P and its rule-based representation P_{rr}, and let $\mathcal{M}$ be a cost model (the same name is used for both bytecode and RBR versions). Given a trace t in P and an equivalent trace t_{rr} in P_{rr}, then $\mathcal{M}(t) = \mathcal{M}(t_{rr})$.* □

PROOF. This theorem is a direct consequence of Theorem 3.10. □

5. COST ANALYSIS OF RULE-BASED PROGRAMS

Given a program P and a cost model $\mathcal{M}$, the classical approach to cost analysis, as proposed by Wegbreit [1975], consists in obtaining a set of *Recurrence Relations* (RRs for short) which capture the cost of running P on some input data $\overline{x}$ w.r.t. $\mathcal{M}$. As usual, in order to analyze programs which use data structures, in the recurrence relations, data structures are replaced by their *sizes*. This section describes how this approach can be applied to rule-based programs in order to obtain *Cost Relation Systems* (CRSs), an extended form of RRs, which describe their costs. In our approach, each rule in the RBR program results in an equation in the CRS. For instance, using the $\mathcal{M}_{inst}$ cost model, the rule defining $Main.add_8$ in Figure 6 results in the equation:

$$\begin{aligned} Main.add_8(this, n, o, res, i) \;=\; & \langle 1, s'_1{=}res\rangle + \\ & \langle 1, s'_2{=}i\rangle + \\ & \langle 1, s''_1{=}s'_1{+}s'_2\rangle + \\ & \langle 1, res'{=}s''_1\rangle + \\ & \langle 1, s'''_1{=}o\rangle + \\ & \langle 1, s''_2{=}i\rangle + \\ & \langle 1, true\rangle + \\ & \langle Main.add^c_8(this, n, o, res', i, s'''_1, s''_2), true\rangle \end{aligned}$$

In the above equation, the variables are constraint variables that correspond to those of the corresponding rule, for example s'_1 and s''_1 both correspond to values of s_1 but at different program points. Each pair $\langle e, \varphi\rangle$ in the right hand side of the equation corresponds to an instruction in the corresponding rule: e expresses the cost of executing that instruction and φ is the effect of the instruction on the variables (in terms of linear constraints). The pair in the last line corresponds to the cost of executing $Main.add^c_8$ on specific input values $this, n, o, res', i, s'''_1$ and s''_2. The constraint *true* in that pair corresponds to the effect of calling $Main.add^c_8$ on the local variables. Note that the output variable of the rule does not appear in the equation, as it will be explained later.

Under certain conditions, we can merge the constraints and add the corresponding costs together. For instance, for the above equation, we would obtain:

$$\begin{aligned} Main.add_8(this, n, o, res, i) \;=\; & \langle 7, res' = res + i, s'''_1{=}o, s''_2{=}i\rangle + \\ & \langle Main.add^c_8(this, n, o, res', i, s'''_1, s''_2), true\rangle \end{aligned}$$

which states that, given values (sizes) for $this, n, o, res$ and i, the cost of executing $Main.add_8(this, n, o, res, i)$ is 7 units plus the cost of executing the call $Main.add_8^c(this, n, o, res', i, s_1''', s_2'')$. As we will explain later, it is not always sound to anticipate constraints.

A cost equation like the one above is generated by the analyzer for each rule in the RBR, by following the main steps which are explained throughout this section in detail:

(1) The first step in the analysis is to choose *size measures* to represent and manipulate information relevant to cost. Program variables are abstracted to their *size*. For example, a list can be abstracted to its length, since this can give information about the cost of traversing it by means of a loop. The notion of size measure is described in detail in Section 5.1.

(2) Instructions in the original rule are replaced by *linear constraints* which approximate the relation between states with respect to the size measures. For instance, $s_1 := o$ is replaced by the constraint $s_1'''{=}o$, which means that, after the assignment, the size of s_1 at that program point (represented by s_1''') is equal to the size of o. The result of this step is an *abstract program* which can be used to approximate the values of the different variables in concrete traces with respect to the given size measures. This step corresponds to an *abstract compilation* of the RBR, as explained in Section 5.2.

(3) The next step removes output variables from the original rules by inferring the relation between the input variables and the output variable using *input-output size relations* (Section 5.3). Due to this, the output *out* does not appear in the above equation.

(4) Finally, the analysis generates a CRS by using the abstract rules to generate the constraints, and the original rule together with the selected cost model to generate *cost expressions* representing the cost of the bytecodes with respect to the model (Section 5.4). In the example, the cost expressions are the constant values, which correspond to the number of executed instructions.

CRSs resemble the classical recurrence relations, but they are an extended form in the sense that allow, in addition, to handle advanced features such as *non-determinism* and *constraints*. These features are needed for a precise cost analysis of real programs as it will become clear during this section; the differences between RRs and CRSs will be explained in Section 6.2.

This section needs the introduction of some notation. A *linear expression* takes the form $q_0{+}q_1x_1{+}\cdots{+}q_nx_n$, where q_i are rational numbers and x_i are variables. A *linear constraint* (over integers) takes the form l_1 *op* l_2, where l_1 and l_2 are linear expressions, and $op \in \{=, \leq, <, >, \geq\}$. A *size relation* φ is a set of linear constraints, interpreted as a conjunction. The statement $\varphi_1 \models \varphi_2$ indicates that φ_1 implies φ_2. A *substitution* σ maps (constraints) variables to values in $\mathbb{Z}$, and $\sigma \models \varphi$ denotes that σ is a consistent assignment (i.e., a possible solution) for φ. Given φ_1 and φ_2, $\varphi_1 \sqcup \varphi_2$ denotes their convex-hull [Cousot and Halbwachs 1978]. We use $a \ll_c A$ to indicate that an entity a is a renamed apart (from c) element of A, i.e., does not share any variable with the entity c.

5.1 The Notion of Size Measure

As already mentioned, data structures are usually abstracted into their sizes for the purpose of cost analysis. Beginning with [Wegbreit 1975], a large number of different *size measures* or *norms* have been proposed in the context of cost and termination analysis over the years (see, e.g., [Bruynooghe et al. 2007] and its references for an overview of the range of measures used in termination analysis of logic programs).

The choice of a size measure, in particular for heap structures, heavily depends on the program to be analyzed. For example, in termination analysis, the norm should describe some measure which decreases at each loop iteration. If the program input is a list which is traversed by the program, a typical example of a useful measure is the length of the list. This measure can often be used for bounding the number of loop iterations. Similarly, if the input is an array, the array length can often be used for bounding the number of loop iterations. Finally, if the input is an integer value, then the actual numerical value is usually a good measure to estimate the number of iterations of loops with an integer counter. In addition to the previous examples, other norms for data structures have been defined, such as *term-size*, which measures the number of data constructs in a given data structure. It has also been suggested that size measures counting data constructs of specific (recursive) types are very useful in practice [Bruynooghe et al. 2007].

In the rest of this paper, the following size measures are used, which in practice have been proven to be precise enough for a large class of programs.

DEFINITION 5.1. *(size measure) Given a configuration* $RC \equiv \langle p, bc, lv\rangle \cdot ar; h$*, the size of* $x \in dom(lv)$ *with respect to a static type* stype *is defined as:*

$$\alpha(x, \mathsf{stype}, RC) = \begin{cases} lv(x) & \textit{if } \mathsf{stype} \textit{ is integer type} \\ \textit{path-length}(lv(x), h) & \textit{if } \mathsf{stype} \textit{ is reference type} \\ \textit{array-length}(lv(x), h) & \textit{if } \mathsf{stype} \textit{ is array type} \end{cases}$$

□

The function path-length above corresponds to Definition 5.1 in the previous work by Spoto et al. [2006b]. It takes a heap h and a reference $lv(x) \in dom(h)$, and returns the length of the maximal *path* reachable from that reference by *dereferencing*, i.e., following other references stored as fields. The path-length of null is 0 and that of a *cyclic* data structure is ∞. path-length is useful for both linear and non-linear data structures. In the case of lists, path-length coincides with *list-length*. In the case of trees, it coincides with *tree-depth*, which has been used in [Debray and Lin 1993] for the case of logic programs. However, tracking size information in logic programs is simplified by the fact that size information is *downward closed* because variables (i.e., memory locations) cannot be re-assigned different (non-variable) values in forward execution, as it happens in imperative programming languages.

The choice of a size measure affects abstract compilation, as defined in the next section, while the following steps of the cost analysis are independent of the size measure. The only requirement is that its evaluation must be context-independent, i.e., an object is always measured in the same way independently from its context [Bossi et al. 1991].

5.2 Abstract Compilation

This section describes how to transform a given rule-based program P into an abstract program P^α, which can be seen as an abstraction of P with respect to the chosen size measure α. The translation is based on replacing each instruction by (linear) *constraints* which describe its behavior with respect to the size measure. For example, the instruction $x :=$ new c can be replaced by the constraint $x{=}1$, which indicates that the maximal path length reachable from x is 1. The fact that the formula refers to the path length of variables is due to the previous choice of the size measure for references. To simplify the presentation, and when it is clear from the context, the same name is used for the original variables (possibly primed) and their sizes. That is, given a list l, the name l is also used, in the abstract compilation, to denote its maximal path length. Letting α denote the size measure of Definition 5.1, the translation of the instructions in the RBR is depicted in Figure 8.

An important issue in the presented setting is to be able to obtain relations between the *size* of a variable at different program points. For example, in analyzing $x := x+1$, the interest can be in the relation "*the value of x after the instruction is equal to the value of x before the instruction plus 1*". This important piece of information can be obtained by using a *Static Single Assignment* (SSA) transformation, which, together with the abstract compilation, produces the constraint $x'{=}x+1$, where x and x' refer to, respectively, the value of x before and after the instruction. To implement the SSA transformation, a mapping ρ of variable names (as they appear in the rule) to new variable names (constraint variables) is maintained. Such mapping is referred to as a *renaming*. The expression $\rho[x \mapsto y]$ denotes the update of ρ, such that it maps x to the new variable y.

Let us explain in detail the compilation of some instructions in Figure 8. As already mentioned in Section 5.1, the use of path-length as a size measure for reference requires extra information in order to obtain precise and sound results in the abstract compilation of the two instructions involving references: (a) *sharing* information [Secci and Spoto 2005] is required in order to know whether two variables might point (either directly, by *aliasing*, or indirectly) to a common region of the heap; and (b) *non-cyclicity* information [Rossignoli and Spoto 2006] is required in order to guarantee that a reference points to a non-cyclic data structure (i.e., that the length of its longest path is guaranteed to be finite) at some specific program point. In case (3) in Figure 8, we distinguish two cases depending on the possible cyclicity of y. If it can be determined that x (before executing the instruction) does not reference a cyclic data-structure, then we can ensure that $\rho(y){>}x'$, since the length of the longest path reachable from y is larger than the length of the longest path reachable from x' (i.e., $y.f$). Otherwise, we abstract it to $\rho(y){\geq}x'$. In both cases, it is ensured that $x'{\geq}0$. Our analysis does not handle numeric fields, hence, the instruction $x := y.f$ is abstracted to *true* if f is a numeric field. If the field is an array, then all we can say about its abstract value is that it is greater than or equal to zero. This is because arrays are abstracted to their length. We could mantain two abstractions for such arrays: one for its length and one for its path-length, but for simplicity we only keep their length abstraction. In case (4), if a reference field is modified by $y.f := x$, and x and y do not share, then the

b	b^α w.r.t a renaming ρ	ρ'
(1) $x := exp$	$x'{=}exp^\alpha$	$\rho[x \mapsto x']$
(2) $x :=$ new c	$x'{=}1$	$\rho[x \mapsto x']$
(3) $x := y.f$	$true$ – if f is a numeric field $\rho(y) > x' \wedge x' \geq 0$ – if f is a (non-array) reference field and y is non-cyclic $\rho(y) \geq x' \wedge x' \geq 0$ – if f is a (non-array) reference field and y might be cyclic $x' \geq 0$ – otherwise	$\rho[x \mapsto x']$
(4) $x.f := y$	$true$ – if f is a numeric field – $S = \emptyset$ $\wedge\{v' \leq \rho(v) + \rho(y) \wedge v' \geq 0 \mid v \in S\}$ – if f is a (non-array) reference field and $y \notin \mathsf{SH}_x$ – $S = \{v \mid v \in \mathsf{SH}_x,\ v$ is not of an array type$\}$ $\wedge\{v' \geq 0 \mid v \in S\}$ – otherwise – $S = \{v \mid v \in \mathsf{SH}_x,\ v$ is not of an array type$\}$	$\rho[\forall v \in S.$ $v \mapsto v']$
(5) $x :=$ newarray(c, y)	$x' = \rho(y) \wedge x' \geq 0$	$\rho[x \mapsto x']$
(6) $x[i] := y$	$true$	ρ
(7) $x := y[i]$	$true$ – if y is an array of integers $x' \geq 0$ – otherwise	$\rho[x \mapsto x']$
(8) $x :=$ arraylength(y)	$x' = \rho(y) \wedge x' \geq 0$	$\rho[x \mapsto x']$
(9) $p(\bar{x}, y)$	$\langle p(\rho(\bar{x}), y'), \varphi_1 \wedge \varphi_2 \rangle$ – U_p=$\{v\mid\ v \in \bar{x}$, v might be updated in p $\}$ – S=$\{v\mid\ x \in U_p$, $v \in \mathsf{SH}_x$ and is not an array$\}$ – φ_1= $\wedge\ \{v' \geq 1 \mid v \in S$ is definitely not null before call$\}$ – φ_2= $\wedge\ \{v' \geq 0 \mid v \in S$ might be null before call$\}$	$\rho[\forall v{\in}S{\cup}\{y\}.$ $v \mapsto v']$
(10) type(x, c)	$x \geq 1$	ρ
(11) $exp_1 \neq exp_2$	$true$	ρ
(12) exp_1 op exp_2	exp_1^α op exp_2^α	ρ
(13) nop(any)	$true$	ρ
(14) exp	$exp[\mathsf{null} \mapsto 0, x \mapsto \rho(x)]$	ρ
(15) $true$	$true$	ρ

Fig. 8. Abstract Compilation of Instructions

length of the maximal path reachable from y and any variable sharing with y might change. The set SH_x denotes the set of variables that might share with x before the corresponding instruction. This change can be safely described by $v' \leq \rho(v) + \rho(y)$, where v is any variable which might share with x (including x). If x and y share, then no safe information can be provided, except that the size of the corresponding reference variables is greater than or equal to 0. Further details on the path-length analysis can be found in related work [Spoto et al. 2006b] and they are outside the scope of this article.

In (9), the abstraction of calls to procedures (or methods) requires the computation of the set of input variables such that the data structures they point to may be modified during the execution of the invoked procedure, denoted by U_p in Figure 8.

This set can be approximated by applying *constancy analysis* [Genaim and Spoto 2008], which aims at detecting such arguments. The set of updated input variables (closed by sharing), denoted by S in the figure, is renamed in order to "*forget*" them after the call (i.e., not to propagate) the constraints that involved the updated variables before the call to the state. For instance, consider the following call $p(x, z, out)$ and assume that z is updated during the execution of p. Let ψ be a constraint over x and z which holds before the call. Then, a fresh variable z' must be used after the call instead of z, i.e., $p(x, z', out)$, in order to distinguish between the variable z before and after the call to p. Let us note that the update property refers to updating heap structures. If only numeric fields are modified, it is not considered an update and, thus, the length is preserved. When an argument might be updated during a call, we can still say that the size (after the call) of every variable x that might share with that argument (including itself) is: (a) greater than or equal to one if it is guaranteed that x is not null; (b) otherwise, greater than or equal to 0. In the system, we have a simple nullity analysis, as explained in Section 7, that we can use to verify this condition. Note that for a freshly created object, case (2), it is always able to say that it is not-null before calling the constructor. The abstraction of calls can be improved by using shallow variables for the procedure arguments (specially for constructors). This is a well-known technique that can improve the precision of OO analysis and therefore of path-length. In general, such shallow variables come on very high price of performance.

Several instructions are abstracted to *true*. In this case, no information is provided. For instance, in array operations, no information is available on the elements since an array is abstracted to its length. Also, numeric fields are abstracted to *true* since a more sophisticated analysis is required to handle them (see [Miné 2006]). Moreover, *non-linear* arithmetic such as $x * y$ and x/y would be also abstracted to *true*, as linear constraints cannot approximate the behavior without extra information. In our system, we apply *constant propagation* analysis in order to identify cases where y is constant and therefore improve the precision.

Sharing and non-cyclicity information is precise only if it is computed with respect to a specific *context*. Therefore, the soundness of the transformation is guaranteed under an initial context description which contains sharing and non-cyclicity information. In practice, if the initial call is a Java-like main method, then such an initial description is not required, as all data structures are created at runtime, instead of being provided as an input. In what follows, an initial description of a context takes the form $Q \equiv \langle p(\bar{x}), \mathsf{SH}, \mathsf{NC} \rangle$ (output variables are ignored), where $\mathsf{SH} \subseteq \bar{x} \times \bar{x}$ and $\mathsf{NC} \subseteq \bar{x}$. The statement $(x, y) \in \mathsf{SH}$ means that x and y might have a common data structure on the heap, and $x \in \mathsf{NC}$ means that x points to a non-cyclic data-structure. An initial configuration $\langle start, p(\bar{x}, y), lv \rangle; h$ is said to be safely approximated by Q if: (1) if two reference variables $x, y \in dom(lv)$ share a common region on h, then $(x, y) \in \mathsf{SH}$; and (2) if a reference variable $x \in dom(lv)$ points to a cyclic data structure, then $x \notin \mathsf{NC}$. The information contained in SH and NC is propagated by means of fixpoint computations, as described by, respectively, Secci and Spoto [2005] and Rossignoli and Spoto [2006]. Essentially, such analyses provide the information which is required in order to answer the (program point) queries about sharing and cyclicity in Figure 8.

DEFINITION 5.2. *(abstract compilation) Consider a rule* $r \equiv p(\overline{x}, y) \leftarrow g, b_1, \ldots, b_n$, *and let* ρ_1 *be the* identity renaming *over* $var(r)$. *The abstract compilation of* r *with respect to a size measure* α *is* $r^\alpha \equiv p(\overline{x}, y') \leftarrow \varphi_0 \mid b_1^\alpha, \ldots, b_n^\alpha$ *where:*

(1) g^α *is the abstract compilation of* g *w.r.t. the renaming* ρ_1.
(2) $\varphi_0 = \{\rho_1(z) = 0 \mid z \in var(r) \setminus \bar{x}\} \wedge g^\alpha$;
(3) b_i^α *is the abstract compilation of* b_i *using* ρ_i;
(4) ρ_{i+1}, $1 \leq i \leq n$, *is generated from* ρ_i *and* b_i *as shown in Figure 8;*
(5) $y' = \rho_{n+1}(y)$
(6) $\rho = \langle \rho_1, \ldots, \rho_{n+1} \rangle$ *is the renaming associated to the abstract compilation.*

Given a rule-based program P, *an initial context description* Q *and a size measure* α, *we denote by* P^α *the program obtained by abstracting all its rules using the sharing and cyclicity information obtained when starting from an initial description* Q. □

Recall that in Section 3, for simplicity, we assumed that input and output variables of each procedure have the same names in all rules that correspond to that procedure. This property is preserved in the abstract compilation for the input variables (because we start from the identity renaming) but not for the output variable. In order to simplify the notation, we assume that P^α has been modified such that this property also holds for the output variable. This is simply done by renaming the output variable to a new variable that is not used before P^α.

EXAMPLE 5.3. *The rule shown on the left (taken from Figure 6) is abstracted to the rule which appears on the right.*

$Main.add_8(this, n, o, res, i, out) \leftarrow$	$Main.add_8(this, n, o, res, i, \rho_9(out)) \leftarrow$
	$\{s_1{=}0, s_2{=}0, out{=}0\} \mid$
$s_1 := res,$	$s_1'{=}res,$
$s_2 := i,$	$s_2'{=}i,$
$s_1 := s_1 + s_2,$	$s_1''{=}s_1'{+}s_2',$
$res := s_1,$	$res'{=}s_1''$
$s_1 := o,$	$s_1'''{=}o,$
$s_2 := i,$	$s_2''{=}i,$
nop(invokevirtual `A.incr(I)I`),	$true,$
$Main.add_8^c(this, n, o, res, i, s_1, s_2, out).$	$\langle Main.add_8^c(this, n, o, res', i, s_1''', s_2'', out'), true \rangle.$

The renaming $\rho = \langle \rho_1, \ldots, \rho_9 \rangle$ *used to generate the abstract rule is:*

—ρ_1 *is the identity renaming over the variables* $\{this, n, o, res, i, s_0, s_1, s_2\}$,
—$\rho_2 = \rho_1[s_1 \mapsto s_1']$,
—$\rho_3 = \rho_2[s_2 \mapsto s_2']$,
—$\rho_4 = \rho_3[s_1 \mapsto s_1'']$,
—$\rho_5 = \rho_4[res \mapsto res']$,
—$\rho_6 = \rho_5[s_1 \mapsto s_1''']$,
—$\rho_7 = \rho_6[s_2 \mapsto s_2'']$,
—$\rho_8 = \rho_7$,
—$\rho_9 = \rho_8[out \mapsto out']$.

It can be observed that variables which do not appear in the rule head are initiliazed in the body (first condition in Definition 5.2). As an example, when the abstraction of $s_1 := s_1+s_2$ is going to be done, according to the abstract compilation in Figure 8, we have in the renaming ρ_3 the mappings $s_1 \mapsto s_1'$ and $s_2 \mapsto s_2'$ introduced by the compilation of $s_1 := res$ and $s_2 := i$ respectively. First, such renaming is applied on the expression s_1+s_2 which leads to $s_1'+s_2'$. Next, the abstract compilation of the expression (first row in Figure 8) produces $s_1'' := s_1' + s_2'$ and adds the mapping $s_1 \mapsto s_1''$ to the renaming ρ_3 generating the new renaming ρ_4. In the above rule, the variable o stands for a reference: as it is not updated during the execution of $Main.add_8^c$, there is no renaming. Note also that we have not added the abstraction of the guard since it is true. □

An abstract RBR basically abstracts the behavior of the original program with respect to the size measure α. Its operational semantics is given by the following transition system which simply accumulates the constraints (when possible) and proceeds to execute the calls in the body of the rules:

$$\frac{p(\bar{x},y) \leftarrow \varphi \mid b_1^\alpha,\ldots,b_n^\alpha \ll_{AC} P^\alpha,\ \ \psi\wedge\varphi \not\models \mathit{false}}{AC{=}\langle\langle p(\bar{x},y),\phi\rangle\cdot bc^\alpha,\psi\rangle \rightsquigarrow_\alpha \langle b_1^\alpha\cdots b_n^\alpha\cdot\phi\cdot bc^\alpha,\psi\wedge\varphi\rangle} \qquad \frac{\psi\wedge\varphi \not\models \mathit{false}}{\langle\varphi\cdot bc^\alpha,\psi\rangle \rightsquigarrow_\alpha \langle bc^\alpha,\psi\wedge\varphi\rangle}$$

In the leftmost transition rule, a renamed apart abstract rule (w.r.t. AC) is used for p from P^α, i.e, it does not share any variable with AC. For the sake of simplicity, we avoid another renaming step, by assuming that the renamed apart rule is retrieved with the same input $\bar{x}$ and output y variables that appear in the call.

The rest of this section proves the *soundness* of the abstract compilation with respect to the chosen size measure α. Intuitively, we prove that the size of the variables in a given concrete trace can be observed in a corresponding abstract trace. For this, we prove that, given a concrete trace, we can generate an abstract trace of the same length and instantiate it (i.e, give integer values to all constraints variables using a consistent assignment σ) in such a way that the size of a variable in the i-th concrete state coincides with the value of the corresponding constraint variable in the i-th abstract state.

LEMMA 5.4. *(soundness of abstract compilation) Let P_{rr} be a rule based representation for a program P, $Q \equiv \langle p(\bar{x}), \mathsf{SH}, \mathsf{NC}\rangle$ an initial context description and P^α the corresponding abstract program. Let $RC_0{\equiv}\langle start, b, lv_0\rangle; h$ such that Q is a safe description of RC_0, $b = p(\bar{x},y)$ and $dom(lv_0) = \bar{x}\cup\{y\}$. If $RC_0 \rightsquigarrow_{\mathsf{rr}}^n RC_n$, then there exists an abstract trace $AC_0{\equiv}\langle b^\alpha,\varphi_0\rangle \rightsquigarrow_\alpha^n AC_n{\equiv}\langle \_,\varphi_n\rangle$, a partial map $f : var(P)\times\{0,\ldots,n\} \mapsto var(AC_n)$, and a consistent assignment $\sigma : var(AC_n) \mapsto \mathbb{Z}$ for φ_n such that: for any $RC_i = \langle \_,\_,lv_i\rangle\cdot ar_i; h_i$ and $AC_i{\equiv}\langle \_,\varphi_i\rangle$ $(0 \le i \le n)$ it holds that $\varphi_n \models \varphi_i$ and $\forall z \in dom(lv_i).\alpha(z,\mathsf{static\_type}(z),RC_i) = \sigma(f(z,i))$.*

The above lemma states that each (abstract) state AC_i is a safe approximation (w.r.t. the size measure α) of the corresponding activation record in RC_i. It is easy to observe that we can take φ_0 as the abstraction of RC_0, namely:

$$\varphi_0 = \bigwedge_{z\in\bar{x}\cup\{y\}} id(z) = \alpha(z,\mathsf{static\_type}(z),RC_0)$$

The partial map, f, is used to relate program variables to their corresponding constraints variables. This mapping can be constructed (as shown in the proof) by collecting the renamings (enriched with a state index) of the abstract rules used during the evaluation. Note that by b^α we mean the abstraction of the call b (as explained in Figure 8) with respect to, for example, the identity renaming over $\bar{x} \cup \{y\}$. We use "_" in order to indicate parts of an entity that we are not interested in, instead of assigning them names that will not be used. The proof of this lemma is by induction on the length of the derivation. For the instructions which handle objects it relies on the soundness of the path-length analysis (Theorem 5.12 by Spoto et al. [2006b]).

5.3 Input-Output Size Relations

As an important point, CRSs are *mathematical relations*, in the same way as RRs are mathematical functions. Hence, they cannot have output variables: instead, they should receive a set of input parameters and *return a number*. This step of the analysis is meant to transform the abstract program obtained in the previous section into one with an *equivalent* behavior where output variables do not appear. The basic idea relies on computing abstract *input-output (size) relations* in terms of linear constraints, and using them to propagate the effect of calling a rule. Essentially, we consider the abstract rules obtained in the previous step to infer an abstraction (w.r.t. size) of the input-output ("io" in abbreviations) relation of the program blocks. Concretely, we infer input-output size relations of the form $p(\bar{x}, y) \rightarrow \varphi$, where φ is a constraint describing the relation between the sizes of the input $\bar{x}$ and the output y upon exit from p. This information is needed since the output of one call may be input to another call. For instance, consider the following abstract rule:

$$p(x, y') \leftarrow \{w{=}0, z{=}0, y{=}0\} \mid x{>}0, z'{=}x-1, \langle q(z', w'), true\rangle, \langle p(w', y'), true\rangle$$

Assuming that the call $q(z', w')$ will generate the constraint $z' \geq w'$, this rule can be transformed into:

$$p(x) \leftarrow \{w{=}0, z{=}0, y{=}0\} \mid x{>}0, z'{=}x-1, \langle q(z'), z' \geq w'\rangle, \langle p(w'), true\rangle$$

which does not have any output arguments. Note that this transformation makes it possible to infer $x{>}w'$, which is crucial for bounding the number of loop iterations.

The following definition introduces the notion of input-output relations, which can be seen also as a denotational semantics for the abstract programs of Section 5.2. For each procedure, it describes the (size) relations between its input and output values. The definition is based on a semantics operator $\mathcal{T}_{P^\alpha}$, which describes how from a set of input-output relations I, we learn more relations by applying the rules in the abstract program.

Definition 5.5. *(input-output relations) Consider the following operator* $\mathcal{T}_{P^\alpha}$

defined as:

$$\mathcal{T}_{P^\alpha}(I) = \left\{ p(\bar{x},y) \to \psi \;\middle|\; \begin{array}{ll} 1) & r \equiv p(\bar{x},y) \leftarrow \varphi \mid b_1^\alpha,\ldots,b_n^\alpha \circ \rho \in P^\alpha \\ 2) & \forall 1 \le i \le n, \quad \textit{either} \\ & \quad 2.1)\ b_i^\alpha \textit{ is a constraint } \varphi_i\textit{; or} \\ & \quad 2.2)\ b_i^\alpha \equiv \langle q_i(\bar{w}_i,z_i),\phi\rangle \textit{ where } q_i(\bar{w}_i,z_i) \to \varphi_i' \ll_r I \\ & \qquad \textit{and we let } \varphi_i = \phi \wedge \varphi_i' \\ 3) & \psi = (\varphi \wedge \varphi_1 \wedge \cdots \wedge \varphi_n)|_{\bar{x}\cup\{y\}} \end{array} \right\}$$

where the projection $\varphi|_S$ removes from φ all variables but those in the set S. The input-output relations *of an abstract program P^α, denoted by $\mathcal{I}(P^\alpha)$, is defined as* $\bigcup_{i\in\omega} \mathcal{T}^i_{P^\alpha}(\emptyset)$ *where* $\mathcal{T}^i_{P^\alpha}(I) = \mathcal{T}_{P^\alpha}(\mathcal{T}^{i-1}_{P^\alpha}(I))$. □

As before, we use "$q_i(\bar{w}_i,z_i) \to \varphi_i' \ll_r I$" to select a renamed apart (w.r.t. r) element from I and, besides, we retrieve it with the proper variables $\bar{w}_i$ and z_i in the head.

Example 5.6. *The following input-output relations are obtained from the corresponding procedures in the RBR of the running example:*

$$\begin{array}{l} A.incr(this,i,out) \to \ \{out{=}i{+}1\} \\ B.incr(this,i,out) \to \ \{out{=}i{+}2\} \\ C.incr(this,i,out) \to \ \{out{=}i{+}3\} \end{array}$$

It can be seen that the output variable out is only related to the input variable i in all cases. This piece of information will be crucial for inferring the cost. □

The following lemma is a well-known result in the context of logic programming (see, e.g., [Benoy and King 1997]) which establishes the correctness of input-output size relations obtained by means of a fix-point computation.

Lemma 5.7. *Let P^α be an abstract program. If $t \equiv \langle\langle p(\bar{x},y),\phi\rangle,\psi_0\rangle \rightsquigarrow^*_\alpha \langle\epsilon,\psi\rangle$ is an abstract trace, then there exists $p(\bar{x},y) \to \varphi \ll_t \mathcal{I}(P^\alpha)$ such that $\psi \models \varphi$.* □

Computing $\mathcal{I}(P^\alpha)$ is often impractical, as it might include an infinite number of objects. However, it can be approximated using abstract interpretation techniques [Cousot and Cousot 1977]. In particular, it can be done by using a convex-hull operator $\sqcup$ instead of $\cup$, and incorporating a *widening* operator to guarantee the termination of the process [Cousot and Halbwachs 1978]. The only requirement of our framework is to have a *safe* approximation of the input-output relations as the next definition states.

Definition 5.8. *(safe approximation of input-output relations) A set A is a safe approximation of the input-output relations of a program P^α, iff for any $a \equiv p(\bar{x},y) \to \varphi \in \mathcal{I}(P^\alpha)$ there exists $p(\bar{x},y) \to \psi \ll_a A$ such that $\varphi \models \psi$.* □

In addition, for simplicity, for each procedure p we assume that the set A contains only one input-output relation $p(\bar{x},y) \to \psi$ for p. This can be done by simply merging all the input-output relations of p using the convex hull operator $\sqcup$.

The following definition describes how to remove the output variables from an abstract program P^α by using a safe approximation of the input-output relations. The idea is that, for each call $p(\bar{w},z)$ in a rule r, the input-output relation $p(\bar{w},z) \to \varphi \ll_r A$ is used in order to eliminate z, but still propagate its relation with $\bar{w}$ which is generated by the execution of p.

DEFINITION 5.9. *Given an abstract program P^α and a safe approximation A of its input-output behavior, P^{io} denotes the abstract program which is generated from the rules of P^α, as follows. Each rule $r \equiv p(\bar{x}, y) \leftarrow \varphi \mid b_1^\alpha, \ldots, b_n^\alpha \in P^\alpha$ is replaced by $p(\bar{x}) \leftarrow \varphi \mid b_1^{io}, \ldots, b_n^{io}$, where*

—if $b_i^\alpha = \langle q(\bar{w}, z), \varphi_i \rangle$ then $b_i^{io} = \langle q(\bar{w}), \varphi_i \wedge \psi \rangle$, where $q(\bar{w}, z) \rightarrow \psi \ll_r A$; and
—if b_i^α is a constraint then $b_i^{io} = b_i^\alpha$.

EXAMPLE 5.10. *Using the input-output relations shown in Example 5.6, the output variables of the rules $Main.add_{14:A}$, $Main.add_{14:B}$ and $Main.add_{14:C}$ (defined in Figure 6) are eliminated as follows:*

$$Main.add_{14:A}(n, o, res, i, s_1, s_2) \leftarrow$$
$$\quad true \mid \langle A.incr(s_1, s_2), \{s_1' = s_2 + 1\} \rangle, \langle Main.add_{15}(this, n, o, res, i, s_1'), true \rangle.$$

$$Main.add_{14:B}(n, o, res, i, s_1, s_2) \leftarrow$$
$$\quad true \mid \langle B.incr(s_1, s_2), \{s_1' = s_2 + 2\} \rangle, \langle Main.add_{15}(this, n, o, res, i, s_1'), true \rangle.$$

$$Main.add_{14:C}(n, o, res, i, s_1, s_2) \leftarrow$$
$$\quad true \mid \langle C.incr(s_1, s_2), \{s_1' = s_2 + 3\} \rangle, \langle Main.add_{15}(this, n, o, res, i, s_1'), true \rangle.$$

It can be observed that the above rules contain the essential pieces of information about the increasing of s_2 after executing each implementation of `incr`. □

The generated abstract rules can be executed by using the following transition system. They are identical to the execution of the abstract rules explained in Section 5.2, except that they do not have output variables:

$$\frac{p(\bar{x}) \leftarrow \varphi \mid b_1^{io}, \ldots, b_n^{io} \ll_{AC} P^{io},\ \psi \wedge \varphi \not\models false}{AC = \langle \langle p(\bar{x}), \phi \rangle \cdot bc^{io}, \psi \rangle \rightsquigarrow_{io} \langle b_1^{io} \cdots b_n^{io} \cdot \phi \cdot bc^{io}, \psi \wedge \varphi \rangle} \qquad \frac{\psi \wedge \varphi \not\models false}{\langle \varphi \cdot bc^{io}, \psi \rangle \rightsquigarrow_{io} \langle bc^{io}, \psi \wedge \varphi \rangle}$$

In the leftmost rule, we postpone the application of the input-output relation of constraints ϕ until the body of p has been executed. This was noted at the beginning of Section 5 where we commented that merging the constraints together is not always possible. Indeed, anticipating this relation might be unsound. For example, if the call to p does not terminate, the cost of the corresponding b_1 until b_n would not be considered since ϕ would be equal to $false$.

The soundness of this stage is established by the next lemma. Intuitively, it states that the result (in terms of constraints) of executing the abstract rules without output variables (but with input-output relations) is a safe approximation of the execution of the abstract rules with output variables (as defined in Section 5.2).

LEMMA 5.11. *Let P^α be an abstract program and P^{io} its corresponding program generated as in Definition 5.9 with respect to a safe approximation A of its input-output size relations. Then, If $AC_0 \rightsquigarrow_\alpha^n AC_n$ is a trace in P^α where $AC_0 \equiv \langle \langle p(\bar{x}, y), \phi \rangle, \varphi_0 \rangle$, then $AC'_0 \rightsquigarrow_{io}^n AC'_n$ is an abstract trace in P^{io} such that: (1) $AC'_0 \equiv \langle \langle p(\bar{x}), \phi \wedge \psi \rangle, \varphi_0 \rangle$ where $q(\bar{x}, y) \rightarrow \psi \ll_{AC_0} A$; and (2) for any $AC_i \equiv \langle \_, \varphi_i \rangle$ and $AC'_i \equiv \langle \_, \varphi'_i \rangle$ $(0 \leq i \leq n)$ it holds that $\varphi_i \models \varphi'_i$.* □

Note that in the above lemma, the two traces must use the same variables in order to get such equivalence, otherwise $\varphi_i \models \varphi'_i$ will not hold. This is possible since

during the execution we can select the corresponding clauses (from P^{α} and P^{io}) with same variable names (as shown in the proof).

In practice, the phase of computing the input-output relations can be skipped and directly approximated by *top*, namely, "$p(\bar{x}, y) \rightarrow true$" can be assumed for every rule. This is enough to infer the cost of many programs, and results in a more efficient implementation as it does not require any fix-point computation. As will be described in Section 7, the presented implementation of this cost analysis framework allows the user to decide, by means of an option, whether to compute input-output relations or to take the *top* approximation.

5.4 Building Cost Relation Systems

This section presents the following step in the proposed approach to cost analysis: the automatic generation of *cost relation systems* (CRSs) which capture the computational cost of executing a bytecode method with respect to a cost model of interest. In this step, CRSs are generated by incorporating *symbolic cost expressions* into the abstract rules generated in the previous steps. The idea is to use such cost expressions in order to express the cost of executing rules in the original program, with respect to the cost model. The next definition syntactically characterizes the kind of symbolic cost expressions the presented approach deals with.

Definition 5.12. *(symbolic cost expression) A* symbolic cost expression exp *is an expression of the form:*

$$\mathsf{exp} ::= n \mid x \mid \mathsf{exp}\ op\ \mathsf{exp} \mid \mathsf{exp}^{\mathsf{exp}} \mid \mathsf{log_a(exp)} \qquad op \in \{+, -, /, *\}$$

where a *is a natural number greater than* 1*,* n *is a positive real number and* x *an integer variable.* □

In cost analysis, symbolic cost expressions are used for two purposes: (1) to count the resources we accumulate in the different cost models, thus, to define the cost relation systems; (2) to describe the closed-form solutions (or upper bounds) of the cost relation systems. In the above definition, it can already be observed that we aim at covering a wide range of *complexity classes*: in addition to polynomial cost expressions, exponential and logarithmic expressions (and any combination among them) are also handled. Typically, when we compose the resources consumed by the different relations, the resulting cost expression must also account for other complexity classes.

Definition 4.3 needs to be adapted to the symbolic setting, so that, given an instruction, a cost model returns a symbolic expression (instead of a constant value) which denotes the resources the instruction consumes. In the following, *Instr* denotes the set of all instructions, and *Exprs* denotes the set of all cost expressions which can be generated using the grammar in Definition 5.12.

Definition 5.13. *(symbolic cost model) Let* α *be the size measure of Definition 5.1, and let* $\mathcal{M}$ *be a cost model as introduced by Definition 4.3. The partial map* $\mathcal{M}^s$*:Instr* $\mapsto$ *Exprs is said to be a* symbolic cost model *for* $\mathcal{M}$ *iff for any configuration* $RC{=}\langle m, b \cdot bc, lv\rangle \cdot ar; h$*, if* $e = \mathcal{M}^s(b)$ *then* $e[\forall x \in var(e) \mapsto \alpha(x, \mathsf{static\_type}(x), RC)] = \mathcal{M}(b, lv, h)$*.* □

Intuitively, given a configuration such that b is the next instruction to be executed, then the *evaluation* of the symbolic expression that $\mathcal{M}^s$ returns for b, must be equal to applying $\mathcal{M}$ on the configuration. Note that cost models which do not depend only on program variables (and static information) might not have corresponding symbolic cost models. As explained in Section 4, we are not interested in such cost models since they are based on information that is not observable by the user. In principle, we could eliminate such models directly in Definition 4.1, but in order to keep the definitions simpler, we restrict them at this stage.

EXAMPLE 5.14. *The symbolic version of the $\mathcal{M}_{heap}$ cost model, defined in Example 4.4, is defined as follows:*

$$\mathcal{M}^s_{heap}(b) = \begin{cases} size(c) & b \equiv x := \mathsf{new}\ c \\ y * size(int) & b \equiv x := \mathsf{newarray}(int, y) \\ y * size(c) & b \equiv x := \mathsf{newarray}(c, y) \\ 0 & otherwise \end{cases}$$

The main difference is that Example 4.4 uses $lv(y)$, which refers to the concrete evaluation of y w.r.t. the table of local variables in the current configuration. On the other hand, the symbolic cost model returns the symbolic cost expression y, which can be evaluated a posteriori for any particular configuration. □

At this point, the above definition provide all the elements to introduce cost relation systems.

DEFINITION 5.15. *(cost relation systems) Let P_{rr} be a rule based representation for a program P. Consider a rule $r \equiv p(\bar{x}, y) \leftarrow g, b_1, \ldots, b_n \in P_{rr}$, its abstract compilation with an associated renaming ρ (after eliminating the output variables) $r^{io} \equiv p(\bar{x}) \leftarrow \varphi \mid b_1^{io}, \ldots, b_n^{io} \in P^{io}$, and a symbolic cost model $\mathcal{M}^s$. The cost equation of the rule is $r^{eq} \equiv p(\bar{x}) = \varphi \mid b_1^{eq} + \cdots + b_n^{eq}$, where:*

—*if $b_i^{io} = \langle q(\bar{w}), \phi \rangle$, then $b_i^{eq} = \langle q(\bar{w}), \phi \rangle$*

—*if $b_i^{io} = \varphi_i$, where φ_i is the abstract compilation of b_i, then $b_i^{eq} = \langle \rho_i(\mathcal{M}^s(b_i)), \varphi_i \rangle$ where ρ_i is the i-th renaming in the tuple of renamings ρ.*

We denote by P_{cr} the cost relation system composed by the cost equations obtained from the rules in P. □

Essentially, the output of cost analysis is the above CRS, i.e., a set of *recursive* equations which have been generated by abstracting the iteration constructs of the program (loops and recursion), and by inferring size relations between its arguments. Two important issues about the above definition are that: (1) size relations between the rule variables are associated to the cost equations (at different points) to describe how the sizes of the data change when the equations are applied; and (2) guards do not affect the cost: they are simply used to define the applicability conditions of the equations.

CRSs are powerful tools. They are not limited to any complexity class. For instance, they can capture the cost of exponential methods which contain several recursive calls or that of logarithmic methods where the size of the data is halved with every loop iteration, etc. In principle, there is no restriction on the classes

$$
\begin{array}{lcl}
Main.add(this,n,o) & = & \langle Main.add_1(this,n,o,res,i), true\rangle \\
Main.add_1(\bar{x}_3) & = & \langle 4, res'{=}0, i'{=}0\}\rangle{+}\langle Main.add_5(this,n,o,res',i'), true\rangle \\
Main.add_5(\bar{x}_3) & = & \langle 3, \{s_1'{=}i, s_2'{=}n\}\rangle{+}\langle Main.add_5^c(\bar{x}_3, s_1', s_2'), true\rangle \\
Main.add_5^c(\bar{x}_1) & = & \{\mathbf{s_1} > \mathbf{s_2}\} \mid \langle Main.add_{17}(\bar{x}_3), true\rangle \\
Main.add_5^c(\bar{x}_1) & = & \{\mathbf{s_1} \leq \mathbf{s_2}\} \mid \langle Main.add_8(\bar{x}_3), true\rangle \\
Main.add_{17}(\bar{x}_3) & = & \langle 2, \{y'{=}res\}\rangle \\
Main.add_8(\bar{x}_3) & = & \langle 7, \{res'{=}res{+}i, s_1'''{=}o, s_2''{=}i\}\rangle{+} \\
 & & \quad \langle Main.add_8^c(this,n,o,res',i,s_1''',s_2''), true\rangle \\
Main.add_8^c(\bar{x}_1) & = & \langle Main.add_{14:A}(\bar{x}_1), true\rangle \\
Main.add_8^c(\bar{x}_1) & = & \langle Main.add_{14:B}(\bar{x}_1), true\rangle \\
Main.add_8^c(\bar{x}_1) & = & \langle Main.add_{14:C}(\bar{x}_1), true\rangle \\
Main.add_{14:A}(\bar{x}_1) & = & \langle A.incr(s_1,s_2), \{s_1'{=}s_2{+}1\}\rangle{+} \\
 & & \quad \langle Main.add_{15}(this,n,o,res,i,s_1'), true\rangle \\
Main.add_{14:B}(\bar{x}_1) & = & \langle B.incr(s_1,s_2), \{s_1'{=}s_2{+}2\}\rangle{+} \\
 & & \quad \langle Main.add_{15}(this,n,o,res,i,s_1'), true\rangle \\
Main.add_{14:C}(\bar{x}_1) & = & \langle C.incr(s_1,s_2), \{s_1'{=}s_2{+}3\}\rangle{+} \\
 & & \quad \langle Main.add_{15}(this,n,o,res,i,s_1'), true\rangle \\
Main.add_{15}(\bar{x}_2) & = & \langle 2, \{i'{=}s_1\}\rangle{+}\langle Main.add_5(this,n,o,res,i'), true\rangle \\
A.incr(this,i) & = & \langle A.incr_1(this,i), true\rangle \\
A.incr_1(this,i) & = & \langle 4, \{y' = i{+}1\rangle\} \\
B.incr(this,i) & = & \langle B.incr_1(this,i), true\rangle \\
B.incr_1(this,i) & = & \langle 4, \{y' = i{+}2\}\rangle \\
C.incr(this,i) & = & \langle C.incr_1(this,i), true\rangle \\
C.incr_1(this,i) & = & \langle 4, \{y' = i{+}3\}\rangle
\end{array}
$$

$$\bar{x}_1 = \langle this,n,o,res,i,s_1,s_2\rangle,\ \bar{x}_2 = \langle this,n,o,res,i,s_1\rangle,\ \bar{x}_3 = \langle this,n,o,res,i\rangle$$

Fig. 9. The CRS of the example

of programs whose cost can be represented by means of CRSs. Another feature of CRSs which make them powerful is that they can be used for counting a variety of non-trivial notions of resources, such as those mentioned in Section 4.

Example 5.16. *Consider the recursive representation in Example 3.7, and the size relations derived by size analysis (some of them appear in Example 5.6). By applying Definition 5.15, the CRS in Figure 9 is obtained. Note that the CRS has been simplified to make it more readable: (1) Some input arguments are written as $\bar{x}_1$, $\bar{x}_2$ and $\bar{x}_3$ where each $\bar{x}_i$ is defined at the bottom of the figure; (2) Constraints that stem from the implicit variable initialization (to* 0 *or* null*) are ignored as in this example all variables are explicitly initialized before they are used; (3) "true" guards are omitted; (4) Consecutive pairs $\langle e, \varphi\rangle$ are grouped together when possible. For example, in the equation $Main.add_8$, we have grouped all pairs that have a constant cost together as explained before at the beginning of Section 5; (5) Constraints are simplified, e.g., equalities $x = y$ have been eliminated by unifying the corresponding variables.* □

The evaluation of CRSs is defined by means of the following rules:

$$\frac{p(\bar{x}) \leftarrow \varphi \mid b_1^{eq} + \ldots + b_n^{eq} \ll_{AC} P_{cr},\ \psi \wedge \varphi \not\models false}{AC{=}\langle\langle p(\bar{x}), \phi\rangle \cdot bc^{eq}, exp, \psi\rangle \rightsquigarrow_{cr} \langle b_1^{eq} \cdots b_n^{eq} \cdot \langle 0, \phi\rangle \cdot bc^{eq}, exp, \psi \wedge \varphi\rangle}$$

$$\frac{\psi \wedge \varphi \not\models false}{\langle\langle e, \varphi\rangle \cdot bc^{eq}, exp, \psi\rangle \rightsquigarrow_{cr} \langle bc^{eq}, e{+}exp, \psi \wedge \varphi\rangle}$$

which essentially perform three actions: (1) check the satisfiability of the constraints (and accumulate them); (2) if it is not a call, add its symbolic cost expression to the accumulated cost; and (3) proceed to evaluate the next calls in the rule. Observe that, as in Section 5.3, we delay the application of the effects of executing a call (i.e., ϕ) by adding the pair $\langle 0, \phi\rangle$ to be considered after evaluating the call.

The following theorem establishes the *soundness* of the proposed cost analysis. It amounts to saying that, given a derivation in a rule-based program with associated cost a, there exists a derivation in its CRS with the same cost a.

THEOREM 5.17. *(soundness) Let P_{rr} be a RBR program, $Q{\equiv}\langle p(\bar{x}), \mathsf{SH}, \mathsf{NC}\rangle$ an initial context description, $\mathcal{M}$ a cost model and $\mathcal{M}^s$ its corresponding symbolic cost model. Let P_{cr} be the cost relation system w.r.t. $\mathcal{M}^s$. Then, if $RC_0{\equiv}\langle start, b, lv_0\rangle \rightsquigarrow_{\mathsf{rr}}^n RC_n$ is a trace t_{rr} for P_{rr} such that Q is a safe description for RC_0, $b = p(\bar{x}, y)$ and $dom(lv_0) = \bar{x} \cup \{y\}$, then there exists a trace $\langle b^{eq}, 0, \varphi_0\rangle \rightsquigarrow_{cr}^n \langle \_, e, \varphi_n\rangle$ and a consistent assignment $\sigma : var(\varphi_n) \mapsto \mathbb{Z}$ for φ_n such that $e\sigma{=}\mathcal{M}(t_{rr})$.* □

Note that by b^{eq} we mean the abstraction of the call b (as explained in Figure 8) with respect to, for example, the identity renaming over $\bar{x}{\cup}\{y\}$. Also, the constraint φ_0 is basically an over approximation of the initial state RC_0 w.r.t. α. The proof can be found in Appendix B.

The above theorem together with the equivalence of the bytecode and its RBR (Theorem 3.10) establish the correctness of our cost analysis of bytecode programs.

6. SIMPLIFYING, SOLVING AND APPROXIMATING COST RELATION SYSTEMS

CRSs can be considered a *lingua franca* in the sense that they can be used as the target for cost analysis of any language. They abstract away the particular language features and are simply an instrumentation of the abstracted version of the program—which is independent of the programming language in which the input program to the cost analysis is written—which allows approximating its cost. In every case, the cost relations inferred by the analysis generally depend on the cost of other calls (i.e., they are often recursive). Undoubtly, it is useful in practice to obtain a non-recursive representation of the equations, known as *closed-form*. Such a closed-form representation can be a solution for the input size arguments, or an upper/lower threshold cost, for each relation which approximates its cost. It is important to note that, in spite of the fact that in the previous steps we use static analyses which perform over approximations, it is still possible to obtain a sound lower bound for our CRSs. The intuition is that, given an initial state and its corresponding concrete trace t, the set of abstract traces in the CRSs corresponds to many possible concrete traces, but one of them is guaranteed to be t. If we find a lower bound on the cost among these traces, it is guaranteed to be a lower bound of all corresponding concrete traces, and therefore of t. For example, consider the program:

```
while (x>0) {
 if (*) then x=x-1 else x=x-2
}
```

The following equation is generated for the above program:

$$p(x) = \{\mathbf{x} > \mathbf{0}\} \mid \langle 1,\ x - 2 \leq x' \leq x - 1\rangle + \langle p(x'), true\rangle$$

from which we can see that $x/2$ is a lower bound on the number of iterations, i.e., we consider traces that decrease x by 2 and not 1. In general, lower bounds might be not as precise as upper bounds in cases where path-length is used, but it is ensured to be sound.

This section discusses several practical issues related to simplifying, solving and approximating the CRSs generated by automatic cost analysis, which are not specific to bytecode but common to any programming language. A first step to make the process of solving (or approximating) CRSs simpler and more efficient is to simplify them by restricting them to the subset of input arguments which are relevant to the cost. In Section 6.1, we describe a technique to carry out this simplification. Regarding the solving and approximation process itself, in the field of algorithmic complexity [Wilf 2002], obtaining closed-form solutions for *Recurrence Relation Systems* (RRs) is a well studied problem. As CRSs resemble RRs in many aspects, it has been typically assumed that the output of cost analysis are simply RRs. We highlight the differences between CRSs and RRs in Section 6.2 and overview the existing technology to solve both CRSs and RRs in Section 6.3. Explaining the process of transforming recurrence equations into closed-form is outside the scope of this article (see, e.g., [Albert et al. 2008a; Bagnara et al.]).

6.1 Restricting Cost Relations to (Subsets of) Input Arguments

In program analysis, it is customary to remove information from the program representation, provided the simplified representation preserves the behavior with respect to the analysis of interest. This reduction, performed at some intermediate step of the analysis, can lead to improvements (in terms of efficiency or accuracy) in later steps, and may even be required to make some of them possible at all. In many cases, the *computational overhead* due to the reduction is small, compared to the overall improvements. In the context of cost analysis, one may want to remove from the program representation information which does not affect the cost, in order to obtain a smaller and simpler representation which, possibly, helps the next steps in terms of efficiency and feasibility. In the present framework, this boils down to removing variables of the RBR which are *irrelevant* to cost. In general, *relevant* variables are those which

(1) affect the *abstract control flow* of the program: rule arguments which can have an impact on the cost of the program are those which may affect directly or indirectly the abstract program guards, i.e., those guards whose abstraction is different from *true*;

(2) are relevant to the cost of *single steps* w.r.t. the cost model, i.e., variables such that the cost of some instruction depends on them. For instance, the heap consumption due to a newarray(c, y) instruction is a function of the RBR input variables c and y, meaning, resp., the type of the data and the array length.

More concretely, consider a block p in a CFG, represented by the rule $p(\bar{l}_k, out) \leftarrow bc$, where $\bar{l}_k$ is $l_1, \ldots, l_k$. The cost of this rule is a function of the input variables $\bar{l}_k$, so that the focus is on minimizing the number $n \leq k$ of arguments which need to be taken into account in describing the cost. This can be obtained by only keeping variables which are relevant to (1) or (2), and propagating their effects through the RBR according to *data* and *control dependencies*. The algorithm to detect such relevant variables relies on the well-established technique of *backward program slicing* [Tip 1995]: variables directly relevant w.r.t. (1) or (2) are taken as the *slicing criterion* (i.e., the information which has to be preserved during the transformation). A variable at some program point is kept in the slice if it can affect (according to data and/or control dependencies) the criterion. Slicing propagates dependencies backwards through the RBR, and gets rid of variables which cannot affect the criterion. The analysis is *global*, and a standard *fixpoint* is needed in order to deal with recursion.

As an example, a variable which is only used as an *accumulator* (i.e., to keep a temporary result) does not have any effect on the control flow and, therefore, on the cost. Also, in many cases, the *this* reference takes no part in the computation. Slicing removes such kind of variables. A more detailed description can be found in [Albert et al. 2008c].

EXAMPLE 6.1. *Consider again the program in Figure 6, and assume that the chosen cost model implies that the cost of any instruction is* constant*, i.e., does not depend on any input variables. Slicing finds out, for example, that n and i are the relevant input variables of the rule* $Main.add_5$*, since (1) "this" is never used in the computation (2) res is only an accumulator and (3) o is an object involved in* type*-guards which, as shown in Figure 8, are abstracted to true. In particular, note that n and i are relevant since they are compared at the beginning of procedure* $Main.add_5^c$*. On the other hand, for the rule* $Main.add_{14:A}$*, the stack variable* s_1 *is also relevant at this point. In fact,* s_1 *is the result of executing incr, and it is stored in i in rule* $Main.add_{15}$*. Since i is relevant for* $Main.add_5^c$*, then* s_1 *is also (indirectly) relevant to the guard in* $Main.add_5^c$*. Additionally, our system safely replaces stack variables by the local variables or constants whose value was loaded on the corresponding stack location in the program whenever possible. This step is explained later in Section 7.1.* □

This approach does not properly fall in standard program slicing, since its purpose is not to obtain an executable subset of the program statements. Instead, reducing the RBR leads to a smaller form where some variables have been removed. As an important feature of this RBR slicing, *any* set of relevant variables can be safely used in order to refine the CRS, even if it is not a superset of the actual (semantically) relevant variables. In other words, *soundness* is not required, and removing useful variables may result, in the worst case, in a less precise outcome for cost. Once relevant variables are computed for every rule, this information is used to build a simpler CRS where irrelevant variables have been removed. Thus, after applying slicing (and stack variable elimination) to the RBR in Figure 6, the CRS in Figure 10 is obtained, which is considerably more readable than the one in Example 5.16.

$$
\begin{array}{lcl}
Main.add(n) & = & \langle Main.add_1(n,i), true\rangle \\
Main.add_1(n,i) & = & \langle 4, \{i' = 0\}\rangle + \langle Main.add_5(n,i'), true\rangle \\
Main.add_5(n,i) & = & \langle 3, true\rangle + \langle Main.add_5^c(n,i), true\rangle \\
Main.add_5^c(n,i) & = & \{i > n\} \mid \langle Main.add_{17}(n,i), true\rangle \\
Main.add_5^c(n,i) & = & \{i \leq n\} \mid \langle Main.add_8(n,i), true\rangle \\
Main.add_{17}(n,i) & = & \langle 2, true\rangle \\
Main.add_8(n,i) & = & \langle 7, true\rangle + \langle Main.add_8^c(n,i), true\rangle \\
Main.add_8^c(n,i) & = & \langle Main.add_{14:A}(n,i), true\rangle \\
Main.add_8^c(n,i) & = & \langle Main.add_{14:B}(n,i), true\rangle \\
Main.add_8^c(n,i) & = & \langle Main.add_{14:C}(n,i), true\rangle \\
Main.add_{14:A}(n,i) & = & \langle A.incr(), \{s_1' = i + 1\}\rangle + \langle Main.add_{15}(n,i,s_1'), true\rangle \\
Main.add_{14:B}(n,i) & = & \langle A.incr(), \{s_1' = i + 2\}\rangle + \langle Main.add_{15}(n,i,s_1'), true\rangle \\
Main.add_{14:B}(n,i) & = & \langle A.incr(), \{s_1' = i + 3\}\rangle + \langle Main.add_{15}(n,i,s_1'), true\rangle \\
Main.add_{15}(n,i,s_1) & = & \langle 2, true\rangle + \langle Main.add_5(n,s_1), true\rangle \\
A.incr() & = & \langle A.incr_1(), true\rangle \\
A.incr_1() & = & \langle 4, true\rangle \\
B.incr() & = & \langle B.incr_1(), true\rangle \\
B.incr_1() & = & \langle 4, true\rangle \\
C.incr() & = & \langle C.incr_1(), true\rangle \\
C.incr_1() & = & \langle 4, true\rangle
\end{array}
$$

Fig. 10. The simplified CRS

6.2 CRSs versus Recurrence Equation Systems

After having simplified the CRS, the natural questions are: *(i) is the result of cost analysis a system of recurrence equations? (ii) can thus we use existing recurrence equations solvers? (iii) can we obtain a closed-form solution or upper bound of it which is not recursive?* This section tries to answer question (*i*). Indeed, CRSs have several important features which are a consequence of being obtained by automatic program analysis, and which are not present in traditional RRs and pose new challenges to the solving and approximation processes:

(1) Non-deterministic relations. In contrast to RRs, cost equations for the same relation do not need to be mutually exclusive. The reason for allowing this is because cost analysis needs to use *size abstractions*. Unavoidably, the use of abstraction introduces a loss of precision: some guards which make the execution of the original program deterministic may not be observable when using the size of arguments instead of their actual values. For instance, in our running example, the guards type in procedure $Main.add_8^c$ make the procedure to be deterministic in the RBR. However, after doing the abstract compilation they are not observable anymore due to the use of the path-length abstraction for objects. Hence, the guards are lost and the resulting equations for $Main.add_8^c$ make up a non-deterministic relation to be approximated by a solver.

(2) Size relations. CRS have size relations attached to the cost expressions and, besides, they can contain inequality constraints. This is essential in order to handle cost relations automatically obtained from the analysis of realistic programs with complex data structures, for which size analysis may lose precision. For instance,

analysis may be able to infer that a given data structure strictly decreases in size from one iteration to another, but it may be unable to provide the precise reduction. This typically happens in loops like *while* $(x! = null)\{\ldots,\ x = x.next;\}$, where a list (or any other data structure) is traversed and the analyzer can only infer that variable x decreases from one iteration to another but does not know exactly how much. For the previous loop, the analyzer infers, by using path-length, the size relation $x{>}x'$, where x' is the value of x after executing the loop.

(3) Multiple arguments. After restricting CRSs to (a subset of) the input arguments, cost relations can have several arguments that may increase or decrease at each iteration. Importantly, the maximum number of times a given relation is executed can be a combination of several of its arguments. Most RRs solvers assume that the number of times a function is executed only depends on one argument (often a decreasing variable). In our running example, we can only find an upper bound on the number of iterations that the loop is executed, which is a combination of variables n and i, concretely "$n{-}i$".

As a result of the first two points above, CRSs are not required to define functions, but rather relations, in the sense that, given concrete values for the input variables, there may exist multiple results. These differences make existing methods for solving RRs and computer algebra systems (CAS) sometimes insufficient to bound CRS.

6.3 Existing Solvers for RRs and CRSs

Algorithms for approximating RRs have been studied by a number of researchers (see, e.g., the work by Wilf [2002]) and there are several Computer Algebra Systems (CAS) available (e.g., Mathematica, Maxima, Maple, Matlab, etc.). CAS are usually powerful tools whose aim is to solve complex recurrences. From an algebraic perspective, if we ignore the above features of CRS, cost relations would be cast as a simple class of recurrence equations. However, the recurrence equations solved by CAS can present a more complex structure. For instance, they support equations with coefficients to function calls which can be polynomials, e.g., $p(x) = 2 + x * p(x - 2)$. Importantly, the equations in CRSs generated from cost analysis of programs are never in such a complicated form as their structure is obtained from the structure of the program. As a consequence, when CRSs are indeed RRs, i.e., they do not present the three features mentioned in Section 6.2, existing CAS can often find exact solutions for them.

However, as already mentioned, CAS may be insufficient in the sense that they assume a form of the RRs which does not cover essential features of CRSs. For instance, the aforementioned CAS are not able to directly solve the CRS obtained for our running example, as they do not even accept it as an input. In order to use them, a typical approach for obtaining closed-form upper bounds consists in converting CRSs into RRs and then use an existing CAS. This requires, among other things, removing non-determinacy while preserving the worst-case solution. For this, we need to remove equations from the CRS as well as sometimes to replace inexact size relations by exact ones. For instance, in our example, we need to take the worst case cost of the three non-deterministic equations for $Main.add^c_8$. Considering that the cost of all calls to method *incr* is 4, then the worst case

corresponds to rule $Main.add_{14:A}$, which increases i by one thus maximizing the number of iterations in the loop. Still many CAS require a further non-trivial transformation to have a single decreasing argument in all relations. Neither of these transformations can be safely done in all cases. In particular, by removing some equations, we might no longer obtain the worst case and the resulting closed-form is not guaranteed to correspond to a correct upper bound (see, e.g., the work by Albert et al. [2008a]). This process typically requires a high degree of human intervention.

We believe that automatically converting CRSs into the format accepted by CASs is impractical. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes, such as those mentioned in Section 1.1. The latter problem was already identified by [Wegbreit 1975], and is the motivation for PURRS [Bagnara et al. 2005], which focuses on solving CRSs obtained by cost analysis. Indeed, PURRS has been the first system to provide, in a fully automatic way, non-asymptotic upper and lower bounds for a wide class of recurrences. It unfortunately also requires RRs to be deterministic. There is the recent work by Albert et al. [2008a] which provides an online system `http://www.clip.dia.fi.upm.es/Systems/PUBS` which infers upper bounds of CRSs originated from cost analysis. This solver is not tied to any particular programming language. We have been able to use the system in order to bound the CRSs obtained from Java bytecode programs without any modification on the generated CRSs, as we present in the next section. As described by Albert et al. [2008a], this system generalizes in several ways existing mechanisms, usually applied manually, for obtaining upper bounds in order to make them applicable to CRSs which have the distinguishing features mentioned above. Technical details are described by Albert et al. [2008a].

7. THE COSTA SYSTEM: AN IMPLEMENTATION FOR JAVA BYTECODE

This section describes the design and implementation of COSTA, an abstract interpretation-based COSt and Termination Analyzer for Java bytecode. The system receives as input a bytecode program and (a choice of) a resource of interest in the form of a cost model, and tries to obtain an upper bound of the resource consumption of the program. COSTA can deal with the non-trivial notions of cost mentioned before, i.e., the consumption of the heap, the number of bytecode instructions executed and the number of calls to user-specified methods. Additionally, COSTA tries to prove termination of the bytecode program which implies the boundedness of any resource consumption. The termination module is outside the scope of this article (see previous work [Albert et al. 2008] for details). Experimental results show that COSTA can deal with non-trivial object oriented programs which use the Java API. To the best of our knowledge, this system is the first to provide evidence that cost analysis can be applied to programs written in a realistic object-oriented programming language, in bytecode form.

7.1 The Architecture of the COSTA System

COSTA [Albert et al. 2008a; 2008b] is implemented in Prolog and uses the Parma Polyhedra Library (PPL) [Bagnara et al. 2008] for manipulating linear constraints. COSTA can handle a large subset of the Java bytecode (JBC) language [Lindholm

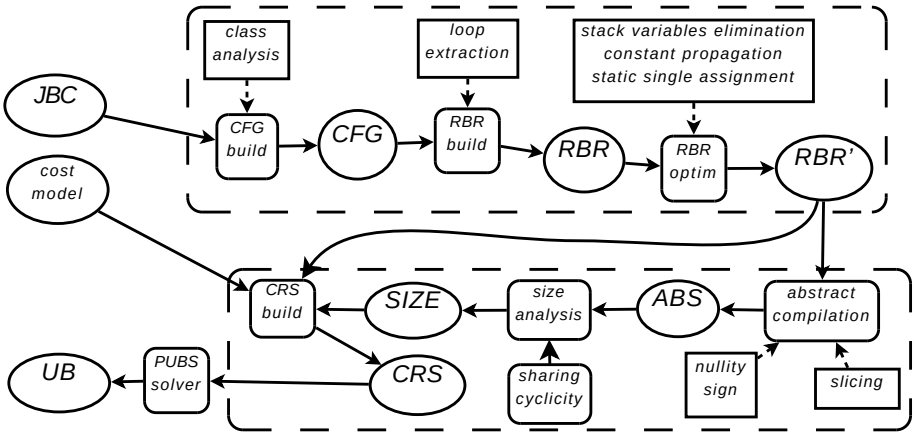


Fig. 11. Architecture of COSTA

and Yellin 1996]. It differs from the simplified bytecode language used in the formalization in several respects: (1) the instruction set includes different variants of the same instruction, according to the type of the operands (e.g., iload, aload, etc. in JBC are variants of the load functionality); (2) procedure calls can take the form invokevirtual for dynamic resolution, invokespecial for special invocation, and invokestatic for static methods; (3) methods are not forced to have a return value; (4) exceptions, either explicitly thrown in the code or resulting from semantic violations, are supported. COSTA deals with all these features and basically follows the analysis steps which are described in the previous sections. As regards *exceptions*, it handles internal exceptions (i.e., those associated to bytecodes as stated in the JVM specification), exceptions which are thrown (bytecode athrow) and possibly propagated back in methods, as well as finally clauses (even if they are compiled using the bytecode jsr). Exceptions are handled by adding edges to the corresponding handlers. When the type of the exception is not statically known, as it happens when exceptions come from calls to methods, mutually exclusive edges are generated, which capture all possible instantiations. In order to infer resource usage, COSTA provides the options of ignoring only internal exceptions, all possible exceptions or considering them all.

Figure 11 shows the overall architecture of the system. Dashed frames represent the two main parts of the analysis: (1) transforming the bytecode into a rule-based representation; and (2) actually performing the cost analysis on the rule-based program. The input and output to the system are depicted on the left: COSTA takes a Java bytecode program JBC and a description of the cost model, and yields as output an upper bound UB for its cost. Rounded boxes correspond to the main steps of the process (e.g., the one labelled CFG build), while ellipses such as the one labelled Square boxes, as class analysis, denote auxiliary analyses which allow us to obtain more precise results or to improve efficiency.

In phase (1), as depicted in the upper half of the figure, the incoming JBC is transformed into the rule-based representation (RBR) through the construction of the control flow graph (CFG), as discussed in Section 3. Several techniques make

this process more efficient and accurate: in particular, class analysis, static single assignment, loop extraction, constant propagation and stack variables elimination.

Class analysis [Spoto and Jensen 2003] (see also Section 3.2) tries to compute the set of method instances which can be actually invoked by a virtual call. This information is particularly useful when building the CFG, since it allows excluding many method instances (e.g., it can be crucial when the declared class is `Object`, as it often happens in libraries). The static single assignment (SSA) transformation (i.e., rule variables are renamed in order to guarantee that every variable is only assigned once) of the RBR helps propagate constants through the rules via unification. Knowing that a variable is actually a constant at a given program point can be very useful when performing some operations. For example, it can allow us to exclude that the second argument of a division is 0 (if a non-zero constant is propagated at that point), so that the division-by-zero exceptional behavior does not need to be considered, also it can improve the size abstraction of idiv (when the divisor is constant) which is crucial for inferring a precise upper-bound for examples such as *binary search*. Unification can also remove, in many cases, stack variables which are only used to perform simple operations, which in turn reduces the size of the abstract states in the different underlying analyses.

EXAMPLE 7.1. *By unification, the following instruction sequence (in SSA form):*

$$s_1 := n,\ s_2 := m,\ s_1' := s_1 + s_2,\ n' := s_1'$$

which is obtained from loading two local variables on the stack and storing their sum in the first argument, is reduced in COSTA *to:*

$$n' := n + m$$

□

As another important technique worth mentioning, COSTA is able to detect and extract loops from CFGs. Indeed, when analyzing bytecode, recognizing iterative structures (usually coming from loops implemented at the source code level) in the CFG may avoid the loss of information which derives from the unstructured control flow. In particular, detecting nested loops allows reasoning compositionally, one loop at a time, so that finding a cost bound from the corresponding equations is easier, and computing the cost can be done locally in the strongly connected components. A loop extraction transformation is applied to the initial CFG in order to separate sub-graphs corresponding to loops. Loop extraction has been well studied in the area of program decompilation [Allen 1970], but, to the best of our knowledge, its use in static analysis of Java bytecode is new. COSTA implements an efficient algorithm [T. Wei and Chen 2007], modified to extract loops which, in addition to having a single entry, also have a single exit (to avoid multiple return branches from loops). Whenever a loop is extracted, the corresponding sub-graph is replaced by a new instruction call_loop. Besides, a new CFG is generated for each sub-graph. Hence, after this step, there is one CFG corresponding to the entry of the method, and one graph for every loop. Due to the RBR design, calls to loops are handled in the same way as method calls.

In phase (2), depicted in the lower half of the figure, cost analysis on the rule-based representation is performed. Abstract compilation, which is helped by a

number of auxiliary static analyses, prepares the input to size analysis (Section 5). COSTA relies on a series of static analysis techniques, such as sign and nullity analysis. Such analyses help in statically excluding some specific behaviors of the program (by removing the corresponding rules): e.g., inferring that a reference o is not null at some program point allows us to disregard (i.e., not include in the program representation) the null-pointer-exception which could originate from a call to $o.m$. Removing such rules might result in sequences of rules without any branching and, in such cases, they are grouped together in a single rule.

Moreover, in the case of data structures with pointers, it is sometimes essential to know whether a pointer refers to a structure with cycles (see Section 5.2): e.g., the cost of a loop on a cyclic data structure may be unbounded. To this end, COSTA comes equipped with a sharing analysis and a (non-)cyclicity analysis which uses the information on sharing variables in order to prove the non-cyclicity of data structures.

Afterward, COSTA sets up a CRS for the selected cost model (Section 5.4). The latter is given as an input, selected among the available models. It is also trivial to define new cost models in the system by just associating a cost to each bytecode instruction, as pointed out in Section 4. Slicing (Section 6.1) of the RBR removes variables which are useless in cost analysis. Finally, COSTA integrates the dedicated upper bound solver PUBS [Albert et al. 2008a], which finds closed-form solutions for CRSs (Section 6.3).

7.2 Experimental Results

In order to assess the practicality of our approach, we present some experimental results obtained using COSTA. Though the efficiency and robustness of the system can be considerably improved, COSTA can already deal with a relatively large class of JBC programs, and gives reasonable results in terms of precision and efficiency. We plan to distribute the system as free software soon. Currently, it can be tried out through a web interface available from the COSTA web site: `http://costa.ls.fi.upm.es`.

COSTA allows choosing between several options, by specifying, for instance,

(1) whether the code of external libraries (i.e., methods in the Java API which do not belong to the benchmark itself, but are called from the methods in the benchmark) should be also analyzed;

(2) whether auxiliary analyses (sign, nullity, slicing, constant propagation) should be included, thus possibly improving both precision and performance;

(3) whether input-output size relations have to be computed (Section 5.3);

(4) if exceptions, either explicitly thrown in the code or resulting from semantic violations, have to be taken into account;

(5) which cost model has to be considered.

The system can deal with most features of JBC. Non-sequential code, dynamic code generation and reflection are not supported. Currently, COSTA handles bytecode programs for Java SE 1.4.2_13. However, there is no fundamental reason for not supporting more recent Java versions and we plan to extend COSTA to also handle Java 5 and 6 soon. As for native code, i.e., methods not implemented in

Bench	#B	#M	#C	#R	#R_o	#E
copy	108	4	3	78	56	55
divByTwo	15	1	1	17	15	16
binsearch	68	1	1	31	30	30
fact	14	1	1	11	10	9
arrayReverse	27	1	1	28	24	25
concat	44	1	1	49	43	45
add	105	4	5	32	27	27
merge	170	3	2	89	61	59
power	15	1	1	11	10	9
copy_cons	92	5	4	56	34	31
evenDigits	31	2	1	34	30	33
selectOrd	51	1	1	58	55	57
doSum	27	2	1	28	25	19
multiply	58	1	1	75	64	67
hanoi	20	1	1	13	11	9
fibonacci	18	1	1	14	13	11
copy_bst	123	6	4	119	73	67
as_push	658	7	6	104	70	59
ns_pop	666	9	7	132	90	77
nq_dequeue	748	8	7	128	90	78
nl_prev	1024	10	9	212	140	118
bst_find	3470	28	15	543	410	415

Table I. Statistics about the Analysis Process

Java, calls to native methods are shown in upper bounds as symbolic constants, since the code for those methods is not written in Java and COSTA cannot analyze them. This could be further improved by providing assertions which describe the cost of the native method for the different cost models and (optionally) a safe approximation of their input-output behavior, but do not support it yet.

Table I presents some figures about the benchmark programs used. Column **Bench** indicates the name of the benchmark. Each benchmark program has been analyzed for a particular method as starting point. The analyzer then pulls all other methods used by the initial method as indicated by class analysis. This results in loading a number of bytecode instructions, shown in column **#B**, of methods, given in column **#M**, from a number of classes, indicated by **#C**. The analyzer then obtains a rule-based representation of the program. Column **#R** shows the number of rules in such representation, column #$\mathbf{R}_o$ shows then number of rules in the optimized representation after performing nullity and sign analyses and some optimizations. The reduction is significant and it is crucial for efficiency and accuracy of subsequent phases. Finally, **#E** shows the number of equations in the final cost relation system.

Both the Java source and the bytecode for all programs used as benchmarks is available at the COSTA web interface. We consider two sets of benchmarks, and the benchmarks in each set are presented in increasing complexity order. As will be seen in Table III, the benchmarks range from constant to exponential complexity. The first set of benchmarks used, from copy to copy_bst, are Java programs which represent classical examples in complexity analysis. The second

Bench	RBR	Opt	Ana	Size	CRS	UB	Sim	Total	TR
copy	21	50	66	362	12	82	0	594	8
divByTwo	2	10	7	54	2	10	0	87	5
binsearch	6	30	24	207	4	158	0	430	14
fact	4	6	6	2	0	8	0	28	3
arrayReverse	5	20	24	137	4	37	0	226	8
concat	10	40	74	348	6	111	0	589	12
add	14	23	18	78	2	94	0	228	7
merge	20	77	348	258	13	270	0	986	11
power	0	7	10	5	0	14	0	38	3
copy_cons	15	35	50	48	4	49	0	201	4
evenDigits	8	27	12	102	3	22	0	175	5
selectOrd	11	51	68	1556	9	253	0	1948	34
doSum	5	19	11	31	0	13	0	81	3
multiply	15	89	225	2411	16	859	0	3616	48
hanoi	4	8	30	10	0	150	0	202	16
fibonacci	5	9	11	14	0	16	0	55	4
copy_bst	30	73	138	513	16	630	0	1399	12
as_push	54	59	160	17	14	67	0	372	4
ns_pop	58	84	205	27	22	100	0	496	4
nq_dequeue	62	82	214	33	22	162	2	577	5
nl_prev	104	150	586	47	53	281	0	1220	6
bst_find	265	533	1765	838	448	1137	1758	6743	12

Table II. Runtimes of Analysis

set of benchmarks are programs taken from the `net.datastructures` Java package [Goodrich et al. 2003], which contains a collection of Java interfaces and classes that implement fundamental data structures and algorithms. They are accessible for non-commercial purposes both in source and bytecode form at [Goodrich et al. 2003] and are discussed in detail in [Goodrich and Tamassia 2004]. The reason for using such programs is that they make intensive use of object-oriented features, the implementation techniques and the algorithms used are comparable to those in java.util.*, and they remain reasonably sized and comprehensible. Among all classes in the `net.datastructures` package, the ones we have selected as starting point for our experiments are: `ArrayStack`, `NodeStack`, `NodeQueue`, `NodeList`, and `BinarySearchTree`. In order not to make the experimental table too large, we only show the analysis results for one method per class. The methods considered are, respectively, `push`, `pop`, `dequeue`, `prev`, and `find`. As Table II shows, the number of bytecode instructions involved is larger than 650 in all experiments in this set, reaching 3470 for the bst_find benchmark, which involves 28 methods from 15 different classes or interfaces.

Table II shows the runtimes of the different phases of the analysis. All experiments have been performed on an Intel Core 2 Quad Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.28-11. Times are in milliseconds and have been computed as the average of five runs. As regards the options for the analysis previously described, in our experiments we (1) analyze Java API, (2) activate all auxiliary analyses (3) compute input-output size relations (4) consider all exceptions. Column **RBR** shows the time taken to obtain the rule-based representation of the program. This includes computing the CFGs and performing class analysis. Then,

Opt indicates the time needed to obtain the optimized rule-based representation. This includes the times for nullity and sign analyses. Column **Ana** corresponds to the time needed by the different preliminary analyses which are required before performing size analysis, whose time is shown in Column **Size**. Column **CRS** is the time taken to obtain the cost relation system. The time taken to obtain a closed form solution is shown in **UB**. Then, a post-processing phase is performed which is in charge of syntactically simplifying as much as possible the upper bound obtained. This time is shown in **Sim**. The total time taken is displayed in **Total**. Finally, **TR** aims at evaluating how the analysis time varies w.r.t. the size of the program being analyzed. For this, we divide the total analysis time by the number of rules in the rule-based representation of the program. Thus, this number can be interpreted as the average number of milliseconds required to analyze a rule. We argue that, at least in our experiments, analysis time is acceptable. It ranges from 3 to 48 milliseconds. Interestingly, the analysis time per rule does not seem to increase significantly with the number of rules. There are important variations though, but this probably has to do more with other features of the code, such as the number of loops and whether loops are nested or not. It is important to mention that the current implementation of several components of the system is not optimized for efficiency, since the main aim for the time being has been to see whether the approach allows obtaining useful results. Now that the applicability of the approach is proved, we are working on more efficient and robust implementations.

It is worth mentioning that **Size** includes both the time for abstract compilation of instructions and for inferring the input-output size analysis, which can be rather expensive. As already pointed out in Section 5.3, sometimes such step is not required in order to infer upper bounds, in particular, when loop guards do not depend on the return value from a method.

The simplification process also deserves some comments. As it can be seen in Table II, in most cases the simplification time is negligible. However, it is well known from the first works on the topic (see e.g. [Wegbreit 1975]), that one of the important threats to the applicability of cost analysis is that the computed closed forms can rapidly grow very large. Therefore, it is essential to have a smart simplification procedure available. Our current simplifier succeeds to significantly simplify closed-forms. As discussed above, it is not yet optimized for efficiency and its running time can be important in some cases, as in **bst_find**. We argue that it is a good investment to have a powerful simplifier since, due to the compositionality of cost analysis, such simplified closed-forms can later be used for analyses of other programs.

Table III shows the closed-form upper-bounds computed by COSTA for the benchmarks shown in the previous tables. In all cases, we show the upper bound obtained when using the $\mathcal{M}_{inst}$ cost model, which counts the number of bytecode instructions executed (indicated by a ln in the last column, **M**). We also show another upper bound w.r.t. the $\mathcal{M}_{heap}$ cost model, which counts the number of bytes allocated in the heap, but only for those programs which actually contain instructions for allocating objects or arrays. For the programs for which an upper bound w.r.t. the $\mathcal{M}_{heap}$ cost model is not shown, COSTA obtains zero as upper bound of their heap consumption.

Note that in Tables I and II we only show one entry per benchmark, regardless of

Bench	Entry	UB	M
copy	copy(A)	228	In
copy	copy(A)	52	By
divByTwo	divByTwo(A)	6+8*log(2,1+nat(2*A-1))	In
binsearch	binarySearch(A,B)	24+24*log(2,1+nat(2*A-1))	In
fact	fact(A)	4+9*nat(A)	In
arrayReverse	arrayReverse(A)	22+14*A	In
arrayReverse	arrayReverse(A)	4*A	By
concat	concat(A,B)	37+(11*(A+B)+11*B)	In
concat	concat(A,B)	4*(A+B)	By
add	add(A,B,C)	16+18*nat(1+B)	In
merge	merge(A,B)	26+29*nat(A+B-1)	In
merge	merge(A,B)	8+8*nat(A+B-1)	By
power	power(A,B)	4+10*nat(B)	In
copy_cons	copy(A)	23+21*nat(A-1)	In
copy_cons	copy(A)	8+8*nat(A-1)	By
evenDigits	evenDigits(A)	9+nat(A)*(16+8*log(2,1+nat(2*A-3)))	In
selectOrd	selectOrd(A)	36+(nat(A-2)*(40+17*nat(A-2))+17*nat(A-2))	In
doSum	doSum(A)	6+nat(1+A)*(16+9*nat(1+A))	In
multiply	multiply(A,B,C)	$52+38*C+58*B+58*B*C+26*B^2+26*B^2*C$	In
hanoi	hanoi(A,B,C,D)	-17+20*pow(2,nat(A))	In
fibonacci	fibonacciMethod(A)	-13+18*pow(2,nat(A-1))	In
copy_bst	copy(A)	-45+88*pow(2,nat(A-1))	In
copy_bst	copy(A)	-12+24*pow(2,nat(A-1))	By
as_push	push(A,B)	38+c(fST)	In
as_push	push(A,B)	16+c(fST)	By
ns_pop	pop(A)	36+c(fST)	In
ns_pop	pop(A)	16+c(fST)	By
nq_dequeue	dequeue(A)	31+c(fST)	In
nq_dequeue	dequeue(A)	16+c(fST)	By
nl_prev	prev(A,B)	124+3*c(fST)	In
nl_prev	prev(A,B)	48+3*c(fST)	By
bst_find	find(A,B)	506+(nat(A-3)*(283+5*c(fST))+10*c(fST))	In
bst_find	find(A,B)	160+(nat(A-3)*(80+5*c(fST))+10*c(fST))	By

Table III. Upper Bounds Computed

whether it is analyzed for one or two cost models. The reason for this is that Table I is identical for both cost models and the **RBR**, **Opt**, **Ana**, and **Size** columns in Table II are also the same. Thus, the only figures which can differ are **CRS**, **UB**, and **Sim** (and of course **Total** and **TR**, which depend on them). However, in all cases, the difference in run-time is rather small. Thus, we have preferred skipping this info instead of further cluttering Table II. In the upper bounds shown we use the function $nat(X)$, which stands for *natural* and which is defined as $nat(X) = max(0, X)$. Since we analyze the code of API methods called by the programs, we are able to obtain upper bounds without symbolic constants for all programs, unless they access some API method which is native. In our experiments, this is the case with the `fillInStackTrace` method from the `java.lang.Throwable` class, which in our upper bounds is represented as the symbolic constant c(fST).

We argue that the computed upper bounds are useful since they are both reasonably accurate and simple. As can be seen in the table, the upper bounds obtained

can be constant, as in the case of copy, logarithmic, as in binSearch, linear, as in arrayReverse, n-log-n, as in evenDigits, quadratic, as in selectOrd, polynomial, as in multiply, and exponential, as in hanoi.

As regards the second set of experiments, it is worth mentioning that COSTA has obtained precise upper bounds in all cases. In particular, it has been able to infer that the first four methods have a constant cost both in time and in memory, in spite of the relatively large number of methods and classes involved. The fifth and final experiment is especially relevant since it illustrates how the path-length abstract domain allows computing precise upper bounds for programs which handle dynamically allocated, non linear, data structures. The **find** operation searches for a key in a binary search tree. It is well-known that the worst case cost of such search is linear on the height of the tree, as only one branch of the tree can be explored. This is precisely what the computed upper bound tells us, since A in the expression $\mathrm{nat}(\mathrm{A}-3)*(283+5*\mathrm{c}(\mathrm{fST}))$ stands for the height of the corresponding tree.

8. RELATED WORK

Since the advent of mobile code, the analysis of Java bytecode has become an active research area and a number of analysis tools are available. Especially relevant are the analyses developed on the Soot framework [Vallee-Rai et al. 1999] and the generic analyzer Julia [Spoto 2005]. Soot is a framework for the development of analyses for Java bytecode which already includes points-to analysis, purity analysis, and dynamic data structure analysis, among others. The most similar part between these systems and COSTA is the transformation of the bytecode into an intermediate (procedural) representation. Indeed, intermediate representations are common practice to develop analysis and transformations on JBC. Of relevant importance is BoogiePL [Lehner and Müller 2007] as well. The main differences with our RBR are: (1) they do not provide a uniform treatment of all kinds of loops by means of recursion, (2) they do not perform the loop extraction transformation we propose which is important for compositionality in cost analysis; and (3) the intermediate representation called Shimple in Soot performs SSA, but neither Shimple nor BoogiePL do stack variable elimination as COSTA does. In our representation, in one pass, we can eliminate almost all stack variables, which results in a more efficient subsequent size analysis. The Java bytecode analyzer Julia [Spoto 2005] provides a generic analysis engine for which sharing analysis, class analysis, non-nullness analysis, information flow analysis, escape analysis, constancy analysis, and static initialisation analysis have been integrated. Neither Julia nor Soot include a cost analysis, though Julia also contains implementations of some of the pieces (in particular the class, nullity, sharing and cyclicity analysis) which are required in the size analysis component, as discussed in Section 5.

Focusing on cost analysis, important effort has been devoted to extend the general cost analysis framework originating from the work by Wegbreit [1975] to different languages and programming paradigms. The main objective in this task is to define a cost analysis framework in which it is possible to generate CRS from the programs in the corresponding language. Much of the work on automatic cost analysis is in the context of high-level declarative languages, whose recursive structure simplifies the process of generating cost relations. In general, these analyses do not con-

sider languages with a heap, and they do not deal with objects and exceptions as in our case. Below we review several frameworks defined for the corresponding programming paradigms.

Cost Analysis in Functional Programming. Early work on cost analysis [Wegbreit 1975; Le Metayer 1988; Rosendahl 1989] was developed for a first order subset of Lisp. Rosendahl [1989] presented a system based on transforming a program into a step-counting version which was then analyzed by relying on abstract interpretation. The result of such analysis was expressed as a CRS which was attempted then to be transformed into a closed form by relying on a series of source-to-source transformations. Theoretical advances for analyzing lazy functional languages were made by Wadler [1988] and Bjerner and Holmstrom [1989]. They used projections and demand analysis to model a call-by-need reduction strategy of typed lambda calculus. Still in the context of functional languages, Sands [1995] extends the technique of cost counting programs mentioned above [Rosendahl 1989; Le Metayer 1988] to higher-order programs. The recent work by Jouannaud and Xu [2006] describes a complexity analysis for programs extracted from proofs carried out with the Coq proof assistant. The generated CRSs are solved in this case by relying on MAPLE. Again, the first transformational part is not required and size analysis does not have to deal with object-oriented features. Benzinger [2004] presents an automatic complexity analysis for computing upper bounds on the time complexity of higher-order Nuprl programs. The analysis derives recursive cost equations which are passed to Mathematica.

There exist approaches to cost analysis based on a type-and-effect systems [Bartoletti et al. 2007; Simões et al. 2006; Vasconcelos and Hammond 2003]. Type-and-effect systems [Nielson et al. 2005] are a well-known technique for automatic program analysis. They main difference w.r.t. abstract interpretation approaches like ours is that they avoid have the implementation of specialised inference engines that may be required by abstract interpretation approaches and they simplify the construction of the soundness proofs through analogy with similar and well-understood proofs for the underlying type system. The latest work by Simões et al. [2006] uses a type-and-effect system based on Hindley-Milner types to expose constraints on sized types [Hughes et al. 1996] for higher order, recursive functional programs, to provide improved quality of cost analysis. Apart from the underlying differences between the considered languages, in contrast to our proposal, this approach to cost analysis is restricted to linear upper bounds. Besides, the language does not support recursion and the analysis is restricted to a cost model that counts the number of steps. Bartoletti et al. [2007] propose an extension of the λ-calculus to ensure that resources are correctly used. They also rely on a type and effect system to over approximate the set of histories of events (i.e., the usage of resources) that a program can generate at runtime. A model-checking tecnique then validates such approximations. In essence, this work is focused on the enforcement of resource usage policies, but their techniques cannot be used to generate upper bounds on the resource usage as our method does.

All in all, we conclude that in functional programming, cost analysis focuses on dealing with higher-order functions and lazy evaluation. Due to the high-level and recursive nature of functional programs, the first transformation to obtain a

rule-based representation is greatly simplified in this setting. Also, size analysis in functional programming differs from ours as it does not support object-oriented features.

Cost Analysis in Logic Programming. One of the first cost analysis frameworks was developed by Debray and Lin [1993] in the context of logic programming. In this setting, cost analysis needs to consider peculiar features of logic languages, such as approximating the number of solutions (due to non-deterministic computations), type and mode inference, and non-failure information. The CASLOG system [Debray and Lin 1993] was designed to solve CRSs for logic program and it is currently used in the `CiaoPP` system [Hermenegildo et al. 2005]. As in functional programming, obtaining CRSs is simplified by the fact that they already start from a recursive programming language where recursion is the only form of iteration. Also, size analysis in logic programming differs from ours as it does not support object-oriented features. The cost analysis integrated in the `CiaoPP` system includes a resource analysis [Navas et al. 2007] based on a size analysis for logic programs and hence differs fundamentally from ours.

Cost Analysis in Imperative Programming. In the imperative programming paradigm, most of the work has been done by the real-time and embedded systems community. It has mainly focused on real-time aspects, with major inroads made in WCET (worst case execution time) analysis, see e.g. [Eisinger et al. 2006], which is technically different from our cost analysis. There is also work which studies the relationship between syntactical constructions of programming languages and their computational complexity [Kristiansen and Jones 2005; Ben-Amram et al. 2008]. These analyses are developed on simple imperative languages which are far from our bytecode and, in contrast to our work, complexity classes are infered rather than CRSs. Note that our CRSs are valid not only to infer the complexity class but also to compute non-asymptotic upper bounds. Marion and Pèchoux [2007] show how to apply sub-interpretation (firstly used in first order functional programming to deal with computational complexity) to object-oriented programs without recursion in order to provide upper bounds on the stack usage. The approach does not follow the cost analysis framework originating from [Wegbreit 1975] and thus it is not based on the generation of CRS. Also, it is restricted to polynomial bounds and to the particular resource of stack usage. More recent work develops cost analyses to estimate the memory consumption. In particular, Braberman et al. [2008] describe a technique for Java-like languages which computes symbolic polynomial approximations of the amount of memory required by a program. The work by Chin et al. [2008] studies the memory consumption (including both heap space and stack usage) of low-level programs which are similar to our bytecode programs. In both cases, the analyses are less general than ours, both in the kind of properties they can estimate (specific to memory consumption) and in the kind of upper bounds that they can generate (polynomial bounds).

Resource Usage Certification. As mentioned in Section 1, resource usage certification [Crary and Weirich 2000; Aspinall et al. 2005; Hofmann and Jost 2003; Chander et al. 2005; Niggl and Wunderlich 2006] proposes the use of security properties involving cost requirements, i.e., that the untrusted code adheres to specific

bounds on resource consumption. Our work shows, for the first time, that it is possible to automatically generate resource bounds guarantees, not restricted to polynomial bounds, for a realistic mobile language. Related work in the context of Java bytecode includes the work in the MRG project [Aspinall et al. 2005], which can be considered complementary to ours. MRG focuses on building a proof-carrying code [Necula 1997] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The cost model which has been used to develop the analysis is heap consumption, since applications to be deployed on devices with a limited amount of memory, such as smartcards, must be rejected if they require more memory than that available. The framework is restricted to polynomial bounds and to the above cost model, while our cost analysis can infer a wider set of bounds (including exponential, algorithmic, etc.) and it is parametric with respect to the cost model. More related work is the one proposed by Cachera et al. [2005], where a resource usage analysis is presented. Again, this work focuses on one particular notion of cost, memory consumption, and it aims at verifying that the program executes in bounded memory by making sure that the program does not create new objects inside loops, but it does not infer resource usage bounds. The analysis has been certified by proving its correctness using the Coq proof assistant.

9. CONCLUSIONS AND FUTURE WORK

The presented framework is, to the best of our knowledge, the first automatic approach to the cost analysis of object-oriented bytecode, i.e., compiled code. The analysis is based on generating, at compile-time, *cost relation systems* for an input bytecode w.r.t. a *cost model*. Such relations provide useful approximations of the computational cost, provided an accurate *size analysis* is used to establish relationships between arguments. Essentially, the sources of inaccuracy in size analysis can be: (1) the dependence of the control flow on values which are not captured by the abstraction, such as non-integer values, numeric fields, multidimensional arrays, and cyclic data structures; (2) the imprecision in abstracting (non-linear) arithmetic instructions, and in using domain operations such as *widening*. In such cases, cost relations can still be set up, but might be not precise enough to be useful. Clearly, progress in the area of size analysis for object-oriented languages will be directly applicable to this cost analysis framework as the size analysis is an independent component.

Our cost analysis at the level of the rule-based representation is compositional, e.g., we can analyze one piece of code and then reuse its result. This is because whenever we call a procedure we always know which procedure we refer to (there are no virtual calls). When we analyze Java bytecode programs, in general, we cannot analyze a piece of code without a context because, due to virtual calls, we might invoke different methods which are defined in those contexts (this is the well-known problem of call-backs). In order to analyse in a context-independent way, we need assumptions about the classes that we have in the memory and the possible sharing and the cyclicity between the data structures. Our experimental results show that the presented approach is able to obtain useful cost relations for a relatively large class of object oriented programs which use Java libraries.

We believe that our work opens the door to the application of *resource usage analysis* in the context of realistic programming languages like Java bytecode. The

theoretical framework has already been the basis of two applications of cost analysis, related to different cost models: (1) inference of the number of *executed bytecode instructions* of a series of well known programs used in research on complexity analysis [Albert et al. 2007b]; and (2) inference of *heap consumption* of object-oriented programs which make extensive use of the heap [Albert et al. 2007]. Cost relations were refined in this case in order to take into account the heap space which can be safely deallocated by the *garbage collector* upon exit from a method (as approximated by an escape analysis [Blanchet 1999]). However, in order to infer the *live heap space*, i.e., the maximum of the size of the *live* data on the heap during a program's execution, our cost analysis framework cannot directly be applied. The reason is that live heap usage is a resource which requires reasoning on the memory consumed at *all program states* along an execution, while in our framework we observe the consumption at the *final* state only. A live heap space analysis has been recently proposed in [Albert et al. 2009] which generates cost relation systems in a different way than us.

The transformation to the rule-based representation is interesting *per se* and can be used to develop other kinds of analysis. Indeed, it has been used to prove the *termination* of Java bytecode programs [Albert et al. 2008]. Essentially, after performing size analysis on the rule-based representation, one obtains a *constraint logic program* on which it is straightforward to apply existing analysis techniques and prove termination results. Future work will basically focus on extending both the theoretical foundations and the practical implementation in order to handle a larger class of programs, and obtain improvements both in terms of efficiency and accuracy. For example, one of the most challenging problems is to account for iterative structures where the number of iterations depends on *numeric fields*. Here, an approach working in all cases might not be practical; however, heuristics may allow us to account for special, simple but quite common cases which can significantly enlarge the class of analyzable programs. A first step in this direction, in the context of termination analysis, has been taken in [Albert et al. 2008b].

Acknowledgments

We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped to improve this article. Preliminary versions of this work appeared in the Proceedings of ESOP'07 [Albert et al. 2007a], FMCO'07 [Albert et al. 2008b] and Bytecode'08 [Albert et al. 2008a]. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

REFERENCES

Adachi, A., Kasai, T., and Moriya, E. 1979. A theoretical study of the time analysis of programs. In *MFCS*, J. Becvár, Ed. Lecture Notes in Computer Science, vol. 74. Springer, 201–207.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers – Principles, Techniques and Tools*. Addison-Wesley.

Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., and Zanardini, D. 2008. Termination Analysis of Java Bytecode. In *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, G. Barthe and F. de Boer, Eds. Lecture Notes in Computer Science, vol. 5051. Springer-Verlag, Berlin, Oslo, Norway, 2–18.

Albert, E., Arenas, P., Genaim, S., and Puebla, G. 2008a. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, M. Alpuente and G. Vidal, Eds. Lecture Notes in Computer Science, vol. 5079. Springer-Verlag, 221–237.

Albert, E., Arenas, P., Genaim, S., and Puebla, G. 2008b. Dealing with numeric fields in termination analysis of java-like languages. In *10th Workshop on Formal Techniques for Java-like Programs*, M. Huisman, Ed.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2007a. Cost Analysis of Java Bytecode. In *16th European Symposium on Programming, ESOP'07*, R. D. Nicola, Ed. Lecture Notes in Computer Science, vol. 4421. Springer, 157–172.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2007b. Experiments in Cost Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*. Electronic Notes in Theoretical Computer Science, vol. 190, Issue 1. Elsevier - North Holland, 67–83.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2008a. COSTA: A Cost and Termination Analyzer for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'08)*. Electronic Notes in Theoretical Computer Science. Elsevier, Budapest, Hungary. To appear.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2008b. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*. Number 5382 in LNCS. Springer-Verlag, 113–133.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. 2008c. Removing Useless Variables in Cost Analysis of Java Bytecode. In *ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV08)*. ACM Press, New York, Fortaleza, Brasil, 368–375.

Albert, E., Genaim, S., and Gómez-Zamalloa, M. 2007. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*. ACM Press, New York, NY, USA, 105–116.

Albert, E., Genaim, S., and Gómez-Zamalloa, M. 2009. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*. ACM Press, New York, NY, USA.

Albert, E., Puebla, G., and Hermenegildo, M. 2008. Abstraction-Carrying Code: A Model for Mobile Code Safety. *New Generation Computing 26,* 2 (March), 171–204.

Allen, F. 1970. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*. 1–19.

Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., and Stark, I. 2005. Mobile Resource Guarantees for Smart Devices. In *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds. LNCS, vol. 3362. Springer, 1–27.

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming 72,* 1–2, 3–21.

Bagnara, R., Pescetti, A., Zaccagnini, A., and Zaffanella, E. 2005. PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Tech. rep. `arXiv:cs/0512056` available from `http://arxiv.org/`.

Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E., and Zolo, T. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.

Bartoletti, M., Degano, P., Ferrari, G. L., and Zunino, R. 2007. Types and effects for resource usage analysis. In *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007*. Lecture Notes in Computer Science, vol. 4423. Springer, 32–47.

Ben-Amram, A. M., Jones, N. D., and Kristiansen, L. 2008. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008*. Lecture Notes in Computer Science, vol. 5028. Springer, 67–76.

Benoy, F. and King, A. 1997. Inferring Argument Size Relationships with CLP(R). In *Workshop on Logic-based Program Synthesis and Transformation (LOPSTR)*. Lecture Notes in Computer Science, vol. 1207. Springer-Verlag, 204–223.

Benzinger, R. 2004. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci. 318,* 1-2.

Bjerner, B. and Holmstrom, S. 1989. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*. ACM Press, 157–165.

Blanchet, B. 1999. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*. ACM, 20–34.

Bossi, A., Cocco, N., and Fabris, M. 1991. Proving termination of logic programs by exploiting term properties. In *TAPSOFT, Vol.2*. Lecture Notes in Computer Science, vol. 494. Springer, 153–180.

Braberman, V., Fernández, F., Garbervetsky, D., and Yovine, S. 2008. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory management (ISMM)*. ACM, New York, NY, USA.

Bruynooghe, M., Codish, M., John P. Gallagher, Genaim, S., and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems 29,* 2 (April).

Cachera, D., Jensen, T., Pichardie, D., and Schneider, G. 2005. Certified memory usage analysis. In *13th International Symposium on Formal Methods (FM'05)*. Number 3582 in LNCS. Springer-Verlag, 91–106.

Chander, A., Espinosa, D., Islam, N., Lee, P., and Necula, G. 2005. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*. LNCS, vol. 3444. Springer.

Chin, W.-N., Nguyen, H., Popeea, C., and Qin, S. 2008. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory management (ISMM)*. ACM, New York, NY, USA.

Cousot, P. and Cousot, R. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*. 238–252.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *POPL*.

Craig, S.-J. and Leuschel, M. 2005. Self-Tuning Resource Aware Specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM Press, New York, NY, USA, 23–34.

Crary, K. and Weirich, S. 2000. Resource Bound Certification. In *POPL'00*. ACM, 184–198.

Debray, S. K. and Lin, N. W. 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems 15,* 5 (November), 826–875.

Eisinger, J., Polian, I., Becker, B., Metzner, A., Thesing, S., and Wilhelm, R. 2006. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE Computer Society, 15–20.

Genaim, S. and Spoto, F. 2008. Constancy analysis. In *10th Workshop on Formal Techniques for Java-like Programs*, M. Huisman, Ed.

Goodrich, M. and Tamassia, R. 2004. *Data Structures and Algorithms in Java*, 3rd ed. John Wiley.

Goodrich, M., Tamassia, R., and Zamore, R. 2003. The net.datastructures package, version 3. Available at http://net3.datastructures.net.

Hermenegildo, M., Albert, E., López-García, P., and Puebla, G. 2005. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press.

Hermenegildo, M., Puebla, G., Bueno, F., and López-García, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

Hofmann, M. and Jost, S. 2003. Static prediction of heap space usage for first-order functional programs. In *POPL*.

Hughes, J., Pareto, L., and Sabry, A. 1996. Proving the correctness of reactive systems using sized types. In *POPL*. 410–423.

Jouannaud, J. and Xu, W. 2006. Automatic Complexity Analysis for Programs Extracted from Coq Proof. *ENTCS*.

Kristiansen, L. and Jones, N. D. 2005. The flow of data and the complexity of algorithms. In *CiE*, S. B. Cooper, B. Löwe, and L. Torenvliet, Eds. Lecture Notes in Computer Science, vol. 3526. Springer, 263–274.

Le Metayer, D. 1988. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems 10,* 2 (April), 248–266.

Lehner, H. and Müller, P. 2007. Formal translation of bytecode into BoogiePL. In *Bytecode'07*. ENTCS. Elsevier, 35–50.

Lindholm, T. and Yellin, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

Marion, J.-Y. and Pèchoux, R. 2007. Resource control of object-oriented programs. In *International LICS affiliated Workshop on Logic and Computational Complexity (LCC 2007)*. Wroclaw, Poland.

Miné, A. 2006. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES*, M. J. Irwin and K. D. Bosschere, Eds. ACM, 54–63.

Navas, J., Mera, E., López-García, P., and Hermenegildo, M. 2007. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*. LNCS, vol. 4670. Springer-Verlag, 348–363.

Necula, G. 1997. Proof-Carrying Code. In *Proc. of ACM Symposium on Principles of programming languages (POPL)*. ACM Press, 106–119.

Nielson, F., Nielson, H. R., and Hankin, C. 2005. *Principles of Program Analysis*. Springer. Second Ed.

Niggl, K.-H. and Wunderlich, H. 2006. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM J. Comput. 35,* 5, 1122–1147.

Puebla, G. and Ochoa, C. 2006. Poly-Controlled Partial Evaluation. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*. ACM Press, 261–271.

Rosendahl, M. 1989. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 144–156.

Rossignoli, S. and Spoto, F. 2006. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. LNCS, vol. 3855. S-V.

Sands, D. 1995. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput. 5,* 4.

Secci, S. and Spoto, F. 2005. Pair-sharing analysis of object-oriented programs. In *Static Analysis Symposium (SAS)*. 320–335.

Simões, H. R., Hammond, K., Florido, M., and Vasconcelos, P. B. 2006. Using intersection types for cost-analysis of higher-order polymorphic functional programs. In *Types for Proofs and Programs, International Workshop, TYPES 2006*. Lecture Notes in Computer Science, vol. 4502. Springer, 221–236.

Spoto, F. 2005. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*. Glasgow, Scotland.

Spoto, F., Hill, P. M., and Payet, E. 2006a. Path-length analysis for object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*.

Spoto, F., Hill, P. M., and Payet, E. 2006b. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*. Electronic Notes in Theoretical Computer Science. Elsevier.

Spoto, F. and Jensen, T. 2003. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst. 25,* 5, 578–630.

T. Wei, J. Mao, W. Z. and Chen, Y. 2007. A new algorithm for identifying loops in decompilation. In *SAS'07*. LNCS 4634. 170–183.

Tip, F. 1995. A Survey of Program Slicing Techniques. *J. of Prog. Lang. 3*.

Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. 1999. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 125–135.

Vasconcelos, P. and Hammond, K. 2003. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*. Lecture Notes in Computer Science, vol. 3145. Springer-Verlag, 86–101.

Wadler, P. 1988. Strictness analysis aids time analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 119–132.

Wegbreit, B. 1975. Mechanical Program Analysis. *Comm. of the ACM 18,* 9.

Wilf, H. S. 2002. *Algorithms and Complexity.* A.K. Peters Ltd.

PROOFS (added for reviewers' convenience and to appear in an electronic appendix)

A. EQUIVALENCE BETWEEN A BYTECODE PROGRAM AND ITS CORRESPONDING RBR

A.1 Proof of Theorem 3.10

In this section we prove Theorem 3.10 by induction on the length of the bytecode trace. We prove that the bytecode and the corresponding RBR traces satisfy a stronger notion of equivalence which implies the one of Definition 3.8. The new notion of equivalence is needed in order to define when two configurations have been obtained using a *similar* sequence of execution steps. For initial and final configurations the definition of equivalence remains the same as in Definition 3.8, i.e., we require the single activation records to be equivalent and that the heap structures be identical. For non-final configurations, *in addition* to the requirements in Definition 3.8, we require that the structure of the configurations satisfies an additional property that we explain below.

In order to define this additional property, let us first examine the structure of bytecode configurations. If a bytecode configuration is not initial or final, then it is of the form $C \equiv a_j \dots a_0; h$ where $j \geq 1$ and for all $i < j$ the activation record a_i is of the form $\langle m, pc, lv, stk\rangle$ such that $m[pc-1]$ is an invocation instruction to a method m' (resolved to m' in the case of virtual calls), and the next activation record a_{i+1} corresponds to the method m', namely of the form $\langle m', pc', lv', stk'\rangle$. Now given an RBR configuration $RC \equiv a^{j'}_{rr} \dots a^0_{rr}; h_{rr}$, we say that it is equivalent to C, denoted as before by $C \approx RC$, if in addition to the requirements of Definition 3.8, it holds that $j' \geq j$, and it includes $j-1$ activation records $a^{i_1}_{rr}, \dots, a^{i_{j-1}}_{rr}$ where $i_1 = 0 < i_2 < \dots < i_{j-1} < j$ such that:

(1) $a^{i_1}_{rr} = a^0_{rr} = \langle start[s_1, out], \epsilon, lv'_0\rangle$;

(2) for all k, where $0 < k \leq j-1$, let the bytecode activation record a_k be $\langle m, pc, lv, v_t \cdots v_1\rangle$ where: m is a method with f local variables (including l_0); the instruction $m[pc-1]$ is an invocation to m'' with f' arguments (excluding l_0); and suppose that for that specific call the type of the object whose method we are calling is d. Then the corresponding RBR activation record $a^{i_k}_{rr}$ is of the form $\langle m_{pc-1:d}[s_{t-f'}, out], m_{pc}(\bar{l}, s_1, \dots, s_{t-f'}), lv'\rangle$ where $\bar{l} = \langle l_0, \dots, l_{f-1}\rangle$, such that:

   (a) If resolving m'' w.r.t. the class d results in m', then there exists a rule that corresponds to a dispatching block for the invocation $m[pc-1]$ of the form:

$$m_{pc-1:d}(\bar{l}, s_1, \dots, s_t, out) \leftarrow m'(s_{t-f'}, \dots, s_t, s_{t-f'}), \\ m_{pc}(\bar{l}, s_1, \dots, s_{t-f'}, out)$$

   (b) for any local variable i (where $0 \leq i < f$) in m we have $lv(i) = lv'(l_i)$ and for any stack element v_i in a_k (where $1 \leq i \leq t$) we have $lv'(s_i) = v_i$;

(3) any RBR activation record which is not one of $\{a^{i_1}_{rr}, \dots, a^{i_{j-1}}_{rr}, a^j_{rr}\}$ is of the form $\langle p[out, out], \epsilon, lv''\rangle$.

The idea is that the bytecode activation records have corresponding *RBR* activation records, which preserve the order, and that any other RBR activation record

includes an empty set of instructions and will be used just to propagate the return value back to the caller site. Note that in the last point above, in addition, all activation records between any $a_{rr}^{i_k}$ and $a_{rr}^{i_{k+1}}$ correspond to rules in the same method, and they have a specific order which corresponds to a path in the corresponding CFG starting from the initial block. But this property is not really required for the proof and therefore we omit it. In what follows we refer to this additional equivalence condition as *structural equivalence of configurations.*

It is clear that this refined notion of equivalence implies the one of Definition 3.8, since all conditions mentioned in that definition are included in the new notion of equivalence. Now we proceed to prove Theorem 3.10 for the structural equivalence of configurations. The proof is by induction on the length n of the bytecode trace.

Base case. This case is for traces of length 0, i.e., when $n = 0$. Recall that the theorem defines the initial states as

—C_0=$\langle start, 1, lv_0, v_t \cdots v_1 \rangle; h_0$
—RC_0=$\langle start, m'(\bar{s}, s_1), lv'_0 \rangle; h_0$, where $s = \langle s_1, \ldots, s_t \rangle$ and $lv'_0(s_i) = v_i$ for all $1 \leq i \leq t$;

Moreover, the initial method call at $start[1]$ corresponds to the method from which the rule m' is obtained, namely $start[1] \equiv$ invokenonvirtual m'. Therefore, the traces are equivalent by definition.

Inductive case. Suppose that the theorem holds for bytecode traces of length $n-1$ where $n \geq 1$. We demonstrate that, then, it holds for traces of length n. Let $t \equiv C_0 \leadsto_{\mathsf{bc}}^{n-1} C_{n-1} \leadsto_{\mathsf{bc}} C_n$ be a bytecode trace of length n. By the induction hypothesis, there exists an equivalent RBR trace $RC_0 \leadsto_{\mathsf{rr}}^{k} RC_k$, where $k \geq n-1$ which is equivalent to $C_0 \leadsto_{\mathsf{bc}}^{n-1} C_{n-1}$. We show that it is possible to extend this RBR trace (by at least one transition) and obtain $RC_0 \leadsto_{\mathsf{rr}}^{k} RC_k \leadsto_{\mathsf{rr}}^{+} RC_{k'}$ such that $C_n \approx RC_{k'}$, and $RC_{k''}$ is redundant for any $k < k'' < k'$, which clearly implies the equivalence between $RC_0 \leadsto_{\mathsf{rr}}^{+} RC_{k'}$ and t. In what follows, we let $C_{n-1} \equiv \langle m, pc, lv_{n-1}, v_t \ldots v_1 \rangle \cdot ar; h_{n-1}$ and $RC_k \equiv \langle m_{id_1}, bc, lv'_k \rangle \cdot ar_{rr}; h^k_{rr}$. We also assume that m has f local variables and we denote them by the tuple $\bar{l} = \langle l_0, \ldots, l_{f-1} \rangle$. Our reasoning is done by cases depending on the instruction $m[pc]$.

Case 1: $m[pc] \equiv$ push v. In this case $m \neq start$, and according to the bytecode operational semantics of Figures 2 and 3, we have $C_n \equiv \langle m, pc+1, lv_n, v_{t+1} \cdot v_t \cdots v_1 \rangle \cdot ar; h_n$ where $v_{t+1} = v$, $lv_n = lv_{n-1}$ and $h_n = h_{n-1}$. Since $C_{n-1} \approx RC_k$, the sequence bc (in RC_k) must correspond to the compilation of the instructions in the block that corresponds to $m[pc]$, starting from the instruction that corresponds to $m[pc]$ in that block. Therefore, according to the compilation as defined in Section 3.2, bc must be of the form $b \cdot bc'$ where $b \equiv s_{t+1} := v$. If we start from RC_k and apply one execution step using the operational semantics of Figure 7, we get $RC_{k+1} \equiv \langle m_{id_1}, bc', lv'_{k+1} \rangle \cdot ar_{rr}; h^{k+1}_{rr}$ where $h^{k+1}_{rr} = h^k_{rr}$ and lv'_{k+1} agrees with lv'_k on all values (stack and local variables) except (maybe) for s_{t+1} which is $lv'_{k+1}(s_{t+1}) = v = v_{t+1}$. At this point we distinguish two cases:

(1) if $m[pc]$ is not the last instruction in the corresponding block, then bc' must correspond to the compilation of the instructions in that block starting from

the instruction that corresponds to $m[pc+1]$ (otherwise $C_{n-1} \not\approx RC_k$), and therefore RC_{k+1} is not redundant. Now, since $lv'_{k+1}(s_i) = v_i$ for all $1 \leq i \leq t+1$, and lv'_{k+1} and lv_n agree on the values of all local variables (because lv'_k does), then we can conclude that the top activation records of C_n and RC_{k+1} are equivalent. Also, it is clear that the structures of C_n and RC_{k+1} are "*identical*" to those of C_{n-1} and RC_k respectively, which means that C_n and RC_{k+1} have equivalent structures since C_{n-1} and RC_k do. Therefore, we can conclude that $C_n \approx RC_{k+1}$ which implies that $RC_0 \rightsquigarrow^k_{\mathsf{rr}} RC_k \rightsquigarrow_{\mathsf{rr}} RC_{k+1}$ is equivalent to $C_0 \rightsquigarrow^n_{\mathsf{rr}} C_n$, and therefore the theorem holds for this case.

(2) if $m[pc]$ is the last instruction in the corresponding block, then $m[pc+1]$ is the first instruction in the block m_{pc+1}. Since "$\mathsf{push}\ v$" is a sequential instruction, this case can occur only if the instruction at $pc+1$ has at least another predecessor different from pc, and therefore, according to the construction of the CFG in Section 3.2, m_{id_1} has a single out-edge that goes to m_{pc+1} with *true* guard. Then bc' must consist of the single (continuation) call $m_{pc+1}(\bar{l}, s_1, \ldots, s_{t+1}, out)$ and therefore RC_{k+1} is redundant. From the generation of the RBR the single rule for m_{pc+1} must have the form:

$$m_{pc+1}(\bar{l}, s_1, \ldots, s_{t+1}, out) \leftarrow true, body.$$

where *body* corresponds to the compilation of the instructions in block m_{pc+1} starting from the first instruction which corresponds to $m[pc+1]$. Now, applying one execution step to RC_{k+1} we get a non-redundant configuration $RC_{k+2} \equiv \langle m_{pc+1}, body, lv'_{k+2}\rangle \cdot \langle m_{id_1}[out, out], \epsilon, lv'_{k+1}\rangle \cdot ar_{rr}; h^{k+2}_{rr}$ such that $h^{k+2}_{rr} = h^{k+1}_{rr}$ and $lv'_{k+2}(s_i) = v_i$ for all $1 \leq i \leq t+1$ and $lv'_{k+2}(l_i) = lv'_{k+1}(l_i) = lv_n(i)$ for any local variable $0 \leq i < f$. Therefore, the top activation records of C_n and RC_{k+2} are equivalent. Also it is clear that the structures of C_n and RC_{k+2} are equivalent, which implies that $C_n \approx RC_{k+2}$, which in turn implies that the $RC_0 \rightsquigarrow^k_{\mathsf{rr}} RC_k \rightsquigarrow_{\mathsf{rr}} RC_{k+1} \rightsquigarrow_{\mathsf{rr}} RC_{k+2}$ is equivalent to $C_0 \rightsquigarrow^n_{\mathsf{rr}} C_n$, and therefore the theorem holds for this case as well.

For $m[pc] \in \{\mathsf{load}\ i, \mathsf{store}\ i, \mathsf{pop}, \mathsf{dup}, \mathsf{add}, \mathsf{sub}, \mathsf{getfield}\ f, \mathsf{aload}, \mathsf{arraylength}\}$, i.e., sequential instructions that do not modify the heap structure, the proof is identical to this case, except of course the justification of compiling $m[pc]$ which is straightforward. For $m[pc] \in \{\mathsf{new}\ c, \mathsf{putfield}\ f, \mathsf{newarray}\ d, \mathsf{astore}\}$, i.e., sequential instructions that might modify the heap, the proof is also similar, but since the heap h_n might be different from h_{n-1} we have to show that this change is reflected also when moving from h^k_{rr} to h^{k+1}_{rr} (resp. h^{k+2}_{rr}) , namely h^{k+1}_{rr} (resp. h^{k+2}_{rr}) is equivalent to h_n, and therefore the equivalence is maintained. This is done by showing that by definition (of the compilation) of the corresponding instructions, in bytecode and RBR, apply the same changes on the corresponding heaps. All these cases are skipped as the proofs are "*identical*".

Case 2: $m[pc] \equiv \mathsf{ifgt}\ pc'$. In this case $m \neq start$. Since "$\mathsf{ifgt}\ pc'$" is a branching instruction, then the corresponding block m_{id_1} has $\mathsf{nop}(\mathsf{ifgt}\ pc')$ as the last

instruction, and the corresponding RBR program has the following rules

$$m^c_{id_1}(\bar{l}, s_1, \ldots, s_t, out) \leftarrow s_{t-1} > s_t, m_{pc'}(\bar{l}, s_1, \ldots, s_{t-2}, out)$$
$$m^c_{id_1}(\bar{l}, s_1, \ldots, s_t, out) \leftarrow s_{t-1} \leq s_t, m_{pc+1}(\bar{l}, s_1, \ldots, s_{t-2}, out)$$
$$m_{pc'}(\bar{l}, s_1, \ldots, s_{t-2}, out) \leftarrow true, body_1$$
$$m_{pc+1}(\bar{l}, s_1, \ldots, s_{t-2}, out) \leftarrow true, body_2$$

Note that the possible consecutive instruction $m[pc']$ (resp. $m[pc+1]$) is the first instruction in the corresponding block $m_{pc'}$ (resp. m_{pc+1}), and moreover, $body_1$ (resp. $body_2$) corresponds to the compilation of the instructions in the block $m_{pc'}$ (resp. m_{pc+1}) starting from the first instruction which corresponds to $m[pc']$ (resp. $m[pc+1]$). Now, suppose that $s_{t-1} > s_t$ in C_{n-1}, then if we apply one execution step to C_{n-1} we obtain $C_n \equiv \langle m, pc', lv_n, v_{t-2} \cdots v_1 \rangle \cdot ar; h_n$ where $lv_n = lv_{n-1}$ and $h_n = h_{n-1}$. Since $C_{n-1} \approx RC_k$ then bc (in RC_k) must be of the form:

$$bc \equiv \mathsf{nop}(\mathsf{ifgt}\ pc'), m^c_{id_1}(\bar{l}, s_1, \ldots, s_t, out)$$

Applying three execution steps starting from RC_k we get

$$RC_{k+1} \equiv \langle m_{id_1}, m^c_{id_1}(\bar{l}, s_1, \ldots, s_t, out), lv'_{k+1} \rangle \cdot ar_{rr}; h^{k+1}_{rr}$$
$$RC_{k+2} \equiv \langle m^c_{id_1}, m_{pc'}(\bar{l}, s_1, \ldots, s_{t-2}, out), lv'_{k+2} \rangle \cdot \langle m_{id_1}[out, out], \epsilon, lv'_{k+1} \rangle \cdot ar_{rr}; h^{k+2}_{rr}$$
$$RC_{k+3} \equiv \langle m_{pc'}, body_1, lv'_{k+3} \rangle \cdot \langle m^c_{id_1}[out, out], \epsilon, lv'_{k+2} \rangle \cdot \langle m_{id_1}[out, out], \epsilon, lv'_{k+1} \rangle \cdot ar_{rr}; h^{k+3}_{rr}$$

where $lv'_{k+3}(l_i) = lv_n(i)$, for all local variable $0 \leq i < f$ of m and $lv'_{k+3}(s_i) = v_i$, for all $1 \leq i \leq t-2$. Therefore the top activation records of C_n and RC_{k+3} are equivalent. It is clear that RC_{k+1} and RC_{k+2} are redundant, and that $C_n \approx RC_{k+3}$ since $h^{k+3}_{rr} = h^{k+2}_{rr} = h^{k+1}_{rr} = h^k_{rr}$ and their structures are based on those of C_{n-1} and RC_k. Therefore the trace $RC_0 \rightsquigarrow^k_{\mathsf{rr}} RC_k \rightsquigarrow_{\mathsf{rr}} RC_{k+1} \rightsquigarrow_{\mathsf{rr}} RC_{k+2} \rightsquigarrow_{\mathsf{rr}} RC_{k+3}$ is equivalent to $C_0 \rightsquigarrow^n_{\mathsf{rr}} C_n$, and thus the theorem holds for this case. The other case, $s_{t-1} \leq s_t$, is similar and it uses the other continuation. Also, the cases for $m[pc] \in \{\mathsf{iflt}\ pc', \mathsf{ifeq}\ pc', \mathsf{ifnull}\ pc'\}$ are "*identical*" to "$\mathsf{ifgt}\ pc'$" and therefore we skip them.

Case 3: $m[pc] \equiv \mathsf{invokenonvirtual}\ \ m'$ where $m = start$. This can happen only in the initial state, therefore $pc = 1$ and $n = 1$. By definition $C_0 \approx RC_0$ and they are of the form

—$C_0 = \langle start, 1, lv_0, v_t \cdots v_1 \rangle; h_0$

—$RC_0 = \langle start, m'(\bar{s}, s_1), lv'_0 \rangle; h^0_{rr}$ where $s = \langle s_1, \ldots, s_t \rangle$ and $lv'_0(s_i) = v_i$ for all $1 \leq i \leq t$;

Moreover, the method m' has $t-1$ arguments (t if we count the *this* reference l_0), and we assume that it has $f' \geq t$ local variables. According to the compilation of the bytecode method m' (Section 3.2), the corresponding RBR must have the following two rules

$$m'(l_0, \ldots, l_{t-1}, out) \leftarrow true, m'_1(l_0, \ldots, l_{f'-1}, out)$$
$$m'_1(l_0, \ldots, l_{f'-1}, out) \leftarrow true, body_1$$

where $body_1$ corresponds to the compilation of block m'_1 starting from the first instruction $m'[1]$. Applying one execution step to C_0 we get $C_1 = \langle m', 1, lv_1, \epsilon \rangle \cdot \langle start, 2, lv_0, \epsilon \rangle; h_1$ where $h_1 = h_0$, $lv_1(i) = v_{i+1}$ for all $0 \leq i < t$, and $lv_1(i) = 0$

or $lv_1(i) = \mathsf{null}$ for all $t \leq i < f'$ (depending on the type). Applying two execution steps to RC_0 we get:

$$\begin{aligned} RC_1 = \ & \langle m', m'_1(l_0, \ldots, l_{f'-1}, out), lv'_1\rangle; \langle start[s_1, out], \epsilon, lv'_0\rangle; h^1_{rr} \\ RC_2 = \ & \langle m'_1, body_1, lv'_2\rangle \cdot \langle m'[out, out], \epsilon, lv'_1\rangle \cdot \langle start[s_1, out], \epsilon, lv'_0\rangle; h^2_{rr} \end{aligned}$$

where $lv'_2(l_i) = lv'_1(l_i) = lv_1(i) = v_{i+1}$ for all $0 \leq i < t$ and $lv'_2(l_i) = lv'_1(l_i) = lv_1(i) = 0$ or $lv'_2(l_i) = lv'_1(l_i) = lv_1(i) = \mathsf{null}$ (depending on the type) for all $t \leq i < f'$. Therefore, the top activation records of C_1 and RC_2 are equivalent, and therefore $C_1 \approx RC_2$ (since $h^2_{rr} = h^1_{rr} = h^0_{rr}$ and they have an equivalent structure). Moreover, the configuration RC_1 is redundant, which implies that $RC_0 \rightsquigarrow_{\mathsf{rr}} RC_1 \rightsquigarrow_{\mathsf{rr}} RC_2$ is equivalent to $C_0 \rightsquigarrow_{\mathsf{bc}} C_1$. Therefore the theorem holds for this case.

Case 4: $m[pc] \equiv \mathsf{invokevirtual}\ \ m''$. Suppose that m'' has f' arguments (excluding the *this* argument l_0), the class of the object whose method is called is d, namely $class(h_{n-1}(lv_{n-1}(v_{t-f'}))) = d$, and that resolving m'' starting from d results in m' (the actual method that we call) with f'' local variables (including l_0). Applying one execution step from C_{n-1}, we obtain the following bytecode configuration:

$$C_n \equiv \langle m', 1, lv_n, \epsilon\rangle \cdot \langle m, pc+1, lv_{n-1}, v_{t-f'-1} \cdots v_1\rangle \cdot ar; h_n$$

where $lv_n(i) = v_{t-f'+i}$ for all local variables $0 \leq i \leq f'$, and $lv_n(i) = 0$ or $lv_n(i) = \mathsf{null}$ for all local variables $f' < i < f''$ (depending on the type), and $h_n = h_{n-1}$. According to the compilation as described in Section 3.2, and since $C_{n-1} \approx RC_k$, it must be that $bc \equiv \mathsf{nop}(\mathsf{invokevirtual}\ \ m'') \cdot m^c_{pc}(\bar{l}, s_1, \ldots, s_t, out)$. Moreover, based on the sound class information that we assumed that is available, the RBR program must include the following rules:

$$\begin{aligned} & m^c_{pc}(\bar{l}, s_1, \ldots, s_t, out) \leftarrow \mathsf{type}(s_{t-f'}, d), m_{pc:d}(\bar{l}, s_1, \ldots, s_t, out) \\ & m_{pc:d}(\bar{l}, s_1, \ldots, s_t, out) \leftarrow true, m'(s_{t-f'}, \ldots, s_t, s_{t-f'}), m_{pc+1}(\bar{l}, s_1, \ldots, s_{t-f'}, out) \\ & m'(l_0, \ldots, l_{f'}, out) \leftarrow true, m'_1(l_0, \ldots, l_{f''-1}, out) \\ & m'_1(l_0, \ldots, l_{f''-1}, out) \leftarrow true, body_1 \end{aligned}$$

where $body_1$ corresponds to the compilation of the block m'_1, starting from the first instruction which corresponds to $m'[1]$. Applying five execution steps starting from RC_k we get:

$$\begin{aligned} RC_{k+1} = \ & \langle m_{id_1}, m^c_{pc}(\bar{l}, s_1, \ldots, s_t, out), lv'_{k+1}\rangle \cdot ar_{rr}; h^{k+1}_{rr} \\ RC_{k+2} = \ & \langle m^c_{pc}, m_{pc:d}(\bar{l}, s_1, \ldots, s_t, out), lv'_{k+2}\rangle \cdot \langle m_{id_1}[out, out], \epsilon, lv'_{k+1}\rangle \cdot ar_{rr}; h^{k+2}_{rr} \\ RC_{k+3} = \ & \langle m_{pc:d}, m'(s_{t-f'}, \ldots, s_t, s_{t-f'}) \cdot m_{pc+1}(\bar{l}, s_1, \ldots, s_{t-f'}, out), lv'_{k+3}\rangle \cdot \\ & \langle m^c_{pc}[out, out], \epsilon, lv'_{k+2}\rangle \cdot \\ & \langle m_{id_1}[out, out], \epsilon, lv'_{k+1}\rangle \cdot ar_{rr}; h^{k+3}_{rr} \\ RC_{k+4} = \ & \langle m', m'_1(l_0, \ldots, l_{f''-1}, out), lv'_{k+4}\rangle \cdot \\ & \langle m_{pc:d}[s_{t-f'}, out], m_{pc+1}(\bar{l}, s_1, \ldots, s_{t-f'}, out), lv'_{k+3}\rangle \cdot \\ & \langle m^c_{pc}[out, out], \epsilon, lv'_{k+2}\rangle \cdot \\ & \langle m_{id_1}[out, out], \epsilon, lv'_{k+1}\rangle \cdot ar_{rr}; h^{k+4}_{rr} \\ RC_{k+5} = \ & \langle m'_1, body_1, lv'_{k+5}\rangle \cdot \\ & \langle m'[out, out], \epsilon, lv'_{k+4}\rangle \cdot \\ & \langle m_{pc:d}[s_{t-f'}, out], m_{pc+1}(\bar{l}, s_1, \ldots, s_{t-f'}, out), lv'_{k+3}\rangle \cdot \\ & \langle m^c_{pc}[out, out], \epsilon, lv'_{k+2}\rangle \cdot \\ & \langle m_{id_1}[out, out], \epsilon, lv'_{k+1}\rangle \cdot ar_{rr}; h^{k+5}_{rr} \end{aligned}$$

Note that RC_{k+1}, RC_{k+2}, RC_{k+3}, and RC_{k+4} are redundant. Therefore, in order to prove that $RC_0 \rightsquigarrow_{\mathsf{rr}}^{k+5} RC_{k+5}$ is equivalent to $C_0 \rightsquigarrow_{\mathsf{rr}}^{n} C_n$, we need to show that $C_n \approx RC_{k+5}$, which holds due to the following points that can be verified using the corresponding semantics rules:

—$lv'_{k+5}(i) = v_{t-f'+i}$ for all local variables $0 \leq i \leq f'$, and $lv'_{k+5}(i) = 0$ or $lv'_{k+5}(i) = \mathsf{null}$ for all local variables $f' < i < f''$ (depending on the type). Therefore, lv'_{k+5} agrees with lv_n on the values of all local variables, which together with that fact that $body_1$ corresponds to the compilation of the block m'_1 starting from $m'[1]$, implies that the top activation records of RC_{k+5} and C_n are equivalent.

—$lv'_{k+3}(i) = lv_{n-1}(i)$ for all variables $0 \leq i < f$, and $lv'_{k+3}(s_i) = v_i$ for all $1 \leq i \leq t - f' - 1$ (since up to that point local and stack variables are just carried around in m). This means that, in RC_{k+5}, the corresponding activation record (the one that has lv'_{k+3}) agrees on the local variables and stack values with $\langle m, pc + 1, lv_{n-1}, v_{t-f'-1} \cdots v_1 \rangle$ in C_n. In addition, it is clear that C_n and RC_{k+5} are structurally equivalent, given that C_{n-1} and RC_k are structurally equivalent.

The proof for $m[pc] \equiv \mathsf{invokenonvirtual}\ \ m''$, with $m \neq start$ is identical to this case, so we skip it.

Case 5: $m[pc] \equiv \mathsf{return}$. This case is the one for which we have needed to refine the notion of equivalence, mainly in order to show that when returning from a method we can return to an equivalent configuration. The configurations C_{n-1} and C_n must have the form:

$$\begin{aligned} \mathrm{C}_{n-1} &\equiv \langle m, pc, lv_{n-1}, v_t \cdots v_1 \rangle \cdot \langle m', pc', lv, v'_{t'} \ldots v'_1 \rangle \cdot ar'; h_{n-1} \\ \mathrm{C}_n &\equiv \langle m', pc', lv, v'_{t'+1} \cdot v'_{t'} \ldots v'_1 \rangle \cdot ar'; h_n \end{aligned}$$

where $v_{t'+1} = v_t$ (the return value), and $h_n = h_{n-1}$. Moreover, since $C_{n-1} \approx RC_k$, then the following must hold (by the induction hypothesis):

—$RC_k \equiv \langle m_{id_1}, out := s_t, lv'_k \rangle \cdot ar_{rr}; h^k_{rr}$;

—There exists an activation record in ar_{rr} of the form

$$\langle m'_{pc'-1:d}[s_{t'+1}, out], m'_{pc'}(\bar{l}', s_1, \ldots, s_{t'+1}, out), lv' \rangle$$

where $\bar{l}' = \langle l_0, \ldots, l_{f'-1} \rangle$ is the list of all local variables in m' (assuming m' has f' local variables), such that $lv'(l_i) = lv(i)$ for all local variables $0 \leq i < f'$, and $lv'(s_i) = v'_i$ for all $1 \leq i \leq t'$;

—All activation records between the above one and the top activation record are of the form $\langle p[out, out], \epsilon, \_ \rangle$. Let us assume that there are $k' \geq 0$ such activation records.

The above points necessarily imply that we can apply $k'+2$ execution steps starting from RC_k, one for the assignment $out := v_t$ and $k' + 1$ using rule $(11)_{rr}$, and get into the following configuration

$$RC_{k+k'+2} \equiv \langle m_{pc'-1:d}[s_{t'+1}, out], m'_{pc'}(\bar{l}', s_1, \ldots, s_{t'+1}, out), lv'' \rangle \cdot ar_{rr}; h^{k+k'+2}_{rr}$$

where lv'' agrees with all values with lv' and in addition $lv''(s_{t-f'}) = v_t$, and therefore agrees on all values with lv in C_n. Applying another step we get:

$$RC_{k+k'+3} \equiv \langle m'_{pc'-1:d}[s_{t'+1}, out], body_1, lv''' \rangle \cdot ar_{rr}; h^{k+k'+3}_{rr}$$

where lv''' is identical to lv'', and $body_1$ corresponds to the compilation of the block $m'_{pc'}$ which starts with the instruction $m'[pc']$, and therefore the top activation records of C_n and $RC_{k+k'+3}$ are equivalent. Moreover it is easy to see that C_n and $RC_{k+k'+3}$ have an equivalent structure since it is inherited from C_{n-1} and RC_k, and also all RC_i such that $k < i < k + k' + 3$ are redundant which implies that $RC_0 \rightsquigarrow_{\mathsf{rr}}^{k+k'+3} RC_{k+k'+3}$ is equivalent to $C_0 \rightsquigarrow_{\mathsf{bc}}^{n} C_n$.

B. CORRECTNESS OF COST ANALYSIS

B.1 Proof of Lemma 5.4

The idea of the proof is to use the renamings that are used during the generation of the abstract rules, in order to construct the mapping f which relates program variables to their abstract ones. In order to do this, we attach to the abstract rule the renamings (ρ) computed in Definition 5.2. Therefore, the abstract rules are now of the form:

$$p(\overline{x}, y') \leftarrow \varphi_0 \mid b_1^\alpha, \ldots, b_n^\alpha \circ \langle \rho_1, \ldots, \rho_{n+1} \rangle$$

where $\langle \rho_1, \ldots, \rho_{n+1} \rangle$ is the tuple of all renamings that were used during the abstract compilation of that specific rule. In addition, we modify the abstract transition system that is defined in Section 5.2, in order carry around all corresponding renamings

$$\frac{p(\bar{x}, y) \leftarrow \varphi \mid b_1^\alpha, \ldots, b_n^\alpha \circ \langle \rho_1, \ldots, \rho_{n+1} \rangle \ll_{AC} P^\alpha, \quad \psi \wedge \varphi \not\models \mathit{false}}{AC = \langle \langle p(\bar{x}, y), \phi \rangle \cdot bc^\alpha, \psi, \rho \cdot \bar{\rho} \rangle \rightsquigarrow_\alpha \langle b_1^\alpha \cdots b_n^\alpha \cdot \phi \cdot bc^\alpha, \psi \wedge \varphi, \rho_1 \cdots \rho_{n+1} \cdot \bar{\rho} \rangle}$$

$$\frac{\psi \wedge \varphi \not\models \mathit{false}}{\langle \varphi \cdot bc^\alpha, \psi, \rho \cdot \bar{\rho} \rangle \rightsquigarrow_\alpha \langle bc^\alpha, \psi \wedge \varphi, \bar{\rho} \rangle}$$

Clearly, this does not affect the abstract execution since the renamings are only carried around. The reason for collecting them is just to make the construction of the mapping f simpler. Note that when selecting a renamed apart abstract rule, we assume also that the constraints variables in the *range* of each ρ_i are also renamed (exactly as those in the body). Now abstract configurations are of the form $\langle p, bc^\alpha, \varphi, \bar{\rho} \rangle$ where $\bar{\rho}$ is a stack of renamings.

For proving the lemma, we need a notion of structural equivalence in order to claim that two traces correspond to the same execution. Therefore, in addition to the requirements in Lemma 5.4, we claim that for each configuration $RC \equiv a_k \cdots a_0; h$ occurring at step number l in the $\rightsquigarrow_{\mathsf{rr}}$-trace, it holds:

(1) $a_i = \langle p_i[w', w], bc_i, lv_i \rangle$ for $0 \leq i < k$; and
(2) $a_k = \langle p_k, bc_k, lv_k \rangle$.
(3) the corresponding abstract configuration (step l of $\rightsquigarrow_\alpha$-trace) has the form

$$AC \equiv \langle bc_k^\alpha \cdot \phi_k \cdot bc_{k-1}^\alpha \cdots \phi_1 \cdot bc_0^\alpha, \varphi, \bar{\rho}_k \cdots \bar{\rho}_0 \rangle$$

where for all $0 \leq i \leq k$

(a) $bc_i \equiv b_{i:1} \cdots b_{i:k_i}$
(b) $bc_i^\alpha \equiv b_{i:1}^\alpha \cdots b_{i:k_i}^\alpha$;
(c) $\bar{\rho}_i = \rho_{i:1} \cdots \rho_{i:(k_i+1)}$;

(d) for all $1 \leq j \leq k_i$, $b^{\alpha}_{i:j}$ is the abstract compilation of $b_{i:j}$ with respect to $\rho_{i:j}$ which generates the new renaming $\rho_{i:(j+1)}$ where $range(\rho_{i:(j+1)}) \setminus range(\rho_{i:j}) \not\subseteq vars(\varphi)$ are fresh variables that do not appear before (i.e., in the renaming to the right of $\rho_{i:(j+1)}$);
(e) If $i < k$ then $\rho_{(i+1):(k_{i+1}+1)}(w') = \rho_{i:1}(w)$;

Note that the above requirements are added to those of Lemma 5.4, and therefore we get a stronger lemma which implies Lemma 5.4. Similarly to the proof of Theorem 3.10, the need for these requirements stems from the fact that we need to state that corresponding concrete and abstract configuration are obtained by executing the same instruction (in the concrete and abstract way), and also that they will execute the same instruction (in the concrete and abstract way) in future steps. We prefer to keep the additional requirements only in the proof in order to simplify the presentation in the paper. Now we proceed with the proof by induction on the length n of the concrete trace.

We start by explaining how to construct the mapping f at each step: given $RC_i = \langle \_, \_, lv_i \rangle$ and its corresponding $AC_i = \langle \_, \_, \rho_i \cdot \bar{\rho} \rangle$, we define f for the i-th step variables as $f(z, i) = \rho_i(z)$, for all $z \in dom(lv_i)$. For a trace of n steps, the function is defined as the union of all mapping for the configurations.

Base Case. If the trace is of length 0, i.e., $(n = 0)$, then:

$$RC_0 \equiv \langle start, p(\bar{x}, y), lv_0 \rangle; h_0$$

Now we define:

$$AC_0 \equiv \langle \langle p(\bar{x}, y'), \phi_0 \rangle, \varphi_0, id \cdot \rho_0 \rangle$$

where $\langle p(\bar{x}, y'), \phi_0 \rangle$ is the abstract compilation of $p(\bar{x}, y)$ with respect to the identity renaming id, which generates ρ_0 as the resulting renaming; and

$$\varphi_0 = \bigwedge_{z \in \bar{x} \cup \{y\}} id(z) = \alpha(z, \mathsf{static\_type}(z), RC_0)$$

Now we define σ as $\sigma(id(z)) = f(z, 0) = \alpha(z, \mathsf{static\_type}(z), RC_0)$, for all $z \in \bar{x} \cup \{y\}$. Clearly $\sigma \models \varphi_0$, and moreover the structural equivalence conditions hold for these configurations. Therefore, the lemma holds for the base case.

Inductive case. Now we consider traces of length $n + 1 > 0$. Assuming that the lemma holds for all $\leadsto_{\mathsf{rr}}$-traces of length $n \geq 0$ (the induction hypothesis), we show that it also holds for traces that consist of $n+1$ steps. Consider a $\leadsto_{\mathsf{rr}}$-trace of length n:

$$\begin{array}{lll} RC_0 & \equiv & \langle start, \overbrace{p(\bar{x}, y)}^{b}, lv_0 \rangle; h_0 \leadsto^n_{\mathsf{rr}} \\ RC_n & \equiv & \langle h, bc_n, lv_n \rangle \cdot ar_n; h_n \end{array}$$

By the induction hypothesis, there exists an $\leadsto_{\alpha}$-trace of the form:

$$\begin{array}{lll} AC_0 & \equiv & \langle b^{\alpha}, \varphi_0, \rho_0 \rangle; \qquad \leadsto^n_{\alpha} \\ AC_n & \equiv & \langle bc^{\alpha}_n \cdot bc^{\alpha}, \varphi_n, \bar{\rho}_n \rangle \end{array}$$

such that the conditions of the lemma are satisfied. Let us analyze how the lemma extends to all possible $\leadsto_{\mathsf{rr}}$-traces of length $n{+}1$ generated from the above concrete and abstract traces. We reason for all possibles cases of Figure 7:

—Rule $(1)_{rr}$. In this case

$$\begin{array}{lll} RC_n & \equiv & \langle q, z := exp \cdot bc_{n+1}, lv_n\rangle \cdot ar_n; h_n \leadsto_{\mathsf{rr}} \\ RC_{n+1} & \equiv & \langle q, bc_{n+1}, lv_{n+1}\rangle \cdot ar_n; h_n \end{array}$$

where $lv_{n+1} = lv_n[z \mapsto v]$ and $v = eval(exp, lv_n)$. By the induction hypothesis, $AC_n = \langle w = exp^\alpha \cdot bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n, \rho_n \cdot \rho_{n+1} \cdot \bar{\rho}\rangle$ and there exists a valuation σ and mapping f satisfying the conditions of the lemma. Applying one execution step we get

$$AC_{n+1} = \langle bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_{n+1}, \rho_{n+1} \cdot \bar{\rho}\rangle$$

where φ_{n+1} is $w = exp^\alpha \wedge \varphi_n$. It holds, by the induction hypothesis, that $w = exp^\alpha$ is the abstract compilation of $z := exp$ w.r.t. ρ_n, which generates the new renaming ρ_{n+1}. Hence $\rho_{n+1}(z) = w$. Also by the induction hypothesis, w is a fresh variable which does not occur in φ_n and $dom(\sigma)$. Then, let us extend σ such that:

$$\sigma(\rho_{n+1}(z)) = \alpha(z, \mathsf{static\_type}, RC_{n+1})$$

Since $\sigma \models \varphi_n$, we have to prove only that $\sigma \models w = exp^\alpha$ in order to get $\sigma \models \varphi_{n+1}$. We distinguish several cases:

—exp is a numeric expression and hence all variables involved in exp are of type integer. By definition of σ it holds $\sigma(\rho_{n+1}(z)) = \alpha(z, \mathsf{static\_type}(z), RC_{n+1}) = lv_{n+1}(z) = v$. Hence $\sigma(w) = v$. On the other hand, by applying the induction hypothesis together with the definition of abstract compilation, exp^α must evaluate to v in σ since exp^α is obtained from exp by changing each program variable by its corresponding abstract one. Hence $\sigma \models w = exp^\alpha$.

—exp is not numeric. Then it has the form $z = \mathsf{null}$ or $z = z'$ where z and z' are either references or arrays. For the first case, by the definition of abstract compilation, it holds that $exp^\alpha \equiv 0$ and also $\sigma(\rho_{n+1}(z)) = \mathsf{path\text{-}length}(lv_{n+1}(z), h_n) = \mathsf{path\text{-}length}(eval(\mathsf{null}, lv_n), h_n) = 0$. Therefore $\sigma \models w = 0$. Suppose now that $exp \equiv z = z'$, where z and z' are references. Then $exp^\alpha \equiv \rho_{n+1}(z) = \rho_n(z')$. But $\sigma(\rho_{n+1}(z)) = \mathsf{path\text{-}length}(lv_{n+1}(z), h_n) = \mathsf{path\text{-}length}(lv_{n+1}(z'), h_n) = \sigma(\rho_n(z'))$, and therefore $\sigma \models w = \rho_n(z')$. For the case of arrays we can reason similarly.

It is clear that the mapping f as defined at the beginning of the proof satisfies the conditions of the lemma, and moreover this step does not affect the structural equivalence and therefore the lemma holds.

—Rule $(2)_{rr}$. In this case:

$$\begin{array}{lll} RC_n & \equiv & \langle q, z := \mathsf{new}\ c \cdot bc_{n+1}, lv_n\rangle \cdot ar_n; h_n \leadsto_{\mathsf{rr}} \\ RC_{n+1} & \equiv & \langle q, bc_{n+1}, lv_{n+1}\rangle \cdot ar_n; h_n[r \mapsto o] \end{array}$$

where $lv_{n+1} = lv_n[z \mapsto r]$, $o = newobject(c)$ and $r \notin dom(h_n)$. By the induction hypothesis, we can build a $\leadsto_\alpha$-trace which finishes in the following abstract configuration $AC_n \equiv \langle w = 1 \cdot bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n, \rho_n \cdot \rho_{n+1} \cdot \bar{\rho}\rangle$, for which all conditions in

the lemma holds. Concretely $\rho_{n+1}(z)$ is a fresh variable which does not occur in φ_n and hence we can extend σ so that $\sigma(\rho_{n+1}(z)) = 1$, i.e. $\sigma(w) = 1$. With such a σ we can execute the following step:

$$AC_n \rightsquigarrow_\alpha \langle bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n \wedge w = 1, \rho_{n+1} \cdot \bar{\rho}\rangle$$

Also, $\sigma(\rho_{n+1}(z)) = \alpha(z, \mathsf{static\_type}(z), RC_{n+1})$ since z points to a new object and therefore $\alpha(z, \mathsf{static\_type}(z), RC_{n+1}) = \mathsf{path\text{-}length}(lv_{n+1}(z), h_n[r \mapsto o]) = 1$. It is clear that the mapping f as defined at the beginning of the proof satisfies the conditions of the lemma, and moreover this step does not affect the structural equivalence and therefore the lemma holds.

—For rules $(3)_{rr}$ and $(4)_{rr}$, the reasoning is similar to the above but in addition it is based on the correctness of path-length (Theorem 5.12 in Spoto et al. [2006b]). Moreover, these instructions do not affect the structural equivalence, and therefore the lemma holds.

—For rules $(5)_{rr}$ and $(6)_{rr}$, $(7)_{rr}$, it is clear that the instructions do not change the structural equivalence defined at the beginning of this proof. For $(5)_{rr}$, the same σ still satisfies the conditions of the lemma for the $n+1$ since the abstract values do no change (updating an array does not change its length), and moreover no new abstract variables are introduced when compiling this instruction. For $(6)_{rr}$, since an array is abstracted to its length, when accessing an array element all we can say is that its (path-length) size is non-negative if it is of reference type (which is clearly an approximation of the concrete one), otherwise no information is obtained ($true$). Extending σ such that $\sigma(\rho_{n+1}(x)) = \alpha(x, \mathsf{static\_type}(x), RC_{n+1})$ clearly satisfies φ_{n+1}.

—For rules $(7)_{rr}$ and $(8)_{rr}$ the proof is similar, we explain the one for rule $(7)_{rr}$. We have that

$$\begin{array}{lll} RC_n & \equiv & \langle q, z := \mathsf{arraylength}(w) \cdot bc_{n+1}, lv_n\rangle \cdot ar_n; h_n \rightsquigarrow_{\mathsf{rr}} \\ RC_{n+1} & \equiv & \langle q, bc_{n+1}, lv_{n+1}\rangle \cdot ar_n; h_n[r \mapsto o] \end{array}$$

where $lv_{n+1} = lv_n[z \mapsto o.length]$, $o = h(lv_n(w))$ and $lv_n(w) \neq \mathsf{null}$. By the induction hypothesis, we can build a $\rightsquigarrow_\alpha$-trace which finishes in the following abstract configuration

$$AC_n = \langle z' = w' \wedge z' \geq 0 \cdot bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n, \rho_n \cdot \rho_{n+1} \cdot \bar{\rho}\rangle$$

and satisfies the conditions of the lemma. Since $z' = w \wedge z' \geq 0$ is the abstract compilation of $z := \mathsf{arraylength}(w)$ with respect to ρ_n which generates as new renaming ρ_{n+1}, then it holds that $\rho_n(w) = w'$ and $\rho_{n+1}(z) = z'$. By the induction hypothesis z' does not occur in φ_n, then we can extend σ so that

$$\sigma(z') = \sigma(\rho_{n+1}(z)) = \alpha(z, \mathsf{static\_type}(z), RC_{n+1})$$

We have to prove that $\sigma(z') = \sigma(w')$ and $\sigma(z') \geq 0$. But $\sigma(w') = \sigma(\rho_n(w))$. And by the induction hypothesis

$$\sigma(w') = \sigma(\rho_n(w)) = \alpha(w, \mathsf{static\_type}(w), RC_n) = \mathsf{array\text{-}length}(lv_n(w), h_n) = o.length \geq 0$$

On the other hand $\sigma(z') = \alpha(z, \mathsf{static\_type}(z), RC_{n+1}) = lv_{n+1}(z)$. But $lv_{n+1}(z) = o.length$. Hence $\sigma(z') = \sigma(w')$ and $\sigma(z') \geq 0$.

—For rules $(9)_{rr}$, since we abstract the instruction to *true* and the corresponding instruction has no effect on the state, then the lemma holds trivially by taking the valuation σ coming from the induction hypothesis.

—Rule $(10)_{rr}$. Then

$$\begin{array}{lll} RC_n & \equiv & \langle h, q(\bar{z}, w) \cdot bc_n, lv_n\rangle \cdot ar_n; h_n \\ RC_{n+1} & \equiv & \langle q, bc_{n+1}, lv_{n+1}\rangle \cdot \langle h[w', w], bc_n, lv_n\rangle \cdot ar_n; h_n \end{array} \quad \leadsto_{\mathsf{rr}}$$

where $r{\equiv}q(\bar{z}', w') \leftarrow g', bc'{\in}P_{rr}$, $bc' = bc_{n+1}$, $lv_{n+1} = newenv(q)$, $lv_{n+1}(\bar{z}') = lv_n(\bar{z})$, $eval(g', lv_{n+1}) = true$. By the induction hypothesis, we can build an abstract derivation $AC_0 \rightsquigarrow_\alpha AC_n$ satisfying the conditions of the lemma, and hence:

$$AC_n{\equiv}\langle\langle q(\bar{a}, b), \phi_n\rangle \cdot bc_n^\alpha \cdot bc^\alpha, \varphi_n, \rho_n \cdot \rho_{n+1} \cdot \bar{\rho}\rangle$$

where $\langle q(\bar{a}, b), \phi_n\rangle$ is the abstract compilation of $q(\bar{z}, w)$ w.r.t. ρ_n which generates the new renaming ρ_{n+1}. Hence, according to the rules in Figure 8, it holds that

$$\begin{array}{ll} \rho_n(\bar{z}) = \bar{a} & (1) \\ \rho_{n+1}(w) = b & \end{array}$$

On the other hand, also by the induction hypothesis, there exists a valuation σ verifying the conditions of the lemma, i.e., for all $c{\in}dom(lv_n)$, $\sigma(\rho_n(c)) = \alpha(c, \mathsf{static\_type}(c), RC_n)$ and $\sigma \models \varphi_n$.

Let us take $r^\alpha{\equiv}q(\bar{a}, b) \leftarrow \varphi \wedge g'^\alpha \mid bc'^\alpha \circ \rho^q_{first} \cdot \bar{\rho}^q \cdot \rho^q_{last} \ll_{AC_n} P^\alpha$. Then, it holds by construction that:

$$\begin{array}{ll} \varphi = \{x = 0 \mid x{\in}var(r^\alpha)\backslash\bar{a}\} & \\ \rho^q_{first}(\bar{z}') = \bar{a} & (2) \\ \rho^q_{last}(w') = b & \end{array}$$

i.e, $\rho^q_{last}(w') = b = \rho_{n+1}(w)$. By definition of r^α, it holds that all variables in r^α different from $\bar{a}$ are fresh variables, i.e, they do not appear in φ_n. Hence, we can extend σ as follows:

$$\sigma(\rho^q_{first}(c)) = \alpha(c, \mathsf{static\_type}(c), RC_{n+1})$$

for all $c{\in}var(r)\backslash\bar{z}'$ and it holds that $\sigma \models \varphi_n$. Let us prove now that for all $z'_i{\in}\bar{z}'$ it holds that $\sigma(\rho^q_{first}(z'_i)) = \alpha(z'_i, \mathsf{static\_type}(c), RC_{n+1})$. But:

$$\begin{array}{lcll} \alpha(z'_i, \mathsf{static\_type}(z'_i), RC_{n+1}) & = & \% & lv_{n+1}(z'_i) = lv_n(z_i) \\ & & \% & \text{and the heap does not change} \\ \alpha(z_i, \mathsf{static\_type}(z_i), RC_n) & = & \% & \text{induction hypothesis} \\ \sigma(\rho_n(z_i)) & = & \% & \text{by (1)} \\ \sigma(a_i) & = & \% & \text{by (2)} \\ \sigma(\rho_{first}(z'_i)) & & & \end{array}$$

Hence we have proven that for all $c \in dom(lv_{n+1})$ it holds that $\sigma(\rho^q_{first}(c)) = \alpha(c, \mathsf{static\_type}(c), RC_{n+1})$.

In order to give the corresponding $\rightsquigarrow_\alpha$-step, we have to prove that $\sigma \models \varphi \wedge g'^\alpha$:

(1) $[\sigma \models \varphi]$ Let x be any variable in φ. By construction of r^α we know that there exists a variable $c \in var(r) \backslash \bar{z}'$ such that $\rho^q_{first}(c) = x$. By definition of σ it holds that $\sigma(\rho^q_{first}(c)) = \alpha(c, \mathsf{static\_type}(c), RC_{n+1})$. But $\alpha(c, \mathsf{static\_type}(c), RC_{n+1})$ depends on the value of $lv_{n+1}(c)$ together with h_n. But since $c \notin \bar{z}'$, then $newenv(q)$ guarantees that $lv_{n+1}(c)$ is either equal to 0 or null depending on the type of c. For both cases the corresponding abstraction carried out by α is 0 and hence $\sigma \models x = 0$, i.e., $\sigma \models \varphi$.

(2) $[\sigma \models g'^\alpha]$. We distinguish two cases:

—*g' is a numeric guard*, i.e., all its variables are of type integer. Let us consider any variable c in g'. Then $\rho^q_{first}(c) \in g'^\alpha$. By definition, $\sigma(\rho_{first}(c)) = \alpha(c, \mathsf{static\_type}(c), RC_{n+1})$. But $\alpha(c, \mathsf{static\_type}(c), RC_{n+1}) = eval(c, lv_{n+1}) = lv_{n+1}\ (c)$. Now, since $eval(g', lv_{n+1}) = true$ then $\sigma \models g'^\alpha$.

—g' contains variables whose static type is either an array or a reference. Then $g' \equiv c = \mathsf{null}$ or $g' \equiv c = d$. For the first case $g'^\alpha \equiv \rho_{first}(c) = 0$. By definition of σ, it holds that $\sigma(\rho_{first}(c)) = 0$. Then $\sigma \models c = \mathsf{null}$.
Let us consider guards g' of the form $c = d$, where c and d are references. We have that $g'^\alpha \equiv \rho_{first}(c) = \rho_{first}(d)$.
Then, by definition of σ, $\sigma(\rho_{first}(c)) = \mathsf{path\text{-}length}(lv_{n+1}(c), h_n)$. But since $eval(g', lv_{n+1}) = true$, then

$$\mathsf{path\text{-}length}(lv_{n+1}(c), h_n) = \mathsf{path\text{-}length}(lv_{n+1}(d), h_n)$$

Hence $\sigma(\rho_{first}(c)) = \sigma(\rho_{first}(d))$ and the result holds. For the case of arrays we can follow the same reasoning.

Note that the resulting configuration at step $n+1$ still satisfies the structural equivalence as specified at the beginning of the proof. This holds since in step $n+1$ we add to the concrete trace a sequence of bytecode and to the abstract one their corresponding abstract formula and the renamings that were used to generate them.

—Rule $(11)_{rr}$. Then:

$$\begin{array}{lcl} RC_n & \equiv & \langle q, \epsilon, lv_n\rangle \cdot \langle h[w', w], bc_{n+1}, lv_{n+1}\rangle \cdot ar_n; h_n \rightsquigarrow_{\mathsf{rr}} \\ RC_{n+1} & \equiv & \langle h, bc_{n+1}, lv_{n+1}[w \mapsto lv_n(w')]\rangle \cdot ar_n; h_n \end{array}$$

By the induction hypothesis it holds that we can build a $\rightsquigarrow_\alpha$-trace verifying the conditions of the lemma and such that:

$$AC_n \equiv \langle \phi_n \cdot bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n, \rho^q_{last} \cdot \rho_{n+1} \cdot \bar{\rho}\rangle$$

Furthermore, it holds, by the induction hypothesis, that $\rho^q_{last}(w') = \rho_{n+1}(w)$ and that there exists a valuation σ defined as $\sigma(\rho_{n+1}(c)) = \alpha(c, \mathsf{static\_type}(c), RC_n)$, for all $c \in dom(lv_n)$ such that $\sigma \models \varphi_n$. Then we have that:

$$(*)\begin{cases} \sigma(\rho_{n+1}(w)) & = \ \% \text{ By induction hypothesis} \\ \sigma(\rho^q_{last}(w')) & = \ \% \text{ By induction hypothesis} \\ \alpha(w', \mathsf{static\_type}(w'), RC_n) & = \ \% \ lv_{n+1}(w) = lv_n(w') \\ & \quad \% \text{ and the heaps are identical} \\ \alpha(w, \mathsf{static\_type}(w), RC_{n+1}) & \end{cases}$$

Let us consider the last activation record RC_k, $k<n$, in which a call to q was the first instruction to be processed.

$$\begin{array}{lll} RC_k & \equiv & \langle h, q(\bar{z}, w) \cdot bc_k, lv_k\rangle \cdot ar_k; h_k \\ RC_{k+1} & \equiv & \langle q, bc', lv_{k+1}\rangle \cdot \langle h[w', w], bc_k, lv_k\rangle \cdot ar_k; h_k \end{array}$$

where $r{\equiv}q(\bar{z}', w') \leftarrow g', bc' \in P_{rr}$, $lv_{k+1}(\bar{z}') = lv_k(\bar{z})$, $lv_{k+1} = newenv(q)$. Note that $dom(lv_{n+1}) = dom(lv_k)$. We have then:

$$RC_0 \rightsquigarrow^k_{\mathsf{rr}} RC_k \rightsquigarrow_{\mathsf{rr}} RC_{k+1} \rightsquigarrow^*_{\mathsf{rr}} RC_n \rightsquigarrow_{\mathsf{rr}} RC_{n+1}$$

By the induction hypothesis, we can build a $\rightsquigarrow_\alpha$-trace of the form:

$$AC_0 \rightsquigarrow^k_\alpha AC_k \rightsquigarrow_\alpha AC_{k+1} \rightsquigarrow^*_\alpha AC_n$$

which satisfies the conditions of the lemma. Concretely:

$$\begin{array}{l} AC_k{\equiv}\langle\langle q(\bar{a}, b), \phi_k\rangle \cdot bc^\alpha_k \cdot \Box, \varphi_k, \rho_k \cdot \rho_{k+1} \cdot \bar{\rho}_k\rangle \\ AC_{k+1}{\equiv}\langle {bc'}^\alpha \cdot \phi_k \cdot bc^\alpha_k \cdot \Box, \varphi_{k+1}, \rho^q_{first} \cdot \bar{\rho}_{k+1}\rangle \\ AC_n{\equiv}\langle \phi_k \cdot bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n, \rho^q_{last} \cdot \rho_{k+1} \cdot \bar{\rho}\rangle \end{array}$$

where $\phi_k{\equiv}\phi_n$ and $\langle q(\bar{a}, b), \phi_k\rangle$ is the abstract compilation of $q(\bar{z}, w)$ w.r.t. ρ_k which generates as new renaming ρ_{k+1} and $\rho_{k+1}{\equiv}\rho_{n+1}$. Furthermore, $\rho_{k+1}(\bar{z}) = a$.

Let us distinguish two cases:

—$\phi_n{\equiv}true$. Then by using σ we can give the following $\rightsquigarrow_\alpha$-step and compute:

$$AC_{n+1}{\equiv}\langle bc^\alpha_{n+1} \cdot bc^\alpha, \varphi_n, \rho_{n+1} \cdot \bar{\rho}\rangle$$

Let us prove now that for all $c{\in}dom(lv_{n+1})$ it holds that $\sigma(\rho_{n+1}(c)) = \alpha(c, \mathsf{static\_type}(c), RC_{n+1})$. To this end, let us consider all possible variables in such a domain:

—If c is different from $\bar{z}$ and w, then the result holds trivially since such variables are not modified by the execution of q and any modification on the heap done by q does not affect them. Note that this holds since $lv_{k+1} = newenv(q)$. Then:

$$\begin{array}{ll} \sigma(\rho_{k+1}(c)) & = \ \% \text{ by definition} \\ \sigma(\rho_k(c)) & = \ \% \text{ induction hypothesis} \\ \alpha(c, \mathsf{static\_type}(c), RC_k) & = \ \% \text{ not affected by the execution of } q \\ \alpha(c, \mathsf{static\_type}(c), RC_{n+1}) & \end{array}$$

—If $c = w$ then we have already proven it in (*)
—Suppose now that $c \in \bar{z}$, i.e., $c = z_i$. Since ϕ_n is *true*, then the information in the heaps h_n and h_k remains the same for such a variables. On the other hand, we have that $lv_k(z_i) = lv_{n+1}(z_i)$. Hence, $\sigma(\rho_{n+1}(z_i)) = \sigma(\rho_{k+1}(z_i)) = \sigma(\rho_k(z_i))$. But by the induction hypothesis,

$$\sigma(\rho_k(z_i)) = \alpha(z_i, \mathsf{static\_type}(z_i), RC_k)$$

But according to the argumentation above, we have then:

$$\alpha(z_i, \mathsf{static\_type}(z_i), RC_k) = \alpha(z_i, \mathsf{static\_type}(z_i), RC_{n+1})$$

—$\phi_n \neq true$. Then we can argue as in the above case except for those variables in $\bar{z}$ which are involved in ϕ_n. For such variables z_i, we have in ϕ_n a constraint of the form $\rho_{k+1}(z_i) \geq 0$ or $\rho_{k+1}(z_i) \geq 1$, according to the definition of abstract compilation. Furthermore, by the induction hypothesis $\rho_{k+1}(z_i)$ are fresh variables. Thus, we can extend σ as $\sigma(\rho_{k+1}(z_i)) = \alpha(z_i, \mathsf{static\_type}(z_i), RC_{n+1})$ and the result holds trivially.

Note that the resulting configuration at step $n + 1$ still satisfies the structural equivalence as specified at the beginning of the proof.

B.2 Proof of Lemma 5.11

We will prove this lemma by induction on the length n of the $\leadsto_\alpha$-trace. In what follows we use ϕ^q_{io} in order to refer to the input-output relation of q, and ϕ^q_{sh} in order to refer to the formula resulted from the abstract compilation of a call, i.e., the information about the variables that might be updated during the execution of call. We enrich the lemma's conditions as follows: For all $0 \leq i \leq n$:

(1) if $AC_i = \langle\langle q(\bar{x}_i, y_i), \phi^q_{sh}\rangle \cdot bc^\alpha_i, \varphi_i\rangle$, then $AC'_i = \langle\langle q(\bar{x}_i), \phi^q_{sh} \wedge \phi^q_{io}\rangle \cdot bc^\alpha_i, \varphi_i \wedge \phi_{io}\rangle$ where ϕ^q_{io} are the input output size relations for q;
(2) if $AC_i = \langle\psi_i \cdot bc^\alpha_i, \varphi_i\rangle$, then $AC'_i = \langle\psi_i \cdot bc^{io}_i, \varphi_i \wedge \phi_{io}\rangle$;
(3) if $AC_i = \langle\phi^q_{sh} \cdot bc^\alpha_i, \varphi_i\rangle$ then $AC'_i = \langle\phi^q_{sh} \wedge \phi^q_{io} \cdot bc^{io}_i, \varphi_i \wedge \phi_{io}\rangle$, where ϕ^q_{io} are the input output size relations corresponding to the last procedure call q occurring in the $\leadsto_\alpha$-trace.
(4) if $AC_i = \langle\epsilon, \varphi_i\rangle$, then $AC'_i = \langle\epsilon, \varphi_i \wedge \phi_{io}\rangle$ and $i = n$.

where ϕ_{io} corresponds to all input output size relations of all procedures whose bodies have been completely derived in the $\leadsto_{io}$-trace before step i.

Base Case ($n = 0$). Then

$$AC_0 \equiv \langle\langle p(\bar{x}, y), \phi^p_{sh}\rangle, \varphi_0\rangle$$
$$AC'_0 \equiv \langle\langle p(\bar{x}), \phi^p_{sh} \wedge \phi^q_{io}\rangle, \varphi_0\rangle$$

and the result holds trivially.

Inductive case ($n>0$)**.** Let us assume that the result holds for $\leadsto_\alpha$-traces of length $n>0$. Let us analyze the step $n+1$.

$$AC_0 \rightsquigarrow^n_\alpha AC_n \equiv \langle bc^\alpha, \varphi_n \rangle$$

and by the induction hypothesis:

$$AC'_0 \rightsquigarrow^n_{io} AC'_n \equiv \langle bc^{io}, \varphi_n \wedge \phi_{io} \rangle$$

and the conditions of the lemma are satisfied. Note that if $bc^\alpha \equiv \epsilon$ then the result holds trivially by the induction hypothesis. Let us assume that $bc^\alpha \not\equiv \epsilon$. Now, let us analyze points (1)...(3) of the statement of the lemma:

(1) Then $AC_n \equiv \langle \langle q(\bar{z}, w), \phi^q_{sh} \rangle \cdot bc^\alpha, \varphi_n \rangle$ and, by the induction hypothesis

$$AC'_n \equiv \langle \langle q(\bar{z}), \phi^q_{sh} \wedge \phi^q_{io} \rangle \cdot bc^{io}, \varphi_n \wedge \phi_{io} \rangle$$

where $q(\bar{z}, w) \leftarrow \varphi' \mid bc'^\alpha \circ \rho \ll_{AC} P^\alpha$ and $q(\bar{z}) \leftarrow \varphi' \mid bc'^{\alpha_{io}} \ll_{AC} P^{io}$. The n+1-step in the abstract compilation generates:

$$AC_{n+1} \equiv \langle bc'^\alpha \cdot \phi^q_{sh} \cdot bc^\alpha, \varphi_n \wedge \varphi' \rangle$$

where σ is a valuation such that $\sigma \models \varphi_n \wedge \varphi'$. By the induction hypothesis $\varphi_n \models \varphi_n \wedge \phi_{io}$. Then we have trivially that $\sigma \models \varphi_n \wedge \varphi' \wedge \phi_{io}$. Then we can give the following $\rightsquigarrow_{io}$-step:

$$AC'_{n+1} \equiv \langle bc'^{\alpha_{io}} \cdot \phi^q_{sh} \wedge \phi^q_{io} \cdot bc^{io}, \varphi_n \wedge \varphi' \wedge \phi_{io} \rangle$$

and the result holds.

(2) Then $AC_n \equiv \langle \psi_n \cdot bc^\alpha, \varphi_n \rangle$ and, by the induction hypothesis

$$AC'_n \equiv \langle \psi_n \cdot bc^{io}, \varphi_n \wedge \phi_{io} \rangle$$

Again, by the induction hypothesis it holds $\varphi_n \models \varphi_n \wedge \phi_{io}$. If we execute the $\rightsquigarrow_\alpha$-step, we get $AC_{n+1} \equiv \langle bc^\alpha, \varphi_n \wedge \psi_n \rangle$, where there exists a valuation σ such that $\sigma \models \varphi_n \wedge \psi_n$. By using the same σ, we have that $\sigma \models \varphi_n \wedge \psi_n \wedge \phi_{io}$. Hence we can give the corresponding $\rightsquigarrow_{io}$-step in order to compute $AC'_{n+1} \equiv \langle bc^{io}, \varphi_n \wedge \psi_n \wedge \phi_{io} \rangle$ which states the lemma.

(3) In this case $AC_n \equiv \langle \phi^q_{sh} \cdot bc^\alpha, \varphi_n \rangle$, and $AC_{n+1} \equiv \langle bc^\alpha, \varphi_n \wedge \phi^q_{sh} \rangle$, where σ is a valuation such that $\sigma \models \varphi_n \wedge \phi^q_{sh}$. By the induction hypothesis, it holds that $AC'_n \equiv \langle \phi^q_{sh} \wedge \phi^q_{io} \cdot bc^{io}, \varphi_n \wedge \phi_{io} \rangle$ and $\varphi_n \wedge \phi^q_{sh} \models \varphi_n \wedge \phi_{io}$.

Let us consider now the subtrace corresponding to the corresponding derivation of q.

$$\langle \langle q(\bar{z}, w), \phi^q_{sh} \rangle, true \rangle \rightsquigarrow^*_\alpha \langle \phi^q_{sh}, \varphi_q \rangle \rightsquigarrow_\alpha \langle \epsilon, \varphi_q \wedge \phi^q_{sh} \rangle$$

It holds trivially that $\varphi_n \models \varphi_q$. From Lemma 5.7, it holds that $\varphi_q \wedge \phi^q_{sh} \models \phi^q_{io}$. By considering σ, we have that $\sigma \models \varphi_n \wedge \phi_{io} \wedge \phi^q_{io} \wedge \phi^q_{sh}$. Hence we can execute the corresponding n+1-step in the $\rightsquigarrow_{io}$-trace which satisfies the lemma.

B.3 Proof of Theorem 5.17

This Lemma is an immediate consequence of Lemmas 5.4 and 5.11 together with the requirements for a symbolic cost model to be valid as described in Definition 5.13. The main point is that in Lemma 5.4, we have proven that at each point of the execution, whenever we execute an instruction b in the concrete trace then in the abstract we will "execute" b^α which is the abstract compilation of b with respect to some renaming ρ. Now since each cost expression is generated from b using the same ρ (possible renamed apart), then it is guaranteed (by Definition 5.13) that the cost expression $\rho(\mathcal{M}^s(b))$ is evaluated (in σ of Lemma 5.4) to exactly the cost of b, and therefore accumulating the cost of the concrete and the abstract trace results in the same cost.

Resource Usage Analysis and its Application to Resource Certification

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2],
Germán Puebla[2], and Damiano Zanardini[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. *Resource usage* is one of the most important characteristics of programs. Automatically generated information about resource usage can be used in multiple ways, both during program development and deployment. In this paper we discuss and present examples on how such information is obtained in COSTA, a state of the art static analysis system. COSTA obtains safe symbolic upper bounds on the resource usage of a large class of general-purpose programs written in a mainstream programming language such as Java (bytecode). We also discuss the application of resource-usage information for *code certification*, whereby code not guaranteed to run within certain user-specified bounds is rejected.

1 Introduction

One of the most important characteristics of a program is the amount of resources that its execution will require, i.e., its *resource usage*. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. Resource usage information has many applications, both during program development and deployment. Therefore, automated ways of estimating resource usage are quite useful and the general area of resource usage analysis (or resource analysis for short) has received considerable attention.

Statically estimating the resource usage of realistic programs is far from trivial. Thus, in the current practice, safe resource usage guarantees are only available for critical applications with strong resource usage constraints. These include real-time applications, which are required to execute within a certain maximum amount of time. Such applications are the subject of *Worst Case Execution Time Analysis* (WCET analysis for short), which is a quite active research area. See e.g. [23]. Unfortunately, WCET analysis for mainstream hardware and software is extremely complicated. On the hardware side, modern computer architectures have multiple memory levels and internal pipelining which make it rather difficult to predict the execution time of machine instructions. On the software side, accurately estimating the number of times each program loop and recursion will execute is a rather complex problem. In order to ease the situation, on the hardware side, real-time applications often run on embedded

systems whose timing behaviour is much more predictable. On the software side, real-time applications are programmed in restricted versions of general languages such as Real-Time Java or are designed and implemented using special languages such as Hume [27] and Timber [1], which are rooted in functional programming and have tool support for performing WCET analysis. Similarly, when strong memory usage limitations are in place, the programming constructs allowed are often very restricted, disallowing recursion or, as in the case of JavaCard, even strongly discouraging the use of dynamic memory allocation after the initialization phase of the applet.

In this paper we discuss the main techniques used in COSTA [5], a static analysis system which allows obtaining safe symbolic upper bounds on the resource usage of Java bytecode (JBC for short). COSTA follows the classical approach to static resource analysis proposed in Wegbreit's seminal work [57] and which consists of two phases. First, given a program and a cost model, the analysis produces *cost relations* (CRs for short). Second, the systems tries to obtain *closed-form upper-bounds* for them. The results are *symbolic* in the sense that they do not refer to concrete, platform dependent, resources such as execution time, but rather they provide platform-independent information. This has the advantage that the results are applicable to any implementation of the Java Virtual Machine (JVM) on any particular hardware. It also has the disadvantage that the information cannot refer to platform specific resources such as run-time. The fact that the analysis handles JBC represents that, at least in principle, it can deal with general-purpose programs written in a mainstream programming language such as Java and potentially other languages compiled to JBC. The upper bounds computed by COSTA can then be compared against user-provided resource usage specifications. This allows automatically rejecting code not guaranteed to execute within the specified resources.

Note that, unlike COSTA, previous resource analyses based on Wegbreit's approach have been formulated on, less widely used, declarative programming languages [57,35,44,22]. There are very few approaches for imperative programming languages [25] and, unlike COSTA, they are formulated at the *source code* level and they do not follow Wegbreit's approach. However, analyzing compile code has wider applicability, since it is quite often the case with Java applications that the *code consumer* has access to the bytecode, often bundled in *jar* files, but no access to the source code, as usual for commercial software and in mobile code. In the context of *mobile code*, programming languages which are compiled to *bytecode* and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET. Mobile code was the motivation for the concept of *Proof-Carrying Code* [41]: in order for the mobile code to be verifiable by the user, security properties (including resource usage limitations) must refer to the code available to the user, i.e., the bytecode, so that it is possible to check the provided proof and verify that the program satisfies the requirements (e.g., that the code does not require more than a certain amount of memory, or that it executes in less than a certain amount of time).

Among all possible applications of resource analysis, in this work we describe its application to *resource certification*, whereby programs are coupled with information about their resource usage. This information allows deciding whether the resources used by the program execution are acceptable or not *before* running the program. Note that resource usage can be considered a security property of untrusted mobile code, possibly in the context of proof-carrying code. Programs whose resource usage is not certified are potentially harmful, since their execution may require more resources than we are willing to spend or they may even have monetary cost by executing *billable events* such as sending text messages or making http connections on a mobile phone. In fact, mobile devices is one of the settings where resource certification is more important, because of the limited computing power typically available on mobile devices.

The rest of the paper is structured as follows. In Section 2 object-oriented bytecode, in the style of Java bytecode, is briefly described. This is required to understand the different examples in the paper, which show how resource analysis of a bytecode program is performed. Then, in Section 3, we describe, by means of examples, how to obtain CRs from a program and a cost model, whereas in Section 4 we illustrate how to obtain closed-form upper-bounds for CRs. Section 5 describes the application of resource analysis to resource certification. In Section 6 we present an overview on the existing large body of work on resource analysis. Finally, the conclusions and some venues for future work are discussed in Section 7.

2 The Context: Object-Oriented Bytecode

In order to simplify the formalization of our analysis, a simple object-oriented bytecode language is considered, which roughly corresponds to a representative subset of sequential Java bytecode. We refer to it as *simple bytecode*. For short, unless we explicitly mention *Java* bytecode, all references to bytecode in the rest of the paper correspond to our simple bytecode. Simple bytecode is able to handle integers and object creation and manipulation. For simplicity, simple bytecode does not include advanced features of Java bytecode, such as exceptions, interfaces, static methods and fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and references. Anyway, such features can be easily handled in this framework, as done in the implementation of the COSTA system.

A bytecode program consists of a set of *classes* C, partially ordered with respect to the *subclass* relation. Each class $c \in C$ contains information about the class it extends, and the fields and methods it declares. Subclasses inherit all the fields and methods of the class they extend. Each method comes with a *signature* m which consists of the class where it is defined, its name and its type. For simplicity, all methods are supposed to return a value. There cannot be two methods with the same signature. The bytecode associated to a method m is a sequence $\langle b_1, \ldots, b_n \rangle$ where each b_i is a bytecode instruction. Local variables of a k-ary method are denoted by $\langle l_0, \ldots, l_n \rangle$ with $n \geq k-1$. In contrast to Java

```
public static int binarySearch(int[ ] t, int v, int l, int u) {
   int m;
   while (l <= u) {
      m = (l + u) / 2;
      if (t[m] == v) return m;
      if (t[m] > v) u = m - 1;
      else l = m + 1;
   }
   return -1;
}
```

```
0  : load 2
1  : load 3
2  : ifgt 31
3  : load 2
4  : load 3
5  : add
6  : push 2
7  : div
8  : store 4
9  : load 0
10 : load 4
11 : aload
12 : load 1
13 : ifneq 16
14 : load 4
15 : return
16 : load 0
17 : load 4
18 : aload
19 : load 1
20 : ifleq 26
21 : load 4
22 : push 1
23 : isub
24 : store 3
25 : goto 0
26 : load 4
27 : push 1
28 : add
29 : store 2
30 : goto 0
31 : push -1
32 : return
```

Fig. 1. A Java source (left) with its corresponding bytecode (right)

source, in bytecode the *this* reference of instance (i.e., non-static) methods is passed explicitly as the first argument of the method, i.e., l_0 and $\langle l_1, \ldots, l_k \rangle$ correspond to the k formal parameters, and the remaining $\langle l_{k+1}, \ldots, l_n \rangle$ are the local variables declared in the method. For static k-ary methods $\langle l_0, \ldots, l_{k-1} \rangle$ are used for the formal parameters and $\langle l_k, \ldots, l_n \rangle$ for the local variables declared in the method. Similarly, each field f has a unique signature which consists of the class where it is declared, its name and its type . A class cannot declare two fields with the same name. The following instructions are included:

$$
\begin{aligned}
\mathit{bcInstr} ::= \ & \mathsf{load}\ i \mid \mathsf{store}\ i \mid \mathsf{push}\ n \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{div} \\
& \mid \mathsf{iflt}\ j \mid \mathsf{ifgt}\ j \mid \mathsf{ifleq}\ j \mid \mathsf{ifeq}\ j \mid \mathsf{ifneq}\ j \mid \mathsf{ifnull}\ j \mid \mathsf{goto}\ j \\
& \mid \mathsf{new}\ c \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f \\
& \mid \mathsf{newarray}\ d \mid \mathsf{aload} \mid \mathsf{astore} \mid \mathsf{arraylength} \\
& \mid \mathsf{invokevirtual}\ m \mid \mathsf{invokenonvirtual}\ m \mid \mathsf{return}
\end{aligned}
$$

Similarly to Java bytecode, simple bytecode is a stack-based language. The instructions in the first row manipulate the *operand stack*. The second row contains *jump* instructions. Instructions in the third row manipulate *objects* and their fields, while the fourth row works on *arrays*. The last row contains instructions dealing with *method invocation*. As regards notation, i is an integer which corresponds to a local variable index, n is an integer or null, j is an integer which corresponds to an index in the bytecode sequence, $c \in C$, m is a method signature, and f is a field signature.

We assume an operational semantics which is a subset of the JVM specification [37]. The execution environment of a bytecode program consists of a *heap* h and a stack A of *activation records*. Each activation record contains a program counter, a local operand stack, and local variables. The heap contains all objects and arrays allocated during the execution of the program. Each method invocation generates a new activation record according to its signature. Different activation records do not share information, but may contain references to the same objects in the heap.

Example 1. (running example) Figure 1 depicts the bytecode and a possible Java source (only shown for clarity of the presentation) of our running example. The Java program implements the binary search of an element v in a sorted array t. Variables l and u represent two indexes in the array t. If there exists a position $l \leq m \leq u$ such that t[m] is equal to v, then m is returned as result. Otherwise the method returns -1. The table of local variables is indexed from 0 to 4 and it contains the values of t, v, l, u and m respectively.

The first three instructions check the negation of the loop condition. If the check succeeds, i.e., the loop condition does not hold, then the body of the loop is not executed and the control goes to instruction 31, where the constant -1 is returned as the result (instruction 32) of the method. Otherwise the value of $(l+u)/2$ is stored in m in instructions 3–8. Then, instructions 9–13 check whether t[m]! =v holds. It the check fails, meaning that t[m]==v, then instruction 14 is executed, where variable m is pushed on the stack and returned as result in 15. Otherwise, the execution jumps to line 16, where the second `if` is checked (instructions $16, \ldots, 20$). If t[m]>v then instructions $21, \ldots, 24$ store the value of $m-1$ in u and at instruction 25 the control goes back to the beginning of the loop, i.e., instruction 0. Otherwise, the value of $m+1$ (instructions $26, \ldots 30$) is assigned to l and, similarly, control returns to instruction 0. □

3 Cost Analysis: from Bytecode to Cost Relations

This section describes how a bytecode program is analyzed in order to produce a *cost relation system* (CRS) which describes its resource consumption. The analysis consists of a number of steps: (1) the *control flow graph* of the program is computed, and afterwards (2) the program is transformed into a *rule-based representation* which facilitates the subsequent steps of the analysis without losing information about the resource consumption; (3) size analysis and abstract compilation are used to generate size relations which describe how the size of data changes during program execution; (4) the chosen cost model is applied to each instruction in order to obtain an expression which represents its cost; (5) finally, a cost relation system is obtained by joining the information gathered in the previous steps.

3.1 Control Flow Graph

The *control flow graph* of a program allows statically considering all possible paths which might be taken at runtime, which is essential information for studying its cost. Unlike structured languages such as Java, bytecode features *unstructured* control flow, due to conditional and unconditional *jumps*. Reasoning about unstructured programs is more complicated for both human-made and automatic analysis. Moreover, the control flow is made even more difficult to deal with by *virtual method invocation* and the need to handle *exceptions*. Each node of a control flow graph contains a (maximal) sequence of *non-branching* instructions, which are guaranteed to be executed sequentially. This amounts

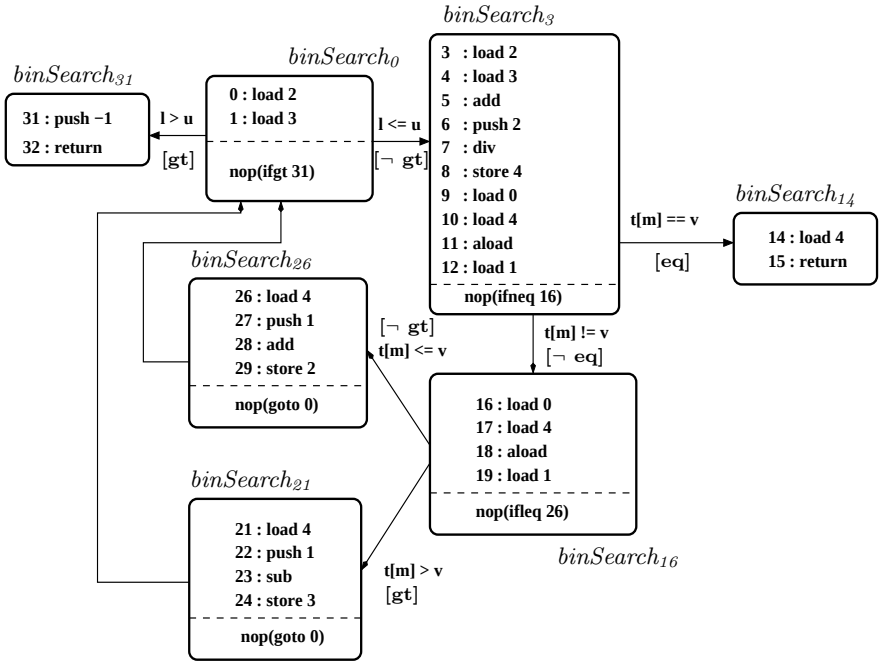


Fig. 2. The control flow graph of the running example

to saying that execution always starts at the beginning of the sequence, and the instructions in the sequence are executed one after the other until the end, without executing any other piece of code in the meanwhile.

The CFG can be built using standard techniques [2], suitably extended in order to deal with virtual method invocation. For this, it is essential to perform *class analysis* (see e.g. [51] and its references) which allows statically obtaining a safe approximation of the set of classes to which an object variable may belong to at runtime. Consider, for example, an object o, and suppose class analysis determines that a set C contains all classes to which o may belong at a given program point which contains a call of the form $o.m()$. Then, such call is translated into a set of calls $o.m_c$, one for every class $c \in C$ where m is defined. This is obtained by adding new *dispatch blocks*, containing calls $\mathsf{invoke}(c.m)$ to each implementation of m. Access to such blocks is guarded by mutually exclusive conditions on the runtime class of o.

Figure 2 shows the CFG of the running example. The graph contains 7 nodes, each composed of non-branching instructions which are always executed sequentially. All nodes end either in a return instruction (as in $binSearch_{31}$ and $binSearch_{14}$), or in an instruction labeled with $\mathsf{nop}()$, which indicates a conditional or unconditional jump in the original bytecode. Edges corresponding to conditional jumps are marked with a guard (which appears in brackets in the

figure; for clarity, conditions in the original Java program are also shown without brackets) representing the condition under which the edge can be traversed. Guards which are *true* are omitted. Instructions wrapped in a nop() have been replaced by edges in the CFG, but their place in the code is kept in order to take into account their costs when generating the corresponding cost relation system, as will be explained in Section 4 below.

3.2 Rule-based Representation

The *rule-based representation* (RBR) of a program is rich enough to preserve the information about cost, while being simple enough to develop a precise cost analysis, since some advanced features are compiled away, and control flow has been simplified. It is a *structured* procedural language with some relevant features:

1. *recursion* becomes the only iterative construct;
2. *guarded rules* are the only form of conditional construct;
3. there is only one kind of variables: *local variables*; and there is no operand stack (instead, the k-th position in the stack becomes an additional local variable s_k, exploiting the fact that, in Java bytecode, the height of the stack at each program point can be statically determined);
4. some object-oriented features are no longer present:
   - objects can be basically regarded as records including an additional field which contains their type;
   - the behaviour due to dynamic dispatch is compiled into *dispatch blocks*;
   - the language deals with the rest of object-oriented features by supporting object creation, field manipulation and arrays;
5. methods are represented as collections of related blocks, and executing a method is equivalent to executing the entry block of its representation.

These design choices help to make the generation of cost relation systems feasible, and consistent with the program structure. The rule-based representation of a program consists of a set of (global) procedures, one for each node in the CFG. Each procedure consists of one or more rules. A rule for a procedure p with k input arguments $\bar{x}$ and a (single) output argument y takes the form $p(\bar{x}, y) \leftarrow g, body$ where $p(\bar{x}, y)$ is the *head*, g is a *guard* expressing a boolean condition, and *body* is (a suitable representation of) the instructions which are contained in the node.

A guard can be either *true*, any linear condition about the value of variables (e.g., $x + y > 10$), or a type check type(x, c). Every (non-branching) instruction in the body is represented in a more readable (and closer to source code) syntax than the original bytecode (Figure 3). E.g., the instruction load i which loads the i-th local variable l_i into a new topmost stack variable s_{t+1} is written as $s_{t+1} := l_i$ (remember that variables named s_k originate from the k-th position in the stack but they are actually local variables in the RBR). Moreover, add is translated to $s_{t-1} := s_{t-1} + s_t$, where t is the current height of the stack in the original program, and putfield f is turned into $s_t.f := s_t$. As in the control

b_j	comp(b_j)
load i	$s_{t+1} := l_i$
store i	$l_i := s_t$
push n	$s_{t+1} := n$
pop	nop(pop)
dup	$s_{t+1} := s_t$
add	$s_{t-1} := s_{t-1} + s_t$
sub	$s_{t-1} := s_{t-1} - s_t$
lt	$s_{t-1} < s_t$
gt	$s_{t-1} > s_t$
eq	$s_{t-1} = s_t$
null	$s_t =$ null
¬ lt	$s_{t-1} \geq s_t$
¬ gt	$s_{t-1} \leq s_t$

b_j	comp(b_j)
¬ eq	$s_{t-1} \neq s_t$
¬ null	$s_t \neq$ null
type(n, c)	type(s_{t-n}, c)
new c	$s_{t+1} :=$ new c
getfield f.	$s_t := s_t.f$
putfield f.	$s_{t-1}.f := s_t$
newarray$c$	$s_t :=$ newarray(c, s_t)
aload	$s_{t-1} := s_{t-1}[s_t]$
astore	$s_{t-2}[s_{t-1}] := s_t$
arraylength	$s_t :=$ arraylength(s_t)
invoke m	$m(s_{t-n}, \ldots, s_t, s_{t-n})$
return	$out := s_t$
nop(b)	nop(b)

Fig. 3. Compiling bytecode instructions (as they appear in the CFG) to rule-based instructions (t stands for the height of the stack before the instruction).

flow graph, branching instructions such as jumps and calls (which have become edges in the CFG, but may still be relevant to the resource consumption) are wrapped into a nop(_) construct, meaning that they are not executed in the RBR, but will be taken into account in the following steps of the analysis. RBR programs are restricted to *strict determinism*, i.e., the guards for all rules for the same procedure are pairwise mutually exclusive, and the disjunction of all such guards is always true.

A CFG can be translated into a rule-based program by building a rule for every node of the graph, which executes its sequential bytecode, and calls the rules corresponding to its successors in the CFG. Figure 4 shows the rule-based program for the CFG of Figure 2. The RBR for the *binSearch* method has an *entry procedure* which simply initializes all local variables and calls $binSearch_0$. In turn, the body of $binSearch_0$ loads l and u (corresponding to l and u), and calls its *continuation* $binSearch_0^c$, that decides which block will be executed next, depending on the comparison between s_1 and s_2 (note that the guards of the continuation rules are mutually exclusive). Procedures which are not continuations are named after the corresponding nodes in the CFG. Note that rules $binSearch_{26}$ and $binSearch_{21}$ contain a call to $binSearch_0$, which is in fact the loop condition.

An *operational semantics* can be given for the rule-based representation, which mimics the bytecode one. In particular, executing an RBR program still needs a *heap* and a *stack* of activation records. The main difference between the two semantics lies in the granularity of procedures: every method in the bytecode program has been partitioned into a set of procedures. In spite of this, it can be proven that any rule-based program is *cost-equivalent* to the bytecode program it comes from. Intuitively, cost-equivalence means that no information about the resource consumption is lost. The main cost-equivalence result states that the

$$
\begin{array}{l}
\mathit{binSearch}(t,v,l,u,r) \leftarrow \mathit{init\_vars}(m), \mathit{binSearch}_0(t,v,l,u,m,r) \\
\mathit{binSearch}_0(t,v,l,u,m,r) \leftarrow \\
\quad s_1 := l, s_2 := u, \mathsf{nop}(\mathit{if\_icmpgt}\ 31), \mathit{binSearch}_0^c(t,v,l,u,m,s_1,s_2,r). \\
\mathit{binSearch}_0^c(t,v,l,u,m,s_1,s_2,r) \leftarrow s_1{>}s_2, \mathit{binSearch}_{31}(t,v,l,u,m,r). \\
\mathit{binSearch}_0^c(t,v,l,u,m,s_1,s_2,r) \leftarrow s_1{\leq}s_2, \mathit{binSearch}_3(t,v,l,u,m,r). \\
\mathit{binSearch}_{31}(t,v,l,u,m,r) \leftarrow s_1 := -1, r := s_1. \\
\mathit{binSearch}_3(t,v,l,u,m,r) \leftarrow \\
\quad s_1 := l, s_2 := u, s_1 := s_1{+}s_2, s_2 := 2, s_1 := s_1/s_2, m := s_1, s_1 := t, s_2 := m, \\
\quad s_1 := s_1[s_2], s_2 := v, \mathsf{nop}(\mathit{if\_icmpne}\ 16), \mathit{binSearch}_3^c(t,v,l,u,m,s_1,s_2,r). \\
\mathit{binSearch}_3^c(t,v,l,u,m,s_1,s_2,r) \leftarrow s_1{=}s_2, \mathit{binSearch}_{14}(t,v,l,u,m,r). \\
\mathit{binSearch}_3^c(t,v,l,u,m,s_1,s_2,r) \leftarrow s_1{\neq}s_2, \mathit{binSearch}_{16}(t,v,l,u,m,r). \\
\mathit{binSearch}_{14}(t,v,l,u,m,r) \leftarrow s_1 := m, r := s_1. \\
\mathit{binSearch}_{16}(t,v,l,u,m,r) \leftarrow \\
\quad s_1 := t, s_2 := m, s_1 := s_1[s_2], s_2 := v, \\
\quad \mathsf{nop}(\mathit{if\_icmple}\ 26), \mathit{binSearch}_{16}^c(t,v,l,u,m,s_1,s_2,r). \\
\mathit{binSearch}_{16}^c(t,v,l,u,m,s_1,s_2,r) \leftarrow s_1{\leq}s_2, \mathit{binSearch}_{26}(t,v,l,u,m,r). \\
\mathit{binSearch}_{16}^c(t,v,l,u,m,s_1,s_2,r) \leftarrow s_1{>}s_2, \mathit{binSearch}_{21}(t,v,l,u,m,r). \\
\mathit{binSearch}_{26}(t,v,l,u,m,r) \leftarrow \\
\quad s_1 := m, s_2 := 1, s_1 := s_1{+}s_2, v := s_1, \mathsf{nop}(\mathit{goto}\ 0), \mathit{binSearch}_0(t,v,l,u,m,r). \\
\mathit{binSearch}_{21}(t,v,l,u,m,r) \leftarrow \\
\quad s_1 := m, s_2 := 1, s_1 := s_1{-}s_2, u := s_1, \mathsf{nop}(\mathit{goto}\ 0), \mathit{binSearch}_0(t,v,l,u,m,r).
\end{array}
$$

Fig. 4. RBR of the example (guards which are *true* are omitted).

execution from cost-equivalent input configurations for a bytecode program and its RBR leads to (1) non-termination in both cases; or (2) cost-equivalent output configurations.

3.3 Cost Relations

Given a program P (without loss of generality, it is supposed here that P has already been translated into its RBR form) and a cost model $\mathcal{M}$, the classical approach to cost analysis [57] consists in generating a set of *recurrence relations* (RRs) which capture the cost (w.r.t. $\mathcal{M}$) of running P on some input. As usual in this area, data structures are replaced by their *sizes* in the recurrence relations. From rule-based programs it is possible to obtain *cost relations* (CRs), an extended form of recurrence relations, which approximate the cost of running the corresponding programs. In the presented approach, each rule in the RBR program results in an equation in the CRS. Figure 5 shows the cost relation system (i.e., a system of cost relations) for the running example, where it is easy to see in the rule names the correspondence with the rule-based representation. In these equations, variables are in fact constraint variables which correspond to the sizes of those of the RBR. The right-hand side of an equation consists of an expression e which gives the cost of executing the body of the rule, and, for simplicity of the subsequent presentation, a linear constraint φ which denotes the

$$
\begin{array}{lll}
(1)\ \mathit{binSearch}(t,v,l,u) & = \mathit{binSearch}_0(t,v,l,u,0) & \\
(2)\ \mathit{binSearch}_0(t,v,l,u,m) & = 3 + \mathit{binSearch}_0^c(t,v,l,u,m,s_1,s_2) & \{s_1{=}l, s_2{=}u\} \\
(3)\ \mathit{binSearch}_0^c(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{31}(t,v,l,u,m) & \{s_1{>}s_2\} \\
(4)\ \mathit{binSearch}_0^c(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_3(t,v,l,u,m) & \{s_1{\leq}s_2\} \\
(5)\ \mathit{binSearch}_{31}(t,v,l,u,m) & = 2 & \\
(6)\ \mathit{binSearch}_3(t,v,l,u,m) & = 11 + \mathit{binSearch}_3^c(t,v,l,u,m',s_1,s_2) & \\
 & \quad \left\{ s_2{=}v, m' \in \left[\frac{l+u}{2} - \frac{1}{2}, \frac{l+u}{2} \right] \right\} & \\
(7)\ \mathit{binSearch}_3^c(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{16}(t,v,l,u,m) & \\
(8)\ \mathit{binSearch}_3^c(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{14}(t,v,l,u,m) & \\
(9)\ \mathit{binSearch}_{16}(t,v,l,u,m) & = 5 + \mathit{binSearch}_{16}^c(t,v,l,u,m,s_1,s_2) & \{s_2{=}v\} \\
(10)\ \mathit{binSearch}_{14}(t,v,l,u,m) & = 2 & \\
(11)\ \mathit{binSearch}_{16}^c(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{26}(t,v,l,u,m) & \\
(12)\ \mathit{binSearch}_{16}^c(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{21}(t,v,l,u,m) & \\
(13)\ \mathit{binSearch}_{26}(t,v,l,u,m) & = 5 + \mathit{binSearch}_0(t,v,l',u,m) & \{l'{=}m{+}1\} \\
(14)\ \mathit{binSearch}_{21}(t,v,l,u,m) & = 5 + \mathit{binSearch}_0(t,v,l,u',m) & \{u'{=}m{-}1\}
\end{array}
$$

Fig. 5. CRS of the running example

effect of the body on the variables. An important point to note is that, there are some cases where the simplification above may be incorrect. We opt by keeping this simplification in the presentation, though not in the implementation, because problems are rare and otherwise the presentation gets more complicated. In more detail, input-output size relations cannot always be merged together in φ. Constraints which originate from input-output relations of procedures called in the body of the rule cannot be taken into account until after the corresponding calls. This is because, by merging them, we can no longer distinguish finite failures from infinite failures. For instance, this happens when we have a procedure, say p, which never terminates. The input-output relation for p is represented with the constraint *false*, indicating that there are no successful executions for p. Any equation which has a call to p will have $\varphi = false$. If, by mistake, we take this *false* as a finite failure, we would incorrectly discard (part of) this equation as unreachable, when in reality execution never returns from this equation. In our running example, this phenomenon does not happen since even after adding constraints originating from input-output relations, no φ becames *false*.

Finally, note also that the output variable of the rule does not appear in the equation, as explained below. The generation of a cost equation for a given RBR rule goes through the following steps.

Size Measures. A *size measure* is chosen to represent and manipulate information relevant to cost, and a variable is abstracted to its *size* w.r.t. such measure. For example, (1) an array may be abstracted to its length, since this can typically give information about the cost of traversing it in a loop; or (2) an object can be abstracted to the longest path reachable from it (in this case, the size measure is well-known and is called *path-length* [52]) in order to describe the

cost of traversing data structures such as trees or linked lists. The choice of a size measure, in particular for heap structures, heavily depends on the program to be analyzed, and is intended to represent the maximum amount of relevant information. E.g., in cost and termination analysis, the measure used to abstract a piece of data or a data structure should give information about the behavior of a loop whose exit condition depends, as in the examples above, on the data.

Abstract Compilation. In the presented setting, one important issue is to capture relations between the size of a variable at different program points. For example, in analyzing $x := x + 1$, the interest usually lies in the relation "the value of x *after* is equal to 1 plus the value of x *before*".

In this steps of the cost analysis, instructions are replaced by *linear constraints* which approximate the relation between states (and, typically, between different program points) w.r.t. the chosen size measure. For instance, $s_1 := o$ is replaced by $s_1{=}o$, meaning that, after the assignment, the size of s_1 at the current program point is equal to the size of o. As another example, $x :=$ new c can be replaced, using the path-length measure, by $x{=}1$, meaning that the maximal path reachable from x after the object creation has length 1.

Importantly, the use of path-length as a size measure for reference requires extra information in order to obtain precise and sound results in the abstract compilation of instructions involving references:

(a) *sharing* information [48] is required in order to know whether two references may point to a common region of the heap; and
(b) *non-cyclicity* information [46] is required to guarantee that, at some specific program point, a reference points to a non-cyclic data structure, i.e., that the length of its longest path (therefore, the number of iteration on a typical traversing loop) is guaranteed to be finite.

A slightly more complicated example where non-cyclicity information is used is represented by a field access $x := y.f$: in this case

- no linear constraint can be inferred if f is a non-reference field;
- if y is detected as non-cyclic, then the size of x after the assignment can be guaranteed to be *strictly less* than the size of y before (since the data structure pointed by x is now a sub-structure of the one pointed by y);
- if y may be cyclic, then the size of x can only be taken to be *not greater* than the size of y (thus basically forbidding to find useful results on x and y in the following steps, as explained in Section 4).

The result of this *abstract compilation* is an *abstract program* which can be used to approximate the values of variables w.r.t. the given size measure.

Input-Output Size Relations. As mathematical relations, CRs cannot have output variables: instead, they should receive a set of input parameters and return a number which represents the cost of the associated computation. This

step of the analysis is meant to transform the abstract program in order to remove output variables from it. The basic idea relies on computing abstract *input-output (size) relations* in terms of linear constraints, and using them to propagate the effect of calling a procedure. Concretely, input-output size relations of the form $p(\bar{x}, y) \rightarrow \varphi$ are inferred, where φ is a constraint describing the relation between the sizes of the input $\bar{x}$ and the output y upon exit from p. This information is needed since the output of one call may be input to another call. Interestingly, input-output relations can be seen also as a denotational semantics for the abstract programs previously obtained. Sound input-output size relations can be obtained by taking abstract rules generated by abstract compilation, and combine them via a fixpoint computation [13], using abstract interpretation techniques [20] in order to avoid infinite computations.

Example 2. Consider the following RBR rules

$$\begin{aligned} incr(this, i, out) &\leftarrow incr_1(this, i, out) \\ incr_1(this, i, out) &\leftarrow s_1 := i,\ s_2 := 2,\ s_1 := s_1 + s_2,\ out := s_1 \end{aligned}$$

which basically come from the method

```
int incr(int i) { return i+2; }
```

All variables relevant to the computation are integers, so that abstract compilation abstracts every variable into itself (due to the choice of the size measure for numeric variables), and the abstract program looks like (constraints $\{s_1 = 0, s_2 = 0, out = 0\}$ describe the initial values of variables)

$$\begin{aligned} incr(this, i, out) &\leftarrow incr_1(this, i, out) \\ incr_1(this, i, out) &\leftarrow \{s_1 = 0, s_2 = 0, out = 0\} \mid \\ &\qquad s'_1 = i,\ s'_2 = 2,\ s''_1 = s'_1 + s'_2,\ out' = s''_1 \end{aligned}$$

By combining the constraints through the bodies, it can be inferred that the output value of *out* is 2 plus the input value of i, which, in the end, is represented by the input-output size relation

$$incr(this, i, out) \leftarrow \{out = i + 2\}$$

□

Cost Models. Resource usage analysis is a clear example of a program analysis where the focus is not only on the input-output behavior (i.e., *what* a program computes), but also on the history of the computation (i.e., *how* the computation is performed). Since the history of a computation can be normally extracted by its trace, it is natural to describe resource usage in terms of execution traces.

The notion of a *cost model* for bytecode programs formally describes how the resource consumption of a program can be calculated, given a resource of interest. It basically defines how to measure the resource consumption, i.e., the cost, associated to each execution step and, by extension, to an entire trace.

In the present setting, a cost model can be viewed as a function from a bytecode instruction, and dynamic information (local variables, stack, and heap) to a real number. Such number is the amount of resources which is consumed when executing the current step in the given configuration. Example 3 below introduces some interesting cost models which will be used in the next sections.

Example 3. The *instructions* cost model, denoted $\mathcal{M}_{inst}$, counts the number of bytecode instructions executed by giving a constant cost 1 to the execution of any instruction in any configuration: it basically measures the length of a trace.

The *heap* cost model, $\mathcal{M}_{heap}$, is used for estimating the amount of memory allocated by the program for dynamically storing objects and arrays (i.e., its heap consumption): it assigns to any instruction the amount of memory which it allocates in the current configuration. For instance, newarray *int* (resp., newarray c) allocates $v * size(int)$ (resp., $v * size(ref)$) bytes in the heap, where v denotes the length of the array (currently stored on the top of the stack), and $size(int)$ (resp., $size(ref)$) is the size of an integer (resp., a reference) as a memory area. □

Generation of Cost Relation Systems. Cost relation in a CRS are generated by using the abstract rules to build the constraints, and the original rule together with the selected cost model to generate *cost expressions* representing the cost of the bytecodes w.r.t. the model. Consider the cost relations identified by equations (6), (7) and (8) in the CRS of the running example (Figure 5), reproduced here for more clarity.

$$(6)\quad binSearch_3(t,v,l,u,m) = 11 + binSearch_3^c(t,v,l,u,m',s_1,s_2) \left\{ s_2{=}v, m' \in \left[\frac{l+u}{2}-\frac{1}{2}, \frac{l+u}{2}\right] \right\}$$

$$(7)\quad binSearch_3^c(t,v,l,u,m,s_1,s_2) = binSearch_{16}(t,v,l,u,m)$$

$$(8)\quad binSearch_3^c(t,v,l,u,m,s_1,s_2) = binSearch_{14}(t,v,l,u,m)$$

This excerpt shows that the inferred cost of executing $binSearch_3$ amounts to 11 plus the cost of executing $binSearch_3^c$. In turn, the cost of $binSearch_3^c$ can be either the cost of $binSearch_{16}$ or the cost of $binSearch_{14}$. In this case, cost expressions (as 11 in (6), or 0, left implicit in (7) and (8)) are simply constant values, which correspond to the number of executed instructions, since the cost model $\mathcal{M}_{inst}$ has been chosen. That is, eleven instruction are executed in $binSearch_3$ before calling $binSearch_3^c$, while no instructions are executed before calling $binSearch_{16}$ or $binSearch_{14}$ from $binSearch_3^c$. The constraints which appear at the end of some equations (as in (6); see also the complete CR for more examples) will be used in the following section. CRs extend recurrence relations in the sense that they allow to handle advanced features such as *non-determinism* (see for instance equations (7) and (8)) *constraints*, and *multiple arguments*, which arise in the cost analysis of realistic programs.

4 From Cost Relations to Closed-Form Upper Bounds

Though cost relations (CRs) are simpler than the programs they originate from, since all variables have integer type, in several respects they are not *as static as* one would expect from the result of a static analysis. First, cost relations are recursive, so that one may need to iterate for computing their value on a concrete input. Second, even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in cost relations which are non-deterministic. This happens in the above example in the loop inside binarySearch: since the array t is abstracted to its length, the contents of the array are lost in the abstraction. In particular, the value of t[m] is unknown in the CR. Hence, the pairs of Equations 7–8 and 11–12 end up having the same guards and the evaluation of this CR turns out to be non-deterministic. In order to find the worst-case cost, one would need to compute and compare many results. In some cases, the number of results may even be infinite. For both reasons, it is clear that it is interesting to compute *closed-form* upper bounds for the cost relation, whenever this is possible, i.e., upper bounds which are in non-recursive form. For example, the goal is to infer that the cost of calling $binSearch(t, v, l, u)$ is $24 * \lceil \mathsf{log}_2(\mathsf{nat}(u - l) + 1) \rceil + 40$, where $\mathsf{nat}(a) = max(a, 0)$.

Since CRs are syntactically quite close to *Recurrence Relations* (*RRs* for short), in most resource analysis frameworks, it has been assumed that cost relations can be easily converted into *RRs*. This has led to the belief that it is possible to use existing *Computer Algebra Systems* (CAS for short) for finding closed forms of the relations generated by resource analysis. As it will be shown, cost relations are far from *RRs*, and using CAS to obtain closed-form upper bounds is, in general, not practical, and requires a considerable amount of human intervention in many phases.

The main idea in the approach used in the COSTA system is to view CRSs as *programs*, and then use semantic-based static-analysis and program-transformations techniques in order to infer closed-form upper bounds [8]. We first explain the basic ideas on small examples, then we explain how they can be extended to the general case.

4.1 Bounds on the Number of Applications of Equations

The first dimension of the problem of obtaining closed-form upper bounds is to bound the number of recursive calls in each relation, which directly affects the number of times an equation can be applied. Consider, for example, the following cost relation:

$$\begin{array}{ll} C(n) = 3 & \{n \leq 0\} \\ C(n) = 9 + C(n') & \{n > 0, n' < n\} \end{array}$$

An evaluation of an initial call $C(v)$, where v is an integer value works as follows: if $v \leq 0$ then we apply the first equation and we accumulate 3 units to the cost, and if $v > 0$ then we apply the second equation, which in turn accumulates 9 units to the cost plus the cost of the recursive call $C(v')$ where v' is an integer

number such that $v' < v$ (which corresponds to the constraint $n' < n$). Clearly, if I_r and I_b are, respectively, upper bounds on the *number of applications* of the recursive and base-case equations, then $9 * I_r + 3 * I_b$ is an upper bound on the corresponding cost. In the above example, in each recursive call the argument of C decreases at least by 1 (since $n' < n$), and therefore the maximum number of applications of the second equation is $I_r = n_0$, where n_0 corresponds to the (initial) input value, and $I_b = 1$, since the base-case equation is applied only once. Note that when n_0 is negative, we do not make any recursive call, therefore in order to capture these cases we define $I_r = \mathsf{nat}(n_0)$ where $\mathsf{nat}(a) = max(a, 0)$. Putting everything together we obtain that an upper bound for the call $C(n_0)$ is $9 * \mathsf{nat}(n_0) + 3$.

The above example demonstrates that inferring how the values of arguments change during evaluation plays an important role in bounding the number of application of each equation. This change might come in different forms, for example if we change the second equation in the above CR to

$$C(n) = 8 + C(n') \quad \{n > 0, n' \leq \tfrac{n}{2}\}$$

then C's argument decreases by at least half in each recursive call, and therefore the maximum number of application of the recursive equation is $I_r = \lceil \mathsf{log}_2(\mathsf{nat}(n_0) + 1) \rceil$ which in turn implies that the upper bound would be $8 * \lceil \mathsf{log}_2(\mathsf{nat}(n_0) + 1) \rceil + 3$.

Another important factor that affects the number of applications of the different equations is the number of recursive calls in a single equation. For example, assuming that the recursive equation in the above CR is of the form

$$C(n) = 7 + C(n') + C(n'') \quad \{n > 0, n' < n, n'' < n\}$$

then the recursive equation would be applied in the worst-case $I_r = 2^{\mathsf{nat}(n_0)} - 1$ times, because each call generates 2 recursive calls, and in each call the argument decreases at least by 1. Note that $2^{\mathsf{nat}(n_0)} - 1$ corresponds to the number of internal nodes which a complete binary tree of height $\mathsf{nat}(n_0)$ has. In addition, unlike the above examples, the base-case equation would be applied in the worst-case $I_b = 2^{\mathsf{nat}(n_0)}$ times, and therefore the upper bound would be $7 * (2^{\mathsf{nat}(n_0)} - 1) + 3 * 2^{\mathsf{nat}(n_0)}$.

In general, a CR does not include only two equations as above. It may include several base cases and/or several recursive equations. In addition, equations are not necessarily mutually exclusive, which means that at each evaluation step there are several equations that can be applied. For example, if all three recursive equations that we have seen above are defined in the same CR, then the upper bound would be $max([7, 8, 9]) * (2^{\mathsf{nat}(n_0)} - 1) + 3 * 2^{\mathsf{nat}(n_0)}$. Note that the worst-case for the cost of each application is determined by the first equation, which contributes the largest cost, i.e., 9. The worst case for the number of applications of the recursive case is determined by the third equation, which has two recursive calls.

As we explained at the beginning, the problem of bounding the number of applications of each equation is related to bounding the number of consecutive

recursive calls, which has been extensively studied in the context of termination analysis. Automatic termination analyzers usually prove that an upper bound of the consecutive recursive calls exists by proving that there exists a function f from the loop's arguments to a *well-founded* partial order, such that f decreases in any two consecutive calls. This, in turn, guarantees the absence of infinite traces, and therefore termination. These functions are usually called *ranking functions*. If instead of proving that such function exists we actually compute one, then we can use it as the upper bound on the number of consecutive calls, which in turn can be used to bound the number of applications.

4.2 Bounds on the Cost of Equations

In the above examples, in each application the corresponding equation contributes a constant number of cost units. This is not the case in general. For example, it is common to have a CR of the following form:

$$\begin{array}{ll} C(n) = 3 & \{n \leq 0\} \\ C(n) = \mathsf{nat}(n+2)^2 + C(n') & \{n > 0, n' \leq \frac{n}{2}\} \end{array}$$

where in the second equation we accumulate a non-constant, i.e., $\mathsf{nat}(n+2)^2$, number of units in each application. In equations with a non-constant direct cost expression, a closed-form upper bound can be obtained by considering the worst-case (the maximum) value that the expression can be evaluated to, multiplied by the number of applications of the corresponding equation. For example, in the above equation, the maximum value that the expression $(n+2)$ can be evaluated to is (n_0+2), and therefore we would produce the upper bound $\mathsf{nat}(n_0+2)^2 * \lceil \mathsf{log}_2(\mathsf{nat}(n_0)+1) \rceil + 3$.

In order to infer the maximum value of non-constant expressions automatically, we first infer *invariants* (linear relations) between the equation's variables and the values which such variables had at the initial call, and then *maximize* the expression w.r.t. these values. For the above example we would infer the relation $\{n_0 \geq n > 0\}$, from which we can see that the maximum value for n is n_0. This in turn implies that (n_0+2) is the maximum value of $(n+2)$ and therefore $\mathsf{nat}(n_0+2)^2$ is the maximum value for $\mathsf{nat}(n+2)^2$. Again, if several recursive equations are involved, we should combine them all using the *max* operator on the corresponding expressions, as we have done above.

4.3 The General Case

In all the above examples, a single relation was involved and all recursions were direct. We refer to such CRs as *stand-alone* CRs. This is not the case in general. Instead, in most cases, CRSs consist of several CRs with complex call graphs. In order to cope with this, we first transform the given CRS into a structured form with only direct recursions, and incrementally apply the above techniques. We do so by first applying *Partial Evaluation* [32], a well-known program transformation technique, to each of the strongly connected components (SCC) in the

corresponding call graph. By applying partial evaluation starting from a cover point of the SCC, it is guaranteed that get rid of mutual recursion. After partial evaluation, there must be at least one stand-alone CR (does not call any other CR), therefore we can apply the techniques described above in order to solve all these stand-alone CRs. Substituting the results in their calling contexts results in more stand-alone CRs that can in turn be solved using the above techniques again. This process is repeated until there are no more CRs left.

4.4 Obtaining an Upper Bound for the Running Example

The CRS for the running example, shown in Figure 5, contains multiple relations and includes non-direct recursion, which avoids obtaining a closed-form upper bound in a compositional way. As explained above, the first step is to transform the CRS into an equivalent one where we have only direct recursion. The CRS depicted in Figure 5 has 2 SCCs: the first SCC only has Equation 1, and it is not recursive; the second SCC contains Equations 2–14 and corresponds to the loop in *binSearch* and is therefore recursive. After applying partial evaluation to the recursive SCC we obtain the following transformed CRS:

$$
\begin{array}{lll}
(1)\ \mathit{binSearch}(t,v,l,u) & = \mathit{loop}(t,v,l,u,0) & \\
(2)\ \mathit{loop}_b(t,v,l,u,m) & = 5 & \{l{>}u\} \\
(3)\ \mathit{loop}_b(t,v,l,u,m) & = 16 & \{l{\leq}u\} \\
(4)\ \mathit{loop}_b(t,v,l,u,m) & = 24 + \mathit{loop}_b(t,v,l',u,m') & \\
 & & \{l{\leq}u, m' \in [\frac{l+u}{2}-\frac{1}{2}, \frac{l+u}{2}], l'{=}m'{+}1\} \\
(5)\ \mathit{loop}_b(t,v,l,u,m) & = 24 + \mathit{loop}(t,v,l,u',m') & \\
 & & \{l{\leq}u, m' \in [\frac{l+u}{2}-\frac{1}{2}, \frac{l+u}{2}], u'{=}m'{-}1\}
\end{array}
$$

The recursive SCC has been transformed into Equations 2–5 above. Note that Equations 3–5 have the same guard ($l{\leq}u$), which results in a non-deterministic CR. The reason for this is that Equation 3 corresponds to the case where $t[m] == v$, Equation 4 to the case where $t[m] > v$ and Equation 5 to the case where $t[m] < v$. However, the value of $t[m]$ is not observable at the cost relation level and even though the original program is deterministic, its associated CRS is not.

Solving the above CRS starts by solving the stand-alone CR which consists in Equations 2–5. Note that Equations 2–3 are the base-cases and 4–5 are the recursive ones. Examining the recursive equations and their attached constraints, we can automatically infer that the difference between the values of u and l decreases logarithmically at each recursive call and, in particular, we can provide $I_r = \lceil \mathsf{log}_2(\mathsf{nat}(u_0 - l_0) + 1) \rceil + 1$ as an upper bound for the number of applications of the recursive equation. The base-case equations are applied only once. Therefore the closed-form upper bound is:

$$\mathit{loop}_b(t_0, v_0, l_0, u_0, m_0) = 24 * (\lceil \mathsf{log}_2(\mathsf{nat}(u_0 - l_0) + 1) \rceil + 1) + 16$$

This closed form can then be substituted in Equation 1 and after some simplification we obtain the following closed-form upper bound for *binSearch*:

$$\mathit{binSearch}(t_0, v_0, l_0, u_0) = 24 * \lceil \mathsf{log}_2(\mathsf{nat}(u_0 - l_0) + 1) \rceil + 40$$

In order to illustrate the use of invariants and the maximization of cost expressions, let us now assume that the method *binSearch* is used in another method as follows:

```
public static int m(int[ ] t){
  int c=0;
  int u=t.length;
  for (int i=0; i<u;i++)
    if (binarySearch(t,i,i,u) != -1 ) c++;
  return c;
}
```

and that we are interested in inferring closed-form upper bounds for m. We first build the CRS that corresponds to m. For brevity, we show the CRS after partial evaluation:

$$
\begin{array}{lll}
(1)\ m(t) & = 9 + \mathit{loop}_m(t,c,u,i) & \{i{=}0, u{=}t, c{=}0\} \\
(2)\ \mathit{loop}_m(t,c,u,i) & = 3 & \{i{\geq}u\} \\
(3)\ \mathit{loop}_m(t,c,u,i) & = 12{+}\mathit{binSearch}(t,i,i,u){+}\mathit{loop}_m(t,c,u,i') & \{i{<}u, i'{=}i{+}1\} \\
(4)\ \mathit{loop}_m(t,c,u,i) & = 13{+}\mathit{binSearch}(t,i,i,u){+}\mathit{loop}_m(t,c',u,i') & \{i{<}u, i'{=}i{+}1, c'{=}c{+}1\}
\end{array}
$$

First we solve the CR loop_m (Equations 2–4). We start by substituting the close-form upper bound of $\mathit{binSearch}_m$ in the corresponding calls and we obtain the following stand-alone CR

$$
\begin{array}{ll}
\mathit{loop}_m(t,c,u,i) = 3 & \{i{\geq}u\} \\
\mathit{loop}_m(t,c,u,i) = 24 * \lceil \mathsf{log}_2(\mathsf{nat}(u-i)+1)\rceil + 52{+}\mathit{loop}_m(t,c,u,i') & \{i{<}u, i'{=}i{+}1\} \\
\mathit{loop}_m(t,c,u,i) = 24 * \lceil \mathsf{log}_2(\mathsf{nat}(u-i)+1)\rceil + 53{+}\mathit{loop}_m(t,c',u,i') & \{i{<}u, i'{=}i{+}1, c'{=}c{+}1\}
\end{array}
$$

Examining the recursive equations and their attached constraints we automatically infer that the maximum number of applications of the recursive equations is $\mathsf{nat}(u_0{-}i_0)$, since i increases by one until it reaches u (which does not change). Now, in order to infer the closed-form upper bound we need to approximate the maximum values that $\mathsf{log}_2(\mathsf{nat}(u-i)+1)$ can be evaluated to. This happens when $u-i$ is maximal. This occurs for the maximal values of u and the minimal values of i. Since the invariant that we infer includes $i \geq i_0$ and $u \leq u_0$, we can conclude that $u_0 - i_0$ is the maximum value to which $u - i$ can be evaluated. Therefore the closed-form upper bound for loop_m is:

$$\mathit{loop}_m(t_0,c_0,u_0,i_0) = \mathsf{nat}(u_0-i_0) * (24 * \lceil \mathsf{log}_2(\mathsf{nat}(u_0-i_0)+1)\rceil + 53) + 3$$

Substituting this upper bound in the first equation results in a non-recursive CR which consists in a single equation:

$$m(t) = 9 + \mathsf{nat}(u-i) * (24 * \lceil \mathsf{log}_2(\mathsf{nat}(u-i)+1)\rceil + 53) + 3 \ \{i{=}0, u{=}t, c{=}0\}$$

and since in this context we have $i = 0$ and $u = t$, we can conclude with the following closed-form upper bound for m:

$$m(t) = 24 * \mathsf{nat}(t) * \lceil \mathsf{log}_2(\mathsf{nat}(t)+1)\rceil + 53 * \mathsf{nat}(t) + 12\,)$$

5 Application to Resource Certification

In order to motivate the interest of *resource usage certification* or (resource certification for short), we will start by describing *mobile code*. Nowadays, the use of mobile code is widespread. It includes, for example running applets and/or plug-ins downloaded from the net in a web browser or a mobile phone. The current approach to security of mobile code is a combination of static verification of certain properties, which guarantees a certain level of security, with dynamic checking, which supervises all operations which are still potentially unsecure after the static verification. For example, in the Java Virtual Machine, mobile code is subject to *bytecode verification* before being executed, while operations such as array indexing are checked at runtime. Bytecode verification, if successful, provides a number of guarantees on the program, such as being well typed, with jumps to existing instructions, etc. Note that if the bytecode verification process fails, the program is discarded.

Ideally, one would like to extend this model in order to include more sophisticated *security policies* in the static verification part. In particular, and as already sketched in Section 1, the purpose of resource certification is to consider resource usage bounds as security policies. This means that prior to executing a program, it must be guaranteed that the program satisfies a given resource usage policy. This problem can be formulated in two ways. One is to have an automatic system which given a program and a resource usage policy answers *yes* only if it succeeds to prove that the program satisfies the policy. Alternatively, we can split this process in two steps: first, an automatic system obtains an upper bound on the resource usage of the program and second, another automatic system, which in what follows we refer to as *comparator*, checks whether the computed upper bound is smaller than or equal to the resource usage policy for any possible input value. We advocate for the second alternative because we believe it is more flexible: we first use COSTA on the code producer side to infer upper bounds which are independent of any resource policy and consumer, and then, on the code consumer side we check whether the upper bound abides by the policy.

5.1 An Example of Resource Certification

We illustrate through a simple example the fundamental intuition behind resource certification. Let us assume a resource usage policy for method m in Figure 1 that imposes a resource usage policy, which we call *policy*, on the number of instructions executed of:

$$policy{=}60 * [\mathsf{nat}(t)]^2 + 120 * \mathsf{nat}(t) + 13$$

COSTA infers the upper bound $ub{=}24{*}\mathsf{nat}(t){*}\lceil \mathsf{log}_2(\mathsf{nat}(t){+}1)\rceil {+}53{*}\mathsf{nat}(t){+}12$. The code will be acceptable, provided that *policy* is guaranteed, i.e., $ub{\leq}policy$, which happens to be the case in our example and that the comparator succeeds to prove it.

Developing a comparator which handles closed-forms that involve logarithmic, exponential, polynomial expressions, etc. is far from trivial. Based on the ideas in [26], we are currently implementing in COSTA the basics of such comparator, but it is still subject of ongoing work.

Also, though in some contexts, especially when considering memory usage, non-asymptotic policies are to be expected, sometimes it is more reasonable that the policy is asymptotic. In COSTA, policies are currently non-asymptotic and handling of asymptotic policies is also subject of ongoing work. Coming back to our previous example, this would result in a new $policy'$ s.t. $policy'=[\mathsf{nat}(t)]^2$. The comparator should again be able to prove that $policy'$ is satisfied by method m.

5.2 Scenarios for Resource Certification

Within the alternative we propose, in which code certification is performed in two steps, there are several *scenarios* one could imagine. We now describe three different ones.

The Consumer-based Scenario In this first scenario, it is the sole responsibility of the mobile code consumer to both obtain an upper bound and to compare such upper bound with the policy. This scenario is simple, since it does not involve any further actor, but it is inefficient, since the mobile code has to be certified separately for every consumer. Also, it may be unfeasible on devices with limited computing power, such as mobile phones.

The Server-based Scenario In this second scenario, there is an additional actor which acts as the server of the mobile code. Such server not only distributes the mobile code. It also computes once and for all an upper bound for it. Assuming that this server is trusted by the code consumer, the consumer downloads a bundle which contains both the code and its upper bound. In order to guarantee that the bundle is actually produced by the trusted server, the bundle is signed using standard *Public Key Infrastructure* (PKI) techniques. Then, the code consumer, using the public key of the server, checks that the bundle is correct and uses the provided upper bound. Similarly, the comparison phase could also either be outsourced to trusted servers and be accessed using PKI or be performed locally.

The PCC-based Scenario In this final scenario, the situation is somewhat intermediate between the two other extremes. The main advantage of the pcc-based scenario w.r.t. the server-based one is that in the pcc-scenario the server does not need to be trusted by the code consumer. Unlike the simpler notion of PKI (which merely guarantees that the code has been produced or approved by the signing entity, such as a program, person, or organization), now the pcc server provides an unsigned bundle which contains the code, an upper bound,

and some *verifiable evidence*[3] about the upper bound being correct. Then, the code consumer has to have an automatic (and efficient) system for verifying that the provided upper bound is actually valid for the code, by using the provided evidence.

As it is well know from the proof-carrying code [41] theory, the main advantage of this scenario is that the evidence only needs to be generated once and the verification process which occurs at the consumer side should be much more efficient than computing the upper bounds from scratch. Essentially, the hard work is shifted from the code consumer to the code producer (i.e., the programmer and/or the compiler), which now has to not only produce the code, but also an upper bound and the verifiable evidence which must be bundled with it.

In the case of COSTA, a PCC-based scenario can obtained by using ideas from Abstraction-Carrying Code [7] (ACC), which proposes to use abstract interpretation as enabling technology for PCC. The main idea in ACC is to use, at the producer's side, a fixed point-based static analyzer, in order to automatically infer an abstract model (or simply *abstraction*) of the mobile code which can then be used to prove that this code is secure w.r.t. the given policy in a straightforward way. A simple, easy-to-trust (analysis) verifier at the consumer's side could verify the validity of the information on the mobile code. This verifier could be indeed a specialized abstract interpreter whose key characteristic is that it does not need to iterate in order to reach a fixed point (in contrast to standard analyzers). Furthermore, as the process of inferring the abstraction is fully automatic, the analyzer itself could be used at the consumer side, as discussed in the consumer-based scenario above.

We are currently working on building a PCC infrastructure for COSTA by following the principles of ACC. Since the analyzer computes several abstractions of the program (size relations, invariants, ranking functions, etc.) in order to be able to compute an upper bound, the evidence should in principle contain the fixed points of multiple analyses. However, depending on the analysis times and the amount of space required to store its result, for some analyses it may be more efficient to recompute things on the consumer side than to verify the evidence provided by the server. Thus, there are still important practical decisions regarding which analyses results to include in the evidence and which to recompute on the consumer.

In our opinion, regardless of which of the three scenarios are put into practice, generalized use of resource certification will not be a reality until there are fully automatic resource analyzers available which are capable of computing accurate upper bounds for real-life applications. This is the requirement which COSTA aims at solving. Once this is sufficiently solved, the rest of the infrastructure will be in place relatively easily.

[3] In the original PCC framework, this *evidence* was called *certificate*. We prefer avoiding the use of such terminology since it is already rather overloaded.

6 Related Work

In this section, we review related work by focusing first in existing tools developed for the analysis and transformation of Java bytecode in Section 6.1. Then, in Section 6.2, we give a brief overview of the features that resource analyses have on the different programming paradigms and the most interesting aspects of each of them. Later, in Section 6.3, we compare our system for obtaining closed-form upper bounds with existing solvers. Finally, in Section 6.5, we summarize the work on certification of the resource consumption of programs.

6.1 Tools for Analysis of Java Bytecode

Analysis of Java bytecode is currently an active research area with a number of analysis and transformation tools available. Especially relevant are the analyses developed on the Soot framework [54] and the Julia generic analyzer [50]. Soot is a framework for the development of optimizations and analyses for Java bytecode which already includes points-to, purity, and dynamic data structure analyses, among others. The most similar part between these systems and COSTA is the transformation of the bytecode into an intermediate (procedural) representation. Indeed, intermediate representations are common practice to develop analysis and transformations on JBC. Of relevant importance is BoogiePL [36] as well. The main differences with our RBR are: (1) they do not provide a uniform treatment of all kinds of loops by means of recursion, (2) they do not perform the loop extraction transformation we propose, which is important for compositionality in resource analysis; and (3) the intermediate representation called Shimple in Soot performs SSA, but neither Shimple nor BoogiePL convert stack variables into local variables as COSTA does. In our representation, in one pass, we can eliminate almost all variables which originate from stack variables, which results in a more efficient subsequent size analysis. The Julia Java bytecode analyzer [50] provides a generic analysis engine for which sharing, class, non-nullness, information flow, escape, constancy, and static initialisation analyses have been integrated. Neither Julia nor Soot include a resource analysis, though Julia also contains implementations of some of the pieces (in particular the class, nullity, sharing, and cyclicity analyses) which are required in the size analysis component.

6.2 Resource Analysis for Different Programming Paradigms

Focusing on resource analysis, important effort has been devoted to extend Wegbreit's framework [57] to different languages and programming paradigms. The main objective in this task is to define a resource analysis framework in which it is possible to generate CRS from the programs in the corresponding language. As mentioned in Section 1, most of the extensions to Wegbreit's framework have taken place in the context of high-level declarative languages, whose recursive structure simplifies the process of generating cost relations. In general, these analyses consider languages without a mutable heap, and they do not deal with

objects and exceptions as in our case. We are not aware of any work, apart from ours, that applies Wegbreit's framework to imperative languages. Below we review several frameworks defined for the corresponding declarative programming paradigms.

Cost Analysis in Functional Programming. Early work on resource analysis [57,35,44] was developed for a first order subset of Lisp. Rosendahl [44] presented a system based on transforming a program into a step-counting version which was then analyzed by relying on abstract interpretation. The result of such analysis was expressed as a CRS which was then attempted to be transformed into a closed form by relying on a series of source-to-source transformations. Theoretical advances for analyzing lazy functional languages were made by [56] and [15]. They used projections and demand analysis to model a call-by-need reduction strategy of typed lambda calculus. Still in the context of functional languages, the technique of cost counting programs mentioned above [44,35] was extended in [47] to higher-order programs. Recent work [33] describes a complexity analysis for programs extracted from proofs carried out with the Coq proof assistant. The generated CRSs are solved in this case by relying on MAPLE. Again, the first transformational part is not required and size analysis does not have to deal with object-oriented features. An automatic complexity analysis for computing upper bounds on the time complexity of higher-order Nuprl programs is presented in [14]. The analysis derives recursive cost equations which are passed to Mathematica. In general, in functional programming, resource analysis focuses on dealing with higher-order functions and lazy evaluation.

Cost Analysis in Logic Programming. One of the first resource analysis frameworks [22] was developed in the context of logic programming. In this setting, resource analysis needs to consider peculiar features of logic languages, such as approximating the number of solutions (due to non-deterministic computations), type and mode inference, and non-failure information. The CASLOG system [22] was designed to solve CRSs for logic program and it is currently used in the `CiaoPP` system [28]. As in functional programming, obtaining CRSs is simplified by the fact that they already start from a recursive programming language where recursion is the only form of iteration. Also, size analysis in logic programming differs from ours as it does not support object-oriented features. The resource analysis integrated in the `CiaoPP` system includes a resource analysis [40] based on a size analysis for logic programs and hence differs fundamentally from ours.

6.3 Systems for Computing Closed-Form Upper-Bounds

There are two main ways of viewing CRSs which lead to different mechanisms for finding closed-form upper-bounds. We call the first view *algebraic* and the second view *transformational*. The algebraic one is based on regarding CRSs as *recurrence relations*. This view was the first one to be proposed and it is the one which is advocated for in a larger number of works. It allows reusing the large existing body of work in solving recurrence relations. Within this view,

two alternatives have been used in previous analyzers. One alternative consists in implementing restricted recurrence solvers based on standard mathematical techniques within the analyzer, as done in e.g. [57,22]. The other alternative, motivated by the availability of powerful *computer algebra systems* (CASs for short) such as Mathematica, MAXIMA, MAPLE, etc., consists in connecting the analyzer with an external solver, as proposed in [56,47,14,6,38].

The transformational view consists in regarding CRSs as (functional) programs. In this view, closed-form upper-bounds are produced by applying (general-purpose) program transformation techniques on the CRS [44] until a non-recursive program is obtained. The transformational view was first proposed in the ACE system [35], which contained a large number of program transformation rules aimed at obtaining non-recursive representations. It was also used by Rosendahl in [44], who later in [45] provided a series of program transformation techniques based on super-compilation [53] which were able to obtain closed-forms for some classes of programs.

The need for improved mechanisms for automatically obtaining closed-form upper-bounds was already pointed out in Hickey and Cohen [29]. A significant work in this direction is PURRS [10], which has been the first system to provide, in a fully automatic way, non-asymptotic closed-form upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires CRSs to be deterministic. The problem with all the approaches mentioned above is that, though they can be successfully applied for obtaining closed-forms for CRSs generated from simple programs, they do not fulfill the initial expectations in that they are not of general applicability to CRSs generated from real programs.

The main motivation for developing the solver [8] that we use in COSTA was our own experience in trying to apply the algebraic approach on the CRSs generated by [6]. We argue that automatically converting CRSs into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. In contrast, our approach can produce correct and comparatively simple results even in the presence of non-determinism.

6.4 Other Approaches to Cost Analysis

In the imperative programming paradigm, most of the work has been done by the real-time and embedded systems community. It has mainly focused on real-time aspects, with major inroads made in WCET analysis, see e.g. [23], which is technically different from our resource analysis, the main similarity being the need to infer upper bounds on the number of iterations of loops.

There exist other approaches to resource analysis which are not based on Wegbreit's framework. These include analyses based on type-and-effect systems [11,49,55]. Type-and-effect systems [42] are a well-known technique for automatic program analysis. They main difference w.r.t. abstract interpretation approaches like ours is that they avoid having the implementation of specialised

inference engines that may be required by abstract interpretation and they simplify the construction of the soundness proofs through analogy with similar and well-understood proofs for the underlying type system. The latest work by [49] uses a type-and-effect system based on Hindley-Milner types to expose constraints on sized types [31] for higher order, recursive functional programs, to provide improved quality of resource analysis. Apart from the underlying differences between the considered languages, in contrast to our proposal, this approach to resource analysis is restricted to linear upper bounds. Besides, the language does not support recursion and the analysis is restricted to a cost model that counts the number of steps. The analysis presented in [11] proposes an extension of the λ-calculus to ensure that resources are correctly used. They also rely on a type-and-effect system to over approximate the set of histories of events (i.e., the usage of resources) that a program can generate at runtime. A model-checking technique then validates such approximations. In essence, this work is focused on the enforcement of resource usage policies, but their techniques cannot be used to generate upper bounds on the resource usage as our method does.

There is also work which studies the relationship between syntactical constructions of programming languages and their computational complexity [34,12]. These analyses are developed on simple imperative languages which are far from our bytecode and, in contrast to our work, they cannot be used to compute non-asymptotic upper bounds.

The work in [39] shows how to apply sub-interpretation (firstly used in first order functional programming to deal with computational complexity) to object-oriented programs without recursion in order to provide upper bounds on their stack usage. This approach is restricted to polynomial bounds and to the particular resource of stack usage.

More recent work develops resource analyses to estimate the memory consumption. In particular, [16] describes a technique for Java-like languages which computes symbolic polynomial approximations of the amount of memory required by a program. The work by [19] studies the memory consumption (including both heap space and stack usage) of low-level programs which are similar to our bytecode programs. In both cases, the analyses are less general than ours, both in the kind of properties they can estimate (specific to memory consumption) and in the kind of upper bounds that they can generate (polynomial bounds).

The SPEED system [25,24] is able to automatically compute symbolic complexity bounds of procedures written in C/C++. The basic idea of their methodology is to instrument monitor variables to count the number of loop iterations and then statically compute an upper bound on these counter variables in terms of programming inputs using invariant generation tools. They allow the user the possibility of defining some quantitative functions over abstract data-structures to avoid the need of shape analysis. Besides, SPEED performs some program transformations to improve the precision of the analysis when inferring bounds

on certain types of loops. Some of these ideas could be applied in order to improve our framework.

6.5 Resource Usage Certification

As already mentioned in Section 5, resource usage certification [21,9,30,18,43] proposes the use of security properties involving resource requirements, i.e., that the untrusted code adheres to specific bounds on resource consumption. Related work in the context of Java bytecode includes the work in the MRG project [9], which can be considered complementary to ours. MRG focuses on building a proof-carrying code [41] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The cost model which has been used to develop the analysis is heap consumption, since applications to be deployed on devices with a limited amount of memory, such as smartcards, should be rejected if they require more memory than that available. The framework is restricted to polynomial bounds and to the above cost model, while our resource analysis can infer a wider set of bounds (including exponential, algorithmic, etc.) and it is parametric with respect to the cost model. More related work is the one proposed by [17], where a resource usage analysis is presented. Again, this work focuses on one particular notion of cost, memory consumption, and it aims at verifying that the program executes in bounded memory by making sure that the program does not create new objects inside loops, but it does not infer resource usage bounds. The analysis has been certified by proving its correctness using the Coq proof assistant. Compared to previous work, our system shows, for the first time, that it is possible to automatically generate resource bounds guarantees, not restricted to polynomial bounds, for a realistic mobile language.

7 Conclusions and Future Perspectives

In this paper we have illustrated, by means of examples, how the COSTA system performs resource analysis. The analysis is done in two steps. First, *cost relation systems* are generated for an input bytecode w.r.t. a *cost model*. Such relations provide useful approximations of the resource usage of the program w.r.t. the considered cost model, in terms of the size of the input arguments, and provided an accurate *size analysis* is used to establish relationships between arguments. Second, *closed-form upper bounds* for the cost relation systems are obtained. This is possible provided that *ranking functions* are found for all loops which affect the cost and that accurate *invariants* are obtained. To the best of our knowledge, COSTA is the first system to perform fully automatic resource analysis of object-oriented bytecode and we believe that COSTA opens the door to the application of *resource usage analysis* in the context of general purpose applications written in mainstream programming languages.

Though the efficiency and robustness of the system can be considerably improved, COSTA can already deal with a relatively large class of JBC programs, and gives reasonable results in terms of precision and efficiency for different

cost models: the number of executed bytecode instructions, heap consumption, and number of calls to user-specified methods. We plan to distribute the system as free software soon. Currently, it can be tried out through a web interface available from the COSTA web site: `http://costa.ls.fi.upm.es`.

The system can deal with most features of JBC. However, non-sequential code, dynamic class loading and reflection are not supported. Java API methods used by programs are analyzed much in the same way as user code, since their bytecode is available to the analyzer. As for native code, i.e., methods not implemented in Java, calls to native methods are shown in upper bounds as symbolic constants, since the code for those methods is not written in Java and COSTA cannot analyze them. This could be further improved by providing assertions which describe the cost of the native method for the different cost models and (optionally) a safe approximation of their input-output behavior, but it is not supported.

In addition to the web interface, COSTA has a command-line interface and an Eclipse plugin which make interaction with the analyzer quite straightforward, even during program development. The different interfaces allow customizing the behaviour of COSTA by modifying the value of several options, including:

1. whether the code of Java API classes should also be analyzed or not;
2. whether auxiliary analyses (sign, nullity, slicing, constant propagation) should be included, thus possibly improving both precision and performance;
3. whether input-output size relations have to be computed (Section 3.3);
4. if exceptions, either explicitly thrown in the code or resulting from semantic violations, have to be taken into account;
5. which cost model has to be considered.

Also, although not discussed in this paper, COSTA also performs termination analysis of Java bytecode programs (see [3]). When COSTA fails to find an upper bound for a program, sometimes it may be useful to try and find out whether the program is guaranteed to terminate. Maybe COSTA fails because the program contains a bug and loops unexpectedly with a non-zero cost associated to each iteration of the loop. In that case, there exist no upper bound for the program and there is no way that COSTA can find an upper bound. Although COSTA results are safe, they are obviously incomplete, since finding an upper bound is an undecidable problem. This means that there are programs for which it is possible to find an upper bound, but COSTA fails to find one.

As regards future work, there are plenty of ways in which both the theoretical foundations and the practical implementation can be improved in order to handle a larger class of programs, and obtain improvements both in terms of efficiency and accuracy. On the foundations side, progress in the area of object-oriented languages of any of the analyses used by the system will be potentially applicable to COSTA. For example, one of the most challenging problems is to account for loops and recursion where the number of iterations depends on *numeric fields*. Here, an approach working in all cases might not be practical; however, heuristics may allow us to account for special, simple but quite common cases which can significantly enlarge the class of analyzable programs. A first step in

this direction, in the context of termination analysis, has been taken in [4]. On the implementation side, currently, COSTA handles bytecode programs for Java SE 1.4.2_13 and Java ME. The reason for this is that Java SE 1.4.2_13 is the version of Java taken as starting point for Java ME and, in particular, for MIDP. The latter is the profile used by mobile phone applications, i.e., *midlets*, which are the main target in the MOBIUS project. However, there is no fundamental reason for not supporting more recent versions of Java and we plan to extend COSTA to also handle Java 5 and 6 soon.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. The Timber Language. http://www.timber-lang.org.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.
4. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Dealing with numeric fields in termination analysis of java-like languages. In Marieke Huisman, editor, *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.
5. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. The COSTA System web site. `http://costa.ls.fi.upm.es`.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
7. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code: A Model for Mobile Code Safety. *New Generation Computing*, 26(2):171–204, March 2008.
8. Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.
9. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 1–27. Springer, 2005.

10. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, 2005. `arXiv:cs/0512056` available from `http://arxiv.org/`.
11. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007*, volume 4423 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2007.
12. Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 2008.
13. Florence Benoy and Andy King. Inferring Argument Size Relationships with CLP(R). In *Workshop on Logic-based Program Synthesis and Transformation (LOPSTR)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, August 1997.
14. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.
15. B. Bjerner and S. Holmstrom. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.
16. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.
17. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *13th International Symposium on Formal Methods (FM'05)*, number 3582 in LNCS, pages 91–106. Springer-Verlag, 2005.
18. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.
19. W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.
20. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
21. K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.
22. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
23. Jochen Eisinger, Ilia Polian, Bernd Becker, Alexander Metzner, Stephan Thesing, and Reinhard Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 15–20. IEEE Computer Society, April 2006.
24. B. S. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*, LNCS 5123, pages 370–384. Springer, 2008.
25. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.

26. Sumit Gulwani and Ashish Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
27. K. Hammond and G. Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2003.
28. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
29. T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35(1), 1988.
30. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.
31. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423, 1996.
32. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
33. J. Jouannaud and W. Xu. Automatic Complexity Analysis for Programs Extracted from Coq Proof. *ENTCS*, 2006.
34. Lars Kristiansen and Neil D. Jones. The flow of data and the complexity of algorithms. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *CiE*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2005.
35. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.
36. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.
37. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
38. Beatrice Luca, Stefan Andrei, Hugh Anderson, and Siau-Cheng Khoo. Program transformation by solving recurrences. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 121–129, New York, NY, USA, 2006. ACM.
39. J.-Y. Marion and R. Pèchoux. Resource control of object-oriented programs. In *International LICS affiliated Workshop on Logic and Computational Complexity (LCC 2007)*, Wroclaw, Poland, 2007.
40. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.
41. G. Necula. Proof-Carrying Code. In *Proc. of ACM Symposium on Principles of programming languages (POPL)*, pages 106–119. ACM Press, 1997.
42. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. Second Ed.
43. K-H. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.
44. M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

45. M. Rosendahl. Simple driving techniques. In T. Mogensen, D. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 404–419. Springer, 2002.
46. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*. S-V, 2006.
47. D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.
48. S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *Static Analysis Symposium (SAS)*, pages 320–335, 2005.
49. H. R. Simões, K. Hammond, M. Florido, and P. B. Vasconcelos. Using intersection types for cost-analysis of higher-order polymorphic functional programs. In *Types for Proofs and Programs, International Workshop, TYPES 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2006.
50. F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.
51. F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.
52. Fausto Spoto, Patricia M. Hill, and Etienne Payet. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
53. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
54. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.
55. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2003.
56. P. Wadler. Strictness analysis aids time analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM Press, 1988.
57. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

Heap Space Analysis for Java Bytecode

Elvira Albert
DSIC, Complutense University of Madrid, Spain
elvira@sip.ucm.es

Samir Genaim
CLIP, Technical University of Madrid, Spain
samir@clip.dia.fi.upm.es

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid, Spain
mzamalloa@fdi.ucm.es

Abstract

This article presents a heap space analysis for (sequential) Java bytecode. The analysis generates *heap space cost relations* which define at compile-time the heap consumption of a program as a function of its data size. These relations can be used to obtain upper bounds on the heap space allocated during the execution of the different methods. In addition, we describe how to refine the cost relations, by relying on *escape analysis*, in order to take into account the heap space that can be safely deallocated by the garbage collector upon exit from a corresponding method. These refined cost relations are then used to infer upper bounds on the *active heap space* upon methods return. Example applications for the analysis consider inference of constant heap usage and heap usage proportional to the data size (including polynomial and exponential heap consumption). Our prototype implementation is reported and demonstrated by means of a series of examples which illustrate how the analysis naturally encompasses standard data-structures like lists, trees and arrays with several dimensions written in object-oriented programming style.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Heap Space Analysis, Heap Consumption, Low-level Languages, Java Bytecode

1. Introduction

Heap space analysis aims at inferring *bounds* on the heap space consumption of programs. Heap analysis is more typically formulated at the source level (see, e.g., [24, 17, 25, 19] in the context of functional programming and [18, 13] for high-level imperative programming languages). However, there are situations where one has only access to compiled code and not to the source code. An example of this is *mobile code*, where the code consumer receives code to be executed. In this context, Java bytecode [20] is widely used, mainly due to its security features and the fact that it is platform-independent. Automatic heap space analysis has interesting applications in this context. For instance, *resource bound certification* [14, 4, 5, 16, 12] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered. A general framework for the cost analysis of sequential Java bytecode has been proposed in [2]. Such analysis statically generates *cost relations* which define the cost of a program as a function of its input data size. The cost relations are expressed by means of *recursive equations* generated by abstracting the recursive structure of the program and by inferring size relations between arguments. Cost relations are parametric w.r.t. a *cost model*, i.e., the cost unit associated to the bytecode b appears as an abstract value T_b within the equations.

This article develops a novel application of the cost analysis framework of [2] to infer bounds on the heap space consumption of sequential Java bytecode programs. In a first step, we develop a cost model that defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap (memory) units it consumes. E.g., the cost of creating a new object is the number of heap units allocated to that object. The remaining bytecode instructions do not add any cost. With this cost model, we generate heap space cost relations which are then used to infer upper bounds on the heap space usage of the different methods. These upper bounds provide information on the maximal heap space required for executing each method in the program. In a second step, we refine this cost model to consider the effect of garbage collection. This is done by relying on escape analysis [15, 8] to identify those memory allocation instructions which create objects that will be

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.

garbage collected upon exit from the corresponding method. With this information available, we can generate heap space cost relations which contain annotations for the heap space that will be garbage collected. The annotated cost relations in turn are used to infer upper bounds on the *active heap space* upon exit from methods, i.e., the heap space consumed and that might not be garbage collected upon exit.

A distinguishing feature of the approach presented in this article w.r.t. previous type-based approaches (e.g., [5, 17]) is that it is not restricted to linear bounds since the generated cost relations can in principle capture any complexity class. Moreover, in many cases, the relations can be simplified to a *closed form* solution from which one can glean immediate information about the expected consumption of the code to be run. The approach has been assessed by means of a prototype implementation, which originates from the one of [3]. It should be noted that the examples in [3] are simple imperative algorithms which did not make use of the heap, since they were aimed at demonstrating that traditional complexity schemata can be handled by the cost analysis of [2]. In contrast, we demonstrate our heap analysis by means of a series of example applications written in an object-oriented style which make intensive use of the heap and which present novel features like heap consumption that depends on the class fields, multiple inheritance, virtual invocation, etc. These examples allow us to illustrate the most salient features of our analysis: inference of constant heap usage, heap usage proportional to input size, support of standard data-structures like lists, trees, arrays, etc. To the best of our knowledge, this is the first analysis able to infer arbitrary heap usage bounds for Java bytecode.

The rest of the paper is structured as follows: Sec. 2 presents an example that illustrates the ideas behind the analysis. Sec. 3 briefly describes the Java bytecode language. Sec. 4 defines a cost model for heap consumption and describes the analysis framework. Sec. 5 demonstrates the different features of the analysis by means of examples. In Sec.6, we extend our cost model to consider the effect of garbage collection. Sec. 7 reports on a prototype implementation and some experimental results. Finally, Sec. 8 concludes and discusses the related work.

2. Worked Example

Consider the Java classes and their corresponding (structured) Java bytecode depicted in Fig. 1 which define a linked-list data structure in an object-oriented style, as it appears in [18]. The class *Cons* is used for data nodes and the class *Nil* plays the role of *null* to indicate the end of a list. Both classes define a copy function which is used to clone the corresponding object. In the case of *Nil* the copy method just returns *this* since it is the last element of the list, and in the case of *Cons* it clones the current object and its successors recursively (by calling the *copy* method of *next*). The rest of this section describes the different steps applied by the analyzer to approximate the heap consumption of the program depicted in Fig. 1. Note that the Java program is provided here just for clarity, the analyzer works directly on the bytecode which is obtained, for example, by compiling the Java program.

Step I: In the first step, the analyzer recovers the structure of the Java bytecode program by building a control flow graph (CFG) for its methods. The CFG consists of basic blocks which contain a sequence of non-branching bytecode instructions, these blocks are connected by edges that describe the possible flows that originate from the branching instructions like conditional jumps, exceptions, virtual method invocation, etc. In Fig. 1, the CFG of the method *Nil.copy* consists of the single block $Block_0^{Nil}$ and the CFG of the method *Cons.copy* consists of the rest of the blocks. $Block_0^{Cons}$ corresponds to the bytecode of *Cons.copy* up to the recursive method call *this.next.copy()*. Then, depending on the type of the object stored in *this.next* the execution is transferred to either *Nil.copy* or *Cons.copy*. This is expressed by the (guarded) branching to $Block_1^{Cons}$ and $Block_2^{Cons}$. In both cases, the control returns to $Block_3^{Cons}$ which corresponds to the rest of the statements.

Step II: In the second step, the analyzer builds an intermediate representation for the CFG and uses it to infer information about the changes in the sizes of the different data-structures (or in the values of integer variables) when the control passes from one part of the program (e.g., a block or a method) to another part. For example, this step infers that when *Nil.copy* or *Cons.copy* are called recursively, the length of the list decreases by one. This information is essential for defining the heap consumption of one part of the program in terms of the heap consumption of other parts.

Step III: In the third step, the intermediate representation and the size information are used together with the cost model for heap consumption to generate a set of cost relations which describe the heap consumption behaviour of the program. The following equations are the ones we get for the example in Fig. 1:

Heap Space Cost Equations			Size relations
$C_{copy}^{Nil}(a)$	=	0	$\{a{=}1\}$
$C_{copy}^{Cons}(a)$	=	$C_0(a)$	
$C_0(a)$	=	$size(Cons) + CC_0(a,b)$	$\{a{\geq}1, b{\geq}0, a{=}b{+}1\}$
$CC_0(a,b)$	=	$\begin{cases} C_1(a,b) & \hat{b} \in Nil \\ C_2(a,b) & \hat{b} \in Cons \end{cases}$	
$C_1(a,b)$	=	$C_{copy}^{Nil}(b) + C_3(a)$	$\{a{=}1\}$
$C_2(a,b)$	=	$C_{copy}^{Cons}(b) + C_3(a)$	$\{a{\geq}2\}$
$C_3(a)$	=	0	

Each of these equations corresponds to a method entry, block or branching in the CFG. An equation is composed by the left hand side which indicates the block or the method it represents, and the right hand side which defines its heap consumption behaviour. In addition, size relations might be attached to describe how the data size changes when using another equation.

```
abstract class List {
    abstract List copy();
}
class Nil extends List {
    List copy() {
        return this;
    }
}
class Cons extends List {
    int elem;
    List next;
    List copy(){
        Cons aux = new Cons();
        aux.elem = this.elem;
        aux.next = this.next.copy();
        return aux;
    }
}
```

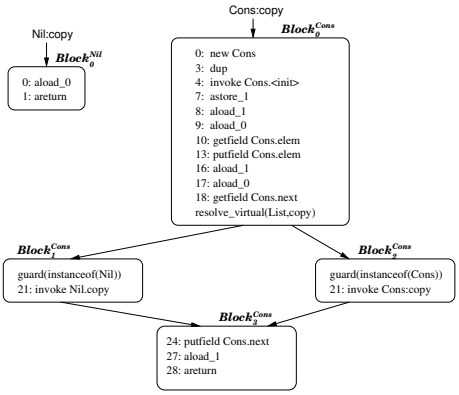


Figure 1. Java source code and CFG bytecode of example

The equation $C^{Nil}_{copy}(a)$ defines the heap consumption of *Nil.copy* in terms of (the size of) its first argument a which corresponds to its *this* reference variable (in Java bytecode the *this* reference variable is the first argument of the method). In this case the heap consumption is zero since the method does not allocate any heap space. The equation $C^{Cons}_{copy}(a)$ defines the heap consumption of *Cons.copy* as the heap consumption of $Block^{Cons}_0$ using the corresponding equation C_0, which in turn defines the heap consumption as the amount of heap units allocated by the *new* bytecode instructions, namely $size(Cons)$, plus the heap consumption of its successors which is defined by the equation CC_0. All other instructions in $Block^{Cons}_0$ contribute zero to the heap consumption. Note that in C_0, the variable b corresponds to *this.next* of *Cons.copy* and that the size analysis is able to infer the relation $a{=}b{+}1$ (i.e., the list a is longer than b by one). The equation CC_0 corresponds to the heap consumption of the branches at the end of $Block^{Cons}_0$, depending on the type of b (denoted as $\hat{b}$) it is equal to the heap consumption of $Block^{Cons}_1$ or $Block^{Cons}_2$ which are respectively defined by the equations C_1 and C_2. The equation C_1 defines the heap consumption of $Block^{Cons}_1$ as the heap consumption of *Nil.copy* (since it is called in $Block^{Cons}_1$) plus the heap consumption of $Block^{Cons}_3$ (using the equation C_3). Similarly C_2 defines the heap consumption of $Block^{Cons}_2$ in terms of the heap consumption of *Cons.copy*. The equation C_3 defines the heap consumption of $Block^{Cons}_3$ to be zero since it does not allocate any heap space.

Step IV: In the fourth step, we can simplify the equations and try to obtain an upper bound in closed form for the cost relation by applying the method described in [1]. In particular, assuming that $size(Cons)$ equals 8 (4 bytes for the integer field *data* and 4 bytes for the reference field *next*), we obtain the following simplified equations:

Equation	Size relations
$C^{Nil}_{copy}(a) = 0$	$\{a{=}1\}$
$C^{Cons}_{copy}(a) = 8$	$\{a{=}2\}$
$C^{Cons}_{copy}(a) = 8 + C^{Cons}_{copy}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

and then obtain an upper bound in closed form $C^{Cons}_{copy}(a) = 8 * (a-1)$.

The main focus of this paper is on the generation of heap space cost relations, as illustrated in Step III. Steps I and II are done as it is proposed in [2] and Step IV as it is described in [1] and hence we will not give many details on how they are performed in this paper.

3. The Java Bytecode Language

Java bytecode [20] is a low-level object-oriented programming language with unstructured control and an *operand stack* to hold intermediate computational results. Moreover, objects are stored in dynamic memory: the *heap*. A Java bytecode program consists of a set of *class files*, one for each class or interface. A class file contains information about its *name* $c \in Class\_Name$, the class it extends, the interfaces it implements, and the fields and methods it defines. In particular, for each method, the class file contains: a method signature which consists of its name and its type; its bytecode $bc_m = \langle pc_0{:}b_0, \ldots, pc_{n_m}{:}b_{n_m}\rangle$, where each b_i is a *bytecode instruction* and pc_i is its address; and the method's exceptions table. In this work we consider a subset of the JVM [20] language which is able to handle operations on integers and references, object creation and manipulation (by accessing fields and calling methods), arrays of primitive and reference types, and exceptions (either generated by abnormal execution or explicitly thrown by the program). For simplicity, we omit static fields and initializers and primitive types different from integers. Such features could be handled by making the underlying abstract interpretation support them by assuming the worst case approximation for them. Thus, our bytecode instruction set ($bcInst$) is:

$bcInst$ ::=
push x | istore v | astore v | iload v | aload v | iconst a
| iadd | isub | imul | idiv | if$\diamond$ pc | goto pc | ireturn | areturn
| return | new $Class\_Name$ |
| newarray int | anewarray $Class\_Name$ | iaload | aaload
| iastore | aastore | athrow | dup
| invokevirtual/invokespecial $Class\_Name.Meth\_Sig$
| getfield/putfield $Class\_Name.Field\_Sig$

where $\diamond$ is a comparison operator (ne,le,_icmpgt, etc.), v a local variable, a an integer, pc an instruction address, and x an integer or the special value `null`.

4. The Heap Space Analysis Framework

Cost analysis of a low-level object-oriented language such as Java bytecode is complicated mainly due to its unstructured control flow (e.g., the use of goto statements rather than recursive structures), its object-oriented features (e.g., virtual method invocation) and its stack-based model. The recent work of [2] develops a generic framework for the automatic cost analysis of Java bytecode programs. Essentially, the complications of dealing with a low-level language are handled in this framework by abstracting the *recursive structure* of the program and by inferring *size relations* between arguments. As we have seen in Sect. 2, this analysis framework is based on transforming the Java bytecode program to an intermediate representation which fits inside the same setting all possible forms of loops. Then, using this intermediate representation, the analysis infers information about the change in the sizes of the relevant data structures as the program goes through its loops (Steps I and II). Finally, this information is used to set up a cost relation which defines the cost of the program in terms of the sizes of the corresponding data structures.

In this section, we present a novel application of this generic cost analysis framework to infer bounds on the heap space consumption of sequential Java bytecode programs. So far, this framework has been only used in [3] to infer the complexity of some classical algorithms while in this paper our purpose is completely different: we aim at computing bounds on the heap usage for programs written in object-oriented programming style which make intensive use of the heap. In Sect. 4.1 and Sect. 4.2, we briefly present the notions of recursive representation and calls-to size-relation in a rather informal style. Then, we introduce our cost model for heap consumption and our notion of heap space cost relation in Sect. 4.3.

4.1 Recursive Representation

Cost relations can be elegantly expressed as systems of *recursive* equations. In order to automatically generate them, we need to capture the iterative behaviour of the program by means of recursion. One way of achieving this is by computing the CFG of the program. Also, advanced features like virtual invocation and exceptions are simply dealt as additional nodes in the graph. To analyze the bytecode, its CFG can be represented by using some auxiliary recursive representation (see, e.g., [2]). In this approach, a *bytecode* is transformed into a set of *guarded rules* of the form $\langle head \leftarrow guard, body \rangle$ where the *guard* states the applicability conditions for the rule. Rules are obtained from blocks in the CFG and *guards* indicate the conditions under which each block is executed. As it is customary in deterministic imperative languages, guards provide mutually exclusive conditions because paths from a block are always exclusive (i.e., alternative) choices.

DEFINITION 4.1 (rec. representation). *Consider a block* $\mathtt{p}$ *in a CFG, which contains a sequence of bytecode instructions* $\mathtt{B}$ *guarded by the condition* $\mathtt{G_b}$ *and whose successor blocks are* $\mathtt{q_1}, \cdots, \mathtt{q_n}$. *The* recursive representation *of* $\mathtt{p}$ *is:*

$$\mathtt{p}(\bar{\mathtt{l}}, \bar{\mathtt{s}}, \mathtt{r}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}); \cdots ; \mathtt{q_n}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}))$$

where:

- $\bar{l}$ *is a tuple of variables which corresponds to the method's local variables,*
- $\bar{s}$ *and* $\bar{s}'$ *are tuples of variables which respectively correspond to the active stack elements at the block's entry and exit,*
- r *is a single variable which corresponds to the method's return value (omitted if there is not return value),*
- G_p *and* B *are obtained from the block's guard and bytecode instructions by adding the local variables and stack elements on which they operate as explicit arguments.*

We denote by $\mathtt{calls}(\mathtt{B})$ *the set of method invocation instructions within* $\mathtt{B}$ *and by* $\mathtt{bytecode}(\mathtt{B})$ *the other instructions.* □

The formal translation of bytecode instructions in `B` to calls within the recursive rules is presented in [2]. In this translation, it is interesting to note that the stack positions are visible in the rules by explicitly defining them as local variables. This intermediate representation is convenient for analysis as in one pass we can eliminate almost all stack variables which results in a more efficient analysis.

EXAMPLE 4.2. *The rules that correspond to the blocks* $Block_0^{Cons}$, $Block_1^{Cons}$ *and* $Block_2^{Cons}$ *in Fig. 1 are:*

$$\begin{array}{l}
\mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{r}) \leftarrow \\
\quad \mathtt{new}(\mathtt{Cons}, \mathtt{s}_0), \mathtt{dup}(\mathtt{s}_0, \mathtt{s}_1), \mathtt{Cons.<init>}(\mathtt{s}_1), \\
\quad \mathtt{astore}(\mathtt{s}_0, \mathtt{aux}'), \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0'), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'), \\
\quad \mathtt{getfield}(\mathtt{Cons.elem}, \mathtt{s}_1', \mathtt{s}_1''), \\
\quad \mathtt{putfield}(\mathtt{Cons.elem}, \mathtt{s}_0', \mathtt{s}_1''), \\
\quad \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0''), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'''), \\
\quad \mathtt{getfield}(\mathtt{Cons.next}, \mathtt{s}_1''''), \\
\quad (\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}) ; \\
\qquad \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r})).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Nil})), \\
\quad \mathtt{Nil.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Cons})), \\
\quad \mathtt{Cons.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

The rule $\mathtt{copy}_0^{\mathtt{Cons}}$ *is not guarded and has two continuation blocks, while the other rules are guarded by the type of*

the object of s_1 (the top of the stack) and have only one successor. The bytecode instructions were transformed to include explicitly the stacks elements and the local variables on which they operate, moreover, all variables are in single static assignment *form. Note that calls to methods take the same form as calls to blocks, which makes all different forms of loops to fit in the same setting.* □

4.2 Size Analysis

A size analysis is then performed on the recursive representation in order to infer the *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) which occur in the body for each program rule. Derivation of constraints is a standard abstract interpretation over a constraints domain such as Polyhedra [2, 3]. Such relations are essential for defining the cost of one block in terms of the cost of its successors. The analysis is done by abstracting the bytecode instructions into the linear constraints they impose on their arguments, and then computing a fixpoint that collects *calls-to* relations.

DEFINITION 4.3 (calls-to size-relations). *Consider the rule in Def. 4.1, its calls-to* size-relations *are triples of the form*

$\langle \mathtt{p}(\bar{\mathtt{x}}), \mathtt{p}'(\bar{\mathtt{z}}), \varphi \rangle$ *where* $\mathtt{p}'(\bar{\mathtt{z}}) \in \mathtt{calls(B)} \cup \mathtt{q_1}(\bar{\mathtt{y}}) \cup \ldots \cup \mathtt{q_n}(\bar{\mathtt{y}})$

The size-relation φ *is given as a conjunction of linear constraints. The tuples of variables $\bar{x}$, $\bar{y}$ and $\bar{z}$ correspond to the variables of the corresponding block.* □

In Java bytecode, we consider three cases within size relations: for integer variables, *size-relations* are constraints on the possible values of variables; for reference variables, they are constraints on the length of the longest reachable paths [21], and for arrays they are constraints on the length of the array. Note that using the path-length notion cyclic structures are not handled since to guarantee soundness the corresponding references are abstracted to "unknown-length" and therefore cost that depends on them cannot be inferred.

EXAMPLE 4.4. *The calls-to-size relation for the first rule in Ex. 4.2 is formed by the triples:*

$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$
$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$

where φ includes, among others, the constraint $this{=}s_1''''{+}1$ which states that the list that $this$ points to is longer by one than the list that s_1'''' points to (s_1'''' corresponds to $this.next$). The meaning of the above relations is explained in Section 2. Note that the call to the constructor Cons.<init> *is ignored for simplicity.* □

4.3 Heap Space Cost Relations

In order to define our heap space cost analysis, we start by defining a cost model which defines the cost of memory allocation instructions (e.g., new, newarray and anewarray) as the the number of heap (memory) units they consume. The remaining bytecode instructions do not add any cost.

DEFINITION 4.5 (cost model for heap space). *We define a cost model $\mathcal{M}_{heap}$ which takes a bytecode instruction* bc *and returns a positive expression as follows:*

$$\mathcal{M}_{heap}(\mathtt{bc}) = \begin{cases} size(\mathtt{Class}) & \text{if } bc\text{=}\mathtt{new(Class,\_)} \\ S_{\mathtt{PrimType}} * \mathtt{L} & \text{if } bc\text{=}\mathtt{newarray(PrimType,L,\_)} \\ S_{\mathtt{ref}} * \mathtt{L} & \text{if } bc\text{=}\mathtt{anewarray(Class,L,\_)} \\ 0 & \text{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ denote, respectively, the heap consumption of primitive types and references. Function size is defined as follows:

$$size(O) = \begin{cases} \sum_{F \in \mathtt{Class.field}} size(type(F)) & \text{if } O\text{=}\mathtt{Class} \\ S_{\mathtt{PrimType}} & \text{if } O \text{ is a primitive type} \\ S_{\mathtt{ref}} & \text{if } O \text{ is a reference type} \end{cases}$$

where the type of a field in a Class *(i.e.,* Class.field*) can be either primitive or reference.* □

In Java bytecode, types are classified into primitive (its size is represented by $S_{\mathtt{PrimType}}$ in our model) and reference types ($S_{\mathtt{ref}}$). In a particular assessment, one has to set the concrete values for $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ of the JVM implementation.

For each rule in the recursive representation of the program and its corresponding size relation, the analysis generates the cost equations which define the heap consumption of executing the block (or possibly a method call) by relying on the above cost model. A *heap space cost relation* is defined as the set of cost equations for each block of the bytecode (or rule in the recursive representation).

DEFINITION 4.6 (heap space cost relation). *Consider a rule $\mathcal{R}$ of the form $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{y}}); \cdots; \mathtt{q_n}(\bar{\mathtt{y}}))$ and let the linear constraints φ be a conjunction of all call-to size-relations within the rule. The* heap space cost *equations for $\mathcal{R}$ are generated as follows:*

$$C_p(\bar{x}) = \sum_{b \in bytecode(B)} \mathcal{M}_{heap}(b) + \sum_{r(\bar{z}) \in calls(B)} + C_r(\bar{z}) + C_{p\_cont}(\bar{y}) \quad \varphi$$
$$C_{p\_cont}(\bar{y}) = C_{q_1}(\bar{y}) \quad G_{q_1}$$
$$\ldots$$
$$C_{p\_cont}(\bar{y}) = C_{q_n}(\bar{y}) \quad G_{q_n}$$

where $\mathtt{G_{q_i}}$ is the guard of $\mathtt{q_i}$. The heap space cost relation associated to the recursive representation of a method is defined as the set of cost equations for its blocks. □

When the rule has multiple *continuation*s, it is transformed into several equations. We specify the cost of each continuation in a separate equation because the guards for determining the alternative path $\mathtt{q_i}$ that the execution will take (with $i = 1, \ldots, n$) are only known at the end of the execution of the bytecode B; thus, they cannot be evaluated before B is executed. The guards appear also decorating the equations. In the implementation, when a rule has only one continuation, it gives rise to a single equation which contains the size relation φ as an attachment.

Java source code

```
abstract class Data{
  abstract public Data copy();
}
class Polynomial extends Data{
  private int deg;
  private int[] coefs;
  public Polynomial() {
    coefs = new int[11];}
  public Data copy() {
    Polynomial aux = new Polynomial();
    aux.deg = deg;
    for (int i=0;i<=deg && i<=10;i++)
      aux.coefs[i] = coefs[i];
    return aux;}}
```

```
class Vector3D extends Data{
  private int x;
  private int y;
  private int z;
  public Vector3D(int x,int y,int z) {
    this.x = x;
    this.y = y;
    this.z = z;}
  public Data copy() {
    return new Vector3D(x,y,z);
  }
}
```

```
class Results{
  Data[] rs;
  public Results() {
    rs = new Data[25];
  }
```

```
public Results copy() {
  Results aux = new Results();
  for (int i = 0;i < 25;i++)
    aux.rs[i] = rs[i].copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = S_{ref} + 25{*}S_{ref} + C_0(a,0)$		
$C_0(a,i) = \underbrace{S_{int} + S_{ref}}_{size(Polyn)} + 11{*}S_{int} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Polyn \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = \underbrace{3{*}S_{int}}_{size(Vect3D)} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Vect3D \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = 0$		$\{i{>}{=}25\}$

Figure 2. Constant heap space example

EXAMPLE 4.7. *The heap space cost equations generated for the rule* $copy_0^{Cons}$ *of Ex. 4.2 and the size relation of Ex. 4.4 are (see Sect. 2):*

$$C_0^{Cons}(this, aux) = size(Cons) + CC_0^{Cons}(this, aux', s_0'', s_1'''')\{this = s_1'''' + 1, \ldots\}$$

$$CC_0^{Cons}(this, aux', s_0'', s_1'''') = \begin{cases} C_1^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Nil \\ C_2^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Cons \end{cases}$$

The cost of $Block_0^{Cons}$ *is captured by* C_0^{Cons}*, among all bytecode instructions in* $Block_0^{Cons}$*, we count only the creation of the object of class* `Cons`*. The continuation of* $Block_0^{Cons}$ *is captured in the relation* CC_0^{Cons}*, where depending on the type of the object* s_1''''*, we choose between two* mutually exclusive *equations* C_1^{Cons} *or* C_2^{Cons}*.* □

In addition, the analyzer performs a slicing step, which aims at removing variables that do not affect the cost. And also tries to simplify the equations as much as possible by applying unfolding steps. These steps lead to simpler cost relations. Due to lack of space, during the rest of the paper we will apply them without giving details on how they were performed.

5. Example Applications of Heap Space Analysis

In this section, we show the most salient features of our heap space analysis by means of a series of examples. All examples are written in object-oriented style and make intensive use of the heap. We intend to illustrate how our analysis is able to deal with standard data-structures like lists, trees and arrays with several dimensions as well as with multiple inheritance, class fields, virtual invocation, etc. We show examples which present heap usage which depends proportionally to the *data size*, namely in some cases it depends on class fields while in another one on the input arguments. An interesting point is that heap consumption is, in the different examples, constant, linear, polynomial or exponentially proportional to the data sizes.

For each example, we show the Java source code and its heap space cost relation. Each relation consists of three parts: the equations, the guards and the size relations. The applicability conditions of each equation are defined by the guards and the size relations. Guards usually provide non-numeric conditions while size relations provide conditions on the sizes of the corresponding variables. In addition, size relations describe how the data changes when the control moves from one to another part of the program. Since our

Java source code

```
abstract class List{
  abstract public List copy();
}
class Nil extends List{
  public List copy() {
    return this;
}
```

```
class Cons extends List
  private Data elem;
  private List next;
  public List copy() {
    Cons aux = new Cons();
    aux.elem = this.elem.copy();
    aux.next = this.next.copy();
    return aux;}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = \overbrace{2{*}S_{ref}}^{size(Cons)} + \overbrace{S_{int}+S_{ref}+11{*}S_{int}}^{this.elem.copy()}$	$\left\langle \begin{array}{l} \hat{a}.elem \in Polyn \wedge \\ \hat{a}.next \in Nil \end{array} \right\rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+S_{int}+S_{ref} +11{*}S_{int}+C_{copy}(b)$	$\left\langle \begin{array}{l} \hat{a}.elem \in Polyn \wedge \\ \hat{a}.next \in Cons \end{array} \right\rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$
$C_{copy}(a) = \underbrace{2{*}S_{ref}}_{size(Cons)} + \underbrace{3{*}S_{int}}_{this.elem.copy()}$	$\left\langle \begin{array}{l} \hat{a}.elem \in Vect3D \wedge \\ \hat{a}.next \in Nil \end{array} \right\rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+3{*}S_{int}+C_{copy}(b)$	$\left\langle \begin{array}{l} \hat{a}.elem \in Vect3D \wedge \\ \hat{a}.next \in Cons \end{array} \right\rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$

Figure 3. Generic list example

system only deals with integer primitive types, we use the cost model presented in Sect. 4.3 with the constants S_{int} and S_{ref} to denote the basic sizes for integers and reference types, respectively. Also note that we provide the Java source code instead of the bytecode just for clarity and space limitations. The analyzer works directly on the bytecode which can be found in the appendix.

5.1 Constant Heap Space Usage

In the first example we consider a method with constant heap space usage, i.e., its heap consumption does not depend on any input argument. Fig. 2 shows both the source code and the heap space cost equations generated by the analyzer. The program implements a data hierarchy which will be used throughout the section. It consists of an abstract class, *Data* and two subclasses, *Polynomial* and *Vector3D*. The class *Polynomial* defines a polynomial expression of degree up to 10 with integer coefficients, the coefficients are stored in the array field *coefs* and the degree in the integer field *deg*. Its *copy* method returns a deep copy of the corresponding polynomial by creating a new array of 11 integers and copying the first $deg{+}1$ original coefficients. The class *Vector3D* represents an integer vector with 3 dimensions. The class *Results* stores 25 objects of type *Data*, which in execution time will be *Polynomial* or *Vector3D* objects. Its *copy* method produces a deep copy of the whole structure where each of the 25 elements is copied by its corresponding *copy* method (hence dynamically resolved).

The cost equations generated by the analyzer for the method *Results.copy* are shown in Fig. 2 (at the bottom left). The first equation $C_{copy}(a)$ defines the heap consumption of the method in terms of its first argument a which corresponds to the abstraction of its *this* reference variable (i.e., its size). It counts the heap space allocated for the creation of an object of type *Results*, namely S_{ref}; the space allocated by its constructor, namely $25{*}S_{ref}$; and the space allocated when executing the loop. The heap space allocated by the loop is captured by C_0 and it depends on the type of the object at the current position of the array (which is specified in the guards by checking the class of $\hat{a}.rs[i]$) such that the call to its corresponding *copy* method contributes $S_{int}+S_{ref}{+}11{*}S_{int}$ if it is an instance of *Polynomial* and $3{*}S_{int}$ if it is an instance of *Vector3D*.

As already mentioned, a further issue is how to automatically infer *closed form* solutions (i.e., without recurrences) from the generated cost relations. In our examples, we can directly apply the method of [1] to compute an upper bound in closed form. However, we will not go into details of this process as it is not a concern of this paper and we will simply show the asymptotic complexity that can be directly obtained from such upper bounds. We can observe from the equations that the asymptotic complexity is $O(1)$, as equation C_{copy} is a constant plus C_0, and C_0 is called a constant number of times (in this case 25 times). By assuming that $S_{int}=4$ and $S_{ref}=4$, we can obtain the following upper bound $C_{copy}=4+25*4+25*52=1404$.

5.2 Bounds Proportional to the Input Data Size

For the second example, we consider a generic data structure of type *List*. Both the source code and the heap space cost equations obtained by our analyzer are depicted in Fig. 3. The list is implemented taking advantage of the polymorphism as in the style of the example in Sect. 1, but in this case the elements of the list are objects extending from *Data*

Java source code

```
class Score{
  private int gt1, gt2;
  public Score() {
    gt1 = 0;
    gt2 = 0;
  }
class Scoreboard{
  private Score[][][] scores;
public Scoreboard(int a,int b) {
  scores = new Score[a][][];
  for (int i = 1;i <= a;i++) {
    scores[i-1] = new Score[i][];
    for (int j = 0;j < (i-1);j++) {
      scores[i-1][j] = new Score[b];
      for (int k = 0;k < b;k++)
        scores[i-1][j][k] = new Score();}}}}
```

Heap space cost equations

Equation			Guard	Size rels.
$C_{<init>}(a,b)$	=	$a{*}S_{ref} + C_1(a,b,1)$		
$C_1(a,b,i)$	=	$i{*}S_{ref} + C_2(b,i,0) + C_1(a,b,d)$		$\{i{\leq}a,\ d{=}i{+}1\}$
$C_1(a,b,i)$	=	0		$\{i > a\}$
$C_2(b,i,j)$	=	$b{*}S_{ref} + C_3(b,0) + C_2(b,i,d)$		$\{j{<}(i{-}1),\ d{=}j{+}1\}$
$C_2(b,i,j)$	=	0		$\{j \geq (i{-}1)\}$
$C_3(b,k)$	=	$2{*}S_{int} + C_3(b,c)$		$\{k{<}b,\ c{=}k{+}1\}$
$C_3(b,k)$	=	0		$\{k \geq b\}$

Figure 4. Multi-dimensional arrays example

(see the classes in Fig. 2) rather than integer primitive types. The *List.copy* method returns a deep copy of the list which, in addition to copying the whole list structure, it copies each element by using the corresponding *Data.copy* method (resolved at execution time).

At the bottom of Fig. 3, we show the heap space cost equations our analyzer generates for the method *List.copy* of class *Cons*. The equation $C_{copy}(a)$ defines the heap consumption of the whole method in terms of its first argument a which represents the size of its *this* reference variable. There are four equations for C_{copy}, two of them (the second and the fourth one) are recursive and correspond to the case in which the rest of the list is not empty, i.e., $\hat{a}.next \in Cons$. Note that, in such recursive equations, the size analysis is able to infer the constraint $a > b$, thus ensuring that recursive calls are made with a strictly decreasing value. The other two equations are constant and correspond to the base case (i.e. the rest of the list is empty). This is abstracted in the size relations with the constraint $a = 2$. Note that the heap usage depends on whether we invoke the *copy* method of a *Polynomial* or a *Vector3D* object. By considering the worst cases for all equations, we can infer the upper bound $C_{copy}(a){\leq}(5{*}S_{ref}{+}15{*}S_{int}){*}a \equiv O(a)$ which describes a heap consumption linear in a, the size of the list.

5.3 Multi-Dimensional Arrays

Let us consider the example in Fig. 4. The class *Scoreboard* is instrumental to show how our heap space analysis deals with complex multi-dimensional array creation. The class has a 3-dimensional array field. The constructor takes two integers a and b and creates an array such that: the first dimension is a; the second dimension ranges from 1 to a; and the third dimension is b. Each array entry $scores[i][j][k]$ stores an object of type *Score*.

At the bottom of Fig. 4 we can see the heap space cost equations generated by the analyzer for the constructor of class *Scoreboard*. The equation $C_{<init>}(a,b)$ represents the heap space consumption of the constructor where a and b correspond to the size of its input parameters. It counts the heap consumed by constructing the first array dimension, $a{*}S_{ref}$, plus the heap consumption when executing the outermost loop which is represented by the call $C_1(a,b,1)$. The heap consumption modeled by C_1 includes the amount of heap allocated for the second array dimension in each iteration, $i{*}S_{ref}$, and the consumption of executing the middle loop which is represented by the call $C_2(b,i,0)$. Note that size analysis infers that within C_1, the value of i increases by 1 at each iteration ($d{=}i{+}1$) until it converges to a ($i{\leq}a$). The equation C_2 defines the heap consumption of the middle loop, which includes the heap allocated for the third array dimension, $b{*}S_{ref}$, plus the consumption of executing the innermost loop which is represented by the call $C_3(b,0)$. Finally, C_3 models the heap space required for creating $b{-}k$ *Score* objects by the innermost loop. In this case, we can infer the upper bound $C_{<init>}(a,b) \leq (((2{*}S_{int}{*}b) + b{*}S_{ref}){*}a + a{*}S_{ref}){*}a + a{*}S_{ref} \equiv O(b{*}a^2)$.

5.4 Complex Data Structures

For the last example, let us consider a more complex tree-like data structure which is depicted in Fig. 5. The class *MultiBST* implements a binary search tree data structure where each node has an object of type List (from Fig. 3) and two successors of type *MultiBST* which correspond to the right and left branches of the tree. The constructor method creates an empty tree whose *data* field is initialized to an empty list, i.e., an instance of class *Nil*. The *copy* method performs a deep copy of the whole tree by relying on the *copy* method of class *List*.

Java source code

```
class BST {
  private List data;
  private MultiBST lc;
  private MultiBST rc;
  public MultiBST() {
    data = new Nil();
    lc = null; rc = null; }

public MultiBST copy() {
  MultiBST aux = new MultiBST();
  aux.data = data.copy();
  if (l==null) aux.lc=null; else aux.lc=lc.copy();
  if (r==null) aux.rc=null; else aux.rc=rc.copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C(a) = 3{*}S_{ref} + D(d)$	$\langle \hat{a}.lc = null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d, a{>}l\}$
$C(a) = 3{*}S_{ref} + D(d) + C(r)$	$\langle \hat{a}.lc = null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}r\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l) + C(r)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}l, a{>}r\}$

Figure 5. Multi binary search tree example

The heap space cost equations generated by the analyzer for the method *MultiBST.copy* are depicted in Fig. 5 (at the bottom). The equations defining $C(a)$ represent the heap space usage of the whole method in terms of the parameter a, which corresponds to the maximal path-length in the tree. There are four cases which correspond to the different possible values for the left and right branches (equal or different from null). Consider for example the last equation: $3 * S_{ref}$ is the heap allocated by the new instruction; $D(d)$ is the heap consumption for copying an object of type *List* which corresponds to C_{copy} from Fig. 3; $C(l)$ and $C(r)$ correspond to the heap consumption of copying the left and right branches respectively. From the cost relation, we infer the upper bound $C(a){\leq}(3{*}S_{ref}{+}D(a)){*}2^a$ where $D(a)$ corresponds to the cost of copying the *data* field (see Sec. 5.2).

6. Active Heap Space with Garbage Collection

One of the safety principles in the Java language is ensured by the use of a garbage collector which avoids errors by the programmer related to deallocation of objects from the heap. The aim of this section is to furnish the heap usage cost relations with *safe annotations* which mark the heap space that will be deallocated by the garbage collector upon exit from the corresponding method. The annotations are then used to infer heap space upper bounds for methods upon exit.

In order to generate such annotations, we rely on the use of *escape analysis* (see, e.g., [8, 15]). Essentially, we assume that the heap allocation instructions `new`, `newarray` and `anewarray` have been respectively transformed by new instructions `new_gc`, `newarray_gc` and `anewarray_gc` as long as it is guaranteed that the lifetime of the corresponding allocated heap space does not exceed the instruction's static scope. In this case, the heap space can be safely deallocated upon exit from the corresponding method. This preprocessing transformation can be done in a straightforward way by using the information inferred by escape analysis. In the following, we refer by *transformed* bytecode instructions to the above transformation performed on the heap allocation instructions. Also, we use $gc(H)$ to denote that the heap space H will safely be garbage collected upon exit from the corresponding method (according to escape analysis) and $ngc(H)$ to denote that it might not be garbage collected.

DEFINITION 6.1. *We define a cost model for heap space with garbage collection which takes a* transformed *bytecode instruction* bc *and returns a positive symbolic expression as:*

$$\mathcal{M}^{gc}_{heap}(\mathtt{bc}) = \begin{cases} ngc(size(\mathtt{Class})) & \textit{if bc=}\mathtt{new(Class,\_)} \\ gc(size(\mathtt{Class})) & \textit{if bc=}\mathtt{new\_gc(Class,\_)} \\ ngc(S_{\mathtt{PrimType}} * \mathtt{L}) & \textit{if bc=}\mathtt{newarray(PrimType,L,\_)} \\ gc(S_{\mathtt{PrimType}} * \mathtt{L}) & \textit{if bc=}\mathtt{newarray\_gc(PrimType,L,\_)} \\ ngc(S_{\mathtt{ref}} * \mathtt{L}) & \textit{if bc=}\mathtt{anewarray(Class,L,\_)} \\ gc(S_{\mathtt{ref}} * \mathtt{L}) & \textit{if bc=}\mathtt{anewarray\_gc(Class,L,\_)} \\ 0 & \textit{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$, $S_{\mathtt{ref}}$ *and* $size()$ *are as in Def. 4.5.* □

The above cost model returns a *symbolic* positive expression which contains the annotations gc and ngc as described above. Therefore, when generating the heap space cost relations as described in Def. 4.6 w.r.t. $\mathcal{M}^{gc}_{heap}$, the cost relations will be of the following form:

$$C(\bar{x}) = gc(H_{gc}) + ngc(H_{ngc}) + \sum C_r(\bar{z}) \quad \varphi$$

where we assume that all symbolic expressions wrapped by gc (resp. by ngc) are grouped together within each cost equation and denote the total heap space H_{gc} that will be garbage collected (resp. H_{ngc} which might not be) after the application of such equation.

EXAMPLE 6.2. *Suppose we add the following methods*

```
abstract List map(Func o);                  // List
List map(Func o) { return this; }           // Nil
List map(Func o) {                          // Cons
    List tail = this.next.map(o);
    Cons head = new Cons();
    head.next = tail;
    head.elem = o.f(new Integer(this.elem));
    return head;
}
```

respectively to the classes List*,* Nil *and* Cons *which are depicted in Fig. 1. The method* map *clones the corresponding list structure, but the value of the field* elem *in the clone is the result of applying the method* o.f *on the corresponding value in the cloned list. Note that the method* o.f*, takes as input an object of type* Integer*, therefore* this.elem *(which is of type* int*) is first converted to* Integer *by creating a temporary corresponding* Integer *object. For simplicity, we do not give a specific definition for* Func*, but we assume that its method f (which is called using* o.f*) does not allocate any heap space and that it returns a value of type* int*. Using* escape analysis*, the creation of the temporary* Integer *object can be annotated as local to* map*, therefore we replace the corresponding* `new` *instruction by* `new_gc`*. Assuming that the size of an* Integer *object is 4 bytes, and using the cost model of Def. 6.1, we obtain the following cost equations:*

Equation	*Size relations*
$C^{Nil}_{map}(a) = 0$	$\{a{=}1\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8)$	$\{a{=}2\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8) + C^{Cons}_{map}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

The symbolic expression $gc(4)$ in the above equations corresponds to the heap space allocated for the temporary Integer *object which can be garbage collected upon exit from* map*, and $ngc(8)$ corresponds to the heap space allocated for the* Cons *object. As before,* a *corresponds to the size of the* this *reference variable (i.e., the list length) and* b *to* this.next*.* □

Using the refined cost relations we can infer different information about the heap space usage depending on the interpretation given to the gc and ngc annotations. Let us first consider the following definitions:

$$\boxed{\forall\, H,\, gc(H) = 0 \text{ and } ngc(H) = H} \qquad (1)$$

where we do not count the heap space that will be deallocated upon exit from the corresponding method. By applying Eq. (1) to a cost relation C_m of a method m, we can infer an upper bound U^{gc}_m of the *active heap space* upon the exit from m, i.e., the heap space consumed by m which might not be deallocated upon exit. In this setting, for the cost relations of Ex. 6.2 we infer the closed form $U^{gc}_{map} \equiv C^{Cons}_{map}(a) = 8 * (a-1)$. It is important to note that, in general, such upper bound does not ensure that the heap space required for executing m does not exceed U^{gc}_m, i.e., it is not an upper bound of the heap usage *during* the execution of m but rather only *after* its execution. Actually, in this simple example, we can observe already that during the execution of the method *map*, if all objects are heap allocated, we need more than $8 * (a-1)$ heap units (as the objects of type *Integer* will be heap allocated and they are not accounted in the upper bound). However, one of the applications of escape analysis is to determine which objects can be stack allocated instead of heap allocated in order to avoid invoking the garbage collector which is time consuming [8]. For instance, in the above example, the objects of type *Integer* can be safely stack allocated. When this stack allocation optimization is performed, then U^{gc}_m is indeed an upper bound for the heap space required to execute m.

In order to infer upper bounds for the heap space required during the execution of m, we define gc and ngc as follows:

$$\boxed{\forall\, H,\, gc(H) = H \text{ and } ngc(H) = H} \qquad (2)$$

In this case, we obtain the same cost relations as in Def. 4.6 which correspond to the worst case heap usage in which we do not discount any deallocation by the garbage collector. In this setting, for the cost relation of Ex. 6.2 we infer the closed form $U_{map}(a) \equiv C^{Cons}_{map}(a) = 12 * (a-1)$.

Analysis for finding upper bounds on the memory high-watermark cannot be directly done using cost relations as introducing decrements in the equations requires computing lower bounds. As a further issue, the *active heap space* upper bound, U^{gc}_m, can be used to improve the accuracy of the upper bound on the heap space required for executing a sequence of method calls. For example, an upper bound of the heap space required for executing a method m_1 and upon its return immediately executing a method m_2 can be approximated by $max(U_{m_1}, U^{gc}_{m_1} + U_{m_2})$ which is more precise than taking $U_{m_1} + U_{m_2}$ as it takes into account that after executing m_1 we can apply garbage collection and only then executing m_2. This idea is the basis for a post-processing that could be done on the program in order to obtain more accurate upper bounds on the heap usage at a *program point* level. This is a subject of ongoing research.

7. Experiments

In order to assess the practicality of our heap space analysis, we have implemented a prototype inter-procedural analyzer in `Ciao` [10] as an extension of the one in [3]. We still have not incorporated an escape analysis in our implementation and hence the upper bounds inferred correspond to those generated using Eq. (2) of Sect. 6. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9. Table 1 shows the run-times of the different phases of the heap space analysis process. The name of the main class to be analyzed is given in the first column, *Benchmark*, and its size (the sum of all its *class* file sizes) in KBytes is given in the second column, *Size*. Columns 3-6 shows the runtime of the different phases in milliseconds, they are computed as the arithmetic mean of five runs: *RR* is the time for obtaining the recursive representation (building CFG, eliminating stack elements, etc., as outlined in Sec. 4.1); *Size An.* is the time for the abstract-interpretation based size analysis for computing size relations; *Cost* is the time taken for building the heap space cost relations for the different blocks and representing them in a simplified form; and *Total* shows the total times of the whole analysis process. In the last column, *Complexity*, we depict the asymptotic complexity of the (worst-case) heap space cost obtained from the cost relations.

Benchmark	Size	RR	Size An.	Cost	Total	Complexity	
ListInt	0.86	24	53	7	83	$O(n)$	$n \equiv$ list length
Results	1.31	83	275	15	374	$O(1)$	–
BSTInt	0.48	37	113	5	156	$O(2^n)$	$n \equiv$ tree depth
List	1.79	71	207	16	293	$O(n)$	$n \equiv$ list length
Queue	1.93	219	570	24	813	$O(n)$	$n \equiv$ queue length
Stack	1.38	89	643	17	749	$O(n)$	$n \equiv$ stack length
BST	1.43	97	238	14	349	$O(2^n)$	$n \equiv$ tree depth
Scoreboard	0.65	280	1539	12	1830	$O(a^2 * b)$	$\{a, b\} \equiv$ input args.
MultiBST	2.35	166	510	34	709	$O(n * 2^n)$	$n \equiv$ tree depth

Table 1. Measured time (in ms) of the different phases of cost analysis

Regarding the benchmarks we have used, on one hand, we have benchmarks implementing some classic data structures using an object-oriented programming style, which expose the analyzer's ability in handling such classical data structures as well as sophisticated object-oriented programming features. In particular, *ListInt*, *List*, *Queue*, *Stack*, *BSTInt*, *BST* and *MultiBST* implement respectively integer and generic lists, generic queues, generic stacks, integer and generic binary search trees which allow data repetitions. On the other hand, we have some benchmarks which expose more particular issues of heap space analysis, such as *Results* which has constant heap space usage and *Scoreboard* which presents a multidimensional arrays creation. For all benchmarks, we have analyzed the corresponding *copy* method which performs a deep copy of the corresponding structure.

We can observe in the table that computing size relations is the most expensive step as it requires a global analysis of the program, whereas *RR* and *Cost* basically involve a single pass on the code. Our prototype implementation supports the full instructions set of sequential Java bytecode, however, it is still preliminary, and there is plenty of room for optimization, mainly in the size analysis phase, which in addition assumes the absence of cyclic data structures, which can be verified using the non-cyclicity analysis [23].

8. Conclusions and Related Work

We have presented an automatic analysis of heap usage for Java bytecode, based on generating at compile-time cost relations which define the heap space consumption of an input bytecode program. By means of a series of examples which allocate lists, trees, trees of lists, arrays, etc. in the heap, we have shown that our analysis is able to infer non-trivial bounds for them (including polynomial and exponential complexities). We believe that the experiments we have presented show that our analysis improves the state of the practice in heap space analysis of Java bytecode.

Related work in heap space analysis includes advanced techniques developed in functional programming, mainly based on type systems with resource annotations (see, e.g., [24, 17, 25, 19]) and, hence, they are quite different technically to ours. But heap space analysis is comparatively less developed for low-level languages such as Java bytecode. A notable exception is the work in [11], where a memory consumption analysis is presented. In contrast to ours, their aim is to verify that the program executes in bounded memory by simply checking that the program does not create new objects inside loops, but they do not infer bounds as our analysis does. Moreover, it is straightforward to check that new objects are not created inside loops from our cost relations. Another related work includes research in the MRG project [5, 7], which focuses on building a proof-carrying code [22] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The analysis is developed for a functional language which then compiles to a (subset of) Java bytecode and it is restricted to linear bounds. In [6] the Bytecode Specification Language is used to annotate Java bytecode programs with memory consumption behaviour and policies, and then verification tools are used to verify those policies.

For Java-like languages, the work of [18] presents a type system for heap analysis without garbage collection, it is developed at the level of the source code and based on amortised analysis (hence it is technically quite different to our work) and, unlike us, they do not present an inference method for heap consumption. On the other hand, the work of [9] deals also with Java source code, it is able to infer polynomial complexity though it does not handle recursion.

Some works consider explicit deallocation of objects by decreasing the cost by the size of the deallocated object (see, e.g., [18, 17]). This approach is interesting when one wants to observe the heap consumption at certain program points. However, it cannot be directly incorporated in our cost relations because they are intended to provide a *global* upper bound of a method's execution. Naturally, it should happen that allocated objects are correctly deallocated and hence our cost relations would provide zero as (global) upper bound. Other work which considers cost with garbage collection is [24]. Unlike ours, it is developed for pure functional programs where the garbage collection behaviour is easier to predict as programs do not have assignments.

In the future, we want to extend our work in several directions. On the practical side, we want to incorporate an escape

analysis to transform the bytecode as outlined in Sect. 6. Regarding scalability, it is a question of performance vs. precision trade-off and depends much on the underlying abstract domain used by the size analysis. We believe our analysis would scale without sacrificing precision if an efficient domain like octagons is used together with [1]. On the theoretical side, we plan to adapt our analysis to infer upper bounds on the heap usage at given program points in the presence of garbage collection. We also would like to develop an analysis which infers upper bounds on the call stack usage.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Cost Equation Systems. *Submitted*, 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *Proc. of BYTECODE'07*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[4] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.

[5] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, number 3362 in LNCS. Springer, 2005.

[6] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[7] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Proc. of LPAR'04*, LNCS 3452, pages 347–362. Springer, 2004.

[8] Bruno Blanchet. Escape Analysis for Java$^{\text{tm}}$: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

[9] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[11] D. Cachera, D. Pichardie T. Jensen, and G. Schneider. Certified memory usage analysis. In *FM'05*, number 3582 in LNCS. Springer, 2005.

[12] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2005.

[13] W. Chin, H. Nguyen, S. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, LNCS 3672, pages 70–86. Springer, 2005.

[14] K. Crary and S. Weirich. Resource bound certification. In *Proc. of POPL'00*, pages 184–198. ACM Press, 2000.

[15] Patricia M. Hill and Fausto Spoto. Deriving Escape Analysis by Abstract Interpretation. *Higher-Order and Symbolic Computation*, (19):415–463, 2006.

[16] M. Hofmann. Certification of Memory Usage. In *Theoretical Computer Science, 8th Italian Conference, ICTCS*, volume 2841 of *Lecture Notes in Computer Science*, page 21. Springer, 2003.

[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM Press, 2003.

[18] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[19] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[21] Patricia M.Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*, 2006.

[22] G. Necula. Proof-Carrying Code. In *POPL'97*. ACM Press, 1997.

[23] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of the 7th workshop on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006. Springer-Verlag.

[24] L. Unnikrishnan, S. Stoller, and Y. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, Lecture Notes in Computer Science, pages 70–85. Springer, 2003.

[25] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

Live Heap Space Analysis for Languages with Garbage Collection

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

Samir Genaim
Complutense University of Madrid
samir.genaim@fdi.ucm.es

Miguel Gómez-Zamalloa
Complutense University of Madrid
mzamalloa@fdi.ucm.es

Abstract

The *peak heap consumption* of a program is the maximum size of the *live* data on the heap during the execution of the program, i.e., the minimum amount of heap space needed to run the program without exhausting the memory. It is well-known that garbage collection (GC) makes the problem of predicting the memory required to run a program difficult. This paper presents, the best of our knowledge, the first *live heap space* analysis for garbage-collected languages which infers accurate upper bounds on the peak heap usage of a program's execution that are not restricted to any complexity class, i.e., we can infer exponential, logarithmic, polynomial, etc., bounds. Our analysis is developed for an (sequential) object-oriented bytecode language with a *scoped-memory* manager that reclaims unreachable memory when methods return. We also show how our analysis can accommodate other GC schemes which are closer to the *ideal* GC which collects objects as soon as they become unreachable. The practicality of our approach is experimentally evaluated on a prototype implementation. We demonstrate that it is fully automatic, reasonably accurate and efficient by inferring live heap space bounds for a standardized set of benchmarks, the JOlden suite.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Live Heap Space Analysis, Peak Memory Consumption, Low-level Languages, Java Bytecode

1. Introduction

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. It is widely recognized that memory usage estimation is important for an accurate prediction of running time, as cache misses and page faults contribute directly to the runtime. Another motivation is to configure real-time garbage collectors to avoid mutator starvation. Besides, upper bounds on the memory requirement of programs have been proposed for resource-bound certification [10] where certificates encode security properties involving resource usage requirements, e.g., the (untrusted) code must adhere to specific bounds on its memory usage. On the other hand, automatic memory management (also known as garbage collection) is a very powerful and useful mechanism which is increasingly used in high-level languages such as Java. Unfortunately, GC makes the problem of predicting the memory required to run a program difficult.

A first approximation to this problem is to infer the *total memory allocation*, i.e., the *accumulated* amount of memory allocated by a program ignoring GC. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, it is an overly pessimistic estimation of the actual memory requirement. *Live heap space analysis* [18, 5, 8] aims at approximating the size of the *live* data on the heap during a program's execution, which provides a much tighter estimation. This paper presents a general approach for inferring the *peak heap consumption* of a program's execution, i.e., the maximum of the live heap usage along its execution. Our live heap space analysis is developed for (an intermediate representation of) an object-oriented *bytecode* language with automatic memory management. Programming languages which are compiled to bytecode and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET.

Analysis of live heap usage is different from total memory allocation because it involves reasoning on the memory consumed at *all program states* along an execution, while total allocation needs to observe the consumption at the *final* state only. As a consequence, the classical approach to static cost analysis proposed by Wegbreit in 1975 [20] has been applied only to infer total allocation. Intuitively, given a program, this approach produces a *cost relation system* (CR for short) which is a set of recursive equations that capture the cost *accumulated* along the program's execution. Symbolic closed-form solutions (i.e., without recursion) are found then from the CR. This approach leads to very accurate cost bounds as it is not limited to any complexity class (infers polynomial, logarithmic, exponential consumption, etc.) and, besides, it can be used to infer different notions of resources (total memory allocation, number of executed instructions, number of calls to specific methods, etc.). Unfortunately, it is not suitable to infer peak heap consumption because it is not an accumulative resource of a program's execution as CR capture. Instead, it requires to reason on all possible states to obtain their maximum. By relying on different techniques which do not generate CR, live heap space analysis is currently restricted to polynomial bounds and non-recursive methods [5] or to linear bounds dealing with recursion [8].

Inspired by the basic techniques used in cost analysis, in this paper, we present a general framework to infer accurate bounds on the peak heap consumption of programs which improves the state-of-the-art in that it is not restricted to any complexity class and deals with all bytecode language features including recursion. To pursue our analysis, we need to characterize the behavior of

ISMM'09, June 19–20, 2009, Dublin, Ireland.

the underlying garbage collector. We assume a standard *scoped-memory* manager that reclaims memory when methods return. In this setting, our main contributions are:

1. *Escaped Memory Analysis.* We first develop an analysis to infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that is allocated during the execution of the method *and* which remains upon exit. The key idea is to infer first an upper bound for the total memory allocation of the method. Then, such bound can be manipulated, by relying on information computed by *escape analysis* [4], to extract from it an upper bound on its escaped memory.
2. *Live Heap Space Analysis.* By relying on the upper bounds on the escaped memory, as our main contribution, we propose a novel form of *peak consumption* CR which captures the peak memory consumption over all program states along the execution for the considered scoped-memory manager. An essential feature of our CRs is that they can be solved by using existing tools for solving standard CRs.
3. *Ideal Garbage Collection.* An interesting, novel feature of our approach is that we can refine the analysis to accommodate other kinds of scope-based managers which are closer to an *ideal* garbage collector which collects objects as soon as they become unreachable.
4. *Implementation.* We report on a prototype implementation which is integrated in the COSTA system [2] and experimentally evaluate it on the JOlden benchmark suite. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way.

2. Bytecode: Syntax and Semantics

Bytecode programs are complicated for both human and automatic analysis because of their unstructured control flow, operand stack, etc. Therefore, it is customary to formalize analyses on intermediate representations of the bytecode (e.g., [3, 19, 13]). We consider a rule-based *procedural* language (in the style of any of the above) in which a *rule-based program* consists of a set of *procedures* and a set of classes. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to the following grammar:

$$
\begin{array}{rl}
\mathit{rule} & ::= p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) ::= g, b_1, \ldots, b_t \\
g & ::= \mathit{true} \mid \mathit{exp}_1 \ \mathit{op} \ \mathit{exp}_2 \mid \mathsf{type}(x, c) \\
b & ::= x := \mathit{exp} \mid x := \mathsf{new}\ c^i \mid x := y.f \mid x.f := y \mid q(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \\
\mathit{exp} & ::= \mathsf{null} \mid \mathit{aexp} \\
\mathit{aexp} & ::= x \mid n \mid \mathit{aexp} - \mathit{aexp} \mid \mathit{aexp} + \mathit{aexp} \mid \mathit{aexp} * \mathit{aexp} \mid \mathit{aexp} / \mathit{aexp} \\
\mathit{op} & ::= > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{array}
$$

where $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_t$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, c)$, which succeeds if the runtime class of x is exactly c. A class c is a finite set of typed field names, where the type can be integer or a class name. The superscript i on a class c is a unique identifier which associates objects with the program points where they have been created. The key features of this language are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be regarded as records, and the behavior induced by dynamic dispatch in the original bytecode program is compiled into *dispatch* blocks

```
class Test {
  static Tree m(int n) {
    if ( n>0 ) return new
      Tree(m(n-1),m(n-1),f(n));
    else return null;
  }
  static int f(int n) {
    int a=0,i=n;
    while ( n>1 ) {
      a += g(n).intValue();
      n=n/2;
    }
    for(; i>1; i=i/2)
      a *= h(i).intValue();
    return a;
  }
  static Integer g(int n) {
    Integer x=new Integer(n);
    return new Integer(x.intValue()+1);
  }
  static Long h(int n) {
    return new Long(n-1);
  }
} // end of class Test

class Tree {
  Tree l,r;
  int d;
  Tree(Tree l,Tree r,int d) {
    this.l = l;
    this.r = r;
    this.d = d;
  }
}
```

Figure 1. Java code of running example

$$
\begin{array}{lll}
(1) & \mathrm{m}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= & \\
 & \quad \mathbf{n > 0}, & \\
 & \quad \mathrm{s_0} := \mathsf{new}\ \mathrm{Tree}^1; & \\
 & \quad \mathrm{s_1} := \mathrm{n} - 1, & \\
 & \quad \mathrm{m}(\langle \mathrm{s_1} \rangle, \langle \mathrm{s_1} \rangle), & \\
 & \quad \mathrm{s_2} := \mathrm{n} - 1, & \\
 & \quad \mathrm{m}(\langle \mathrm{s_2} \rangle, \langle \mathrm{s_2} \rangle), & \\
 & \quad \mathrm{f}(\langle \mathrm{n} \rangle, \langle \mathrm{s_3} \rangle), & \\
 & \quad \mathrm{init}(\langle \mathrm{s_0}, \mathrm{s_1}, \mathrm{s_2}, \mathrm{s_3} \rangle, \langle \rangle), & \\
 & \quad \mathrm{r} = \mathrm{s_0}. &
\end{array}
$$

$$
\begin{array}{lll}
(2) & \mathrm{m}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= & \\
 & \quad \mathbf{n \leq 0}, & \\
 & \quad \mathrm{r} := \mathsf{null}. &
\end{array}
$$

$$
\begin{array}{lll}
(3) & \mathrm{f}(\langle \mathrm{n} \rangle, \langle \mathrm{a} \rangle) ::= & \\
 & \quad \mathrm{a} := 0, & \\
 & \quad \mathrm{i} := \mathrm{n}, & \\
 & \quad \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle), & \\
 & \quad \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle). &
\end{array}
$$

$$
\begin{array}{lll}
(4) & \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle) ::= & \\
 & \quad \mathbf{n > 1}, & \\
 & \quad \mathrm{g}(\langle \mathrm{n} \rangle, \langle \mathrm{s_0} \rangle), & \\
 & \quad \mathrm{intValue_1}(\langle \mathrm{s_0} \rangle, \langle \mathrm{s_0} \rangle) & \\
 & \quad \mathrm{a} := \mathrm{a} + \mathrm{s_0}, & \\
 & \quad \mathrm{n} := \mathrm{n}/2, & \\
 & \quad \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle). &
\end{array}
$$

$$
\begin{array}{lll}
(5) & \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle) ::= & \\
 & \quad \mathbf{n \leq 1}. &
\end{array}
$$

$$
\begin{array}{lll}
(6) & \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle) ::= & \\
 & \quad \mathbf{i > 1}, & \\
 & \quad \mathrm{h}(\langle \mathrm{i} \rangle, \langle \mathrm{s_0} \rangle), & \\
 & \quad \mathrm{intValue_2}(\langle \mathrm{s_0} \rangle, \langle \mathrm{s_0} \rangle) & \\
 & \quad \mathrm{a} := \mathrm{a} * \mathrm{s_0}, & \\
 & \quad \mathrm{i} := \mathrm{i}/2, & \\
 & \quad \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle). &
\end{array}
$$

$$
\begin{array}{lll}
(7) & \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle) ::= & \\
 & \quad \mathbf{i \leq 1}. &
\end{array}
$$

$$
\begin{array}{lll}
(8) & \mathrm{g}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= & \\
 & \quad \mathrm{x} := \mathsf{new}\ \mathrm{Integer}^2, & \\
 & \quad \mathrm{init_1}(\langle \mathrm{x}, \mathrm{n} \rangle, \langle \rangle), & \\
 & \quad \mathrm{intValue_1}(\langle \mathrm{x} \rangle, \langle \mathrm{s_0} \rangle), & \\
 & \quad \mathrm{s_0} := \mathrm{s_0} + 1. & \\
 & \quad \mathrm{r} := \mathsf{new}\ \mathrm{Integer}^3, & \\
 & \quad \mathrm{init_1}(\langle \mathrm{r}, \mathrm{s_0} \rangle, \langle \rangle). &
\end{array}
$$

$$
\begin{array}{lll}
(9) & \mathrm{h}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= & \\
 & \quad \mathrm{s_0} := \mathrm{n} - 1. & \\
 & \quad \mathrm{r} := \mathsf{new}\ \mathrm{Long}^4, & \\
 & \quad \mathrm{init_2}(\langle \mathrm{r}, \mathrm{s_0} \rangle, \langle \rangle). &
\end{array}
$$

$$
\begin{array}{lll}
(10) & \mathrm{init}(\langle \mathrm{this}, \mathrm{l}, \mathrm{r}, \mathrm{d} \rangle, \langle \rangle) ::= & \\
 & \quad \mathrm{this.l} := \mathrm{l}, & \\
 & \quad \mathrm{this.r} := \mathrm{r}, & \\
 & \quad \mathrm{this.d} := \mathrm{d}. &
\end{array}
$$

Figure 2. Intermediate representation of running example.

guarded by a type check, and (5) procedures may have *multiple return* values. The translation from (Java) bytecode to the rule-based form is performed in two steps. First, a *control flow graph* (CFG) is built. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. E.g., this translation is explained in more detail in [3]. For simplicity, our language does not include advanced features of Java bytecode, such as exceptions, interfaces, static methods and fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and references. Such features can be easily handled in our framework and indeed our implementation deals with full (sequential) Java bytecode.

EXAMPLE 2.1. *Fig. 1 depicts our running example in Java, and Fig. 2 depicts its corresponding rule-based representation where the procedures are named as the method they represent and "*$\mathrm{f_c}$*" and "*$\mathrm{f_d}$*" denote intermediate procedures for* f*. The Java program is included only for clarity as the analyzer generates the rule-based representation from the corresponding bytecode only. As an example, we explain rules (1) and (2) which correspond to method* m*. Each rule is guarded by a corresponding condition, resp.* $\mathbf{n} > \mathbf{0}$ *and* $\mathbf{n} \leq \mathbf{0}$*. Variable names of the form* $\mathrm{s_i}$ *indicate that they originate from stack positions. In rule (1), the "*$\mathsf{new}\ \mathrm{Tree}^1$*" instruction creates an object of type* Tree *(the superscript* 1 *is the unique identifier for this program point) and assigns the variable* $\mathrm{s_0}$ *to its reference (which corresponds to pushing the reference on the stack in the original bytecode). Then, the local variable* n *is decremented by one and the result is assigned to* $\mathrm{s_1}$*. Next, the method* m *is recursively invoked which receives as input argument the result of the previous operation (*$\mathrm{s_1}$*) and returns its result in* $\mathrm{s_1}$*. Similar invocations to methods* m*,* f *and* init *follow. In Java bytecode, constructor methods are named* init*. In both rules, the return value is* r *which in (1) is assigned to the object reference and in (2) to* null*. It can be observed that, like in bytecode, all guards and instructions correspond to three-address code, except for calls to procedures which may involve more variables as parameters. The methods* $\mathrm{intValue_1}$ *and* $\mathrm{init_1}$ *belong to class* Integer*, and* $\mathrm{intValue_2}$ *and* $\mathrm{init_2}$ *belong to class* Long*.* □

Observe in the example that, in our syntax, with the aim of simplifying the presentation, we do not distinguish between calls to methods and calls to intermediate procedures. For instance, $\mathrm{f_c}$ and $\mathrm{f_d}$ are intermediate procedures while f is the method. This distinction can be made observable in the translation phase trivially and, when needed, we assume such distinction is available.

2.1 Semantics

The execution of bytecode in rule-based form is exactly like standard bytecode; a thorough explanation is outside the scope of this paper (see [14]). An *operational semantics* for rule-based bytecode is shown in Fig. 3. An *activation record* is of the form $\langle p, bc, tv\rangle$, where p is a procedure name, bc is a sequence of instructions and tv a variable mapping. Executions proceed between *configurations* of the form $A; h$, where A is a stack of activation records and h is the *heap* which is a partial map from an infinite set of *memory locations* to *objects*. We use $h(r)$ to denote the object referred to by the memory location r in h and $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$. An object o is a pair consisting of the object class tag and a mapping from field names to values which is consistent with the type of the fields.

Intuitively, rule (1) accounts for all instructions in the bytecode semantics which perform arithmetic and assignment operations. The evaluation $eval(exp, tv)$ returns the evaluation of the arithmetic or Boolean expression exp for the values of the corresponding variables from tv in the standard way, and for reference variables, it returns the reference. Rules (2), (3) and (4) deal with objects. We assume that $\mathsf{newobject}(c^i)$ creates a new object of class c and initializes its fields to either 0 or null, depending on their types. Rule (5) (resp., (6)) corresponds to calling (resp., returning from) a procedure. The notation $p[\bar{y}, \bar{y}']$ records the association between the formal and actual return variables. It is assumed that newenv creates a new mapping of local variables for the corresponding method, where each variable is initialized as newobject does.

An execution starts from an *initial configuration* of the form $\langle \bot, p(\langle \bar{x}\rangle, \langle \bar{y}\rangle), tv\rangle; h$ and ends when we reach a *final configuration* $\langle \bot, \epsilon, tv'\rangle; h'$ where tv and h are initialized to suitable initial values, tv' and h' include the final values, and $\bot$ is a special symbol indicating an initial state. We assume that any object stored in the initial heap h is reachable from (at least) one of the x_i, namely there are not *collectable* objects that can removed from h at the initial state. Note that $\mathrm{dom}(tv) = \mathrm{dom}(tv') = \bar{x} \cup \bar{y}$. Finite executions can be regarded as *traces* $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_\omega$, denoted $S_0 \rightsquigarrow^* S_\omega$, where S_ω is a final configuration. Infinite traces correspond to non-terminating executions.

3. Total Memory Allocation Analysis

Let us first define the notion of total memory consumption. We let $size(c)$ denote the amount of memory required to hold an instance object of class c, $size(o)$ denotes the amount of memory occupied by an object o, and $size(h)$ denotes the amount of memory occupied by all objects in the heap h, namely $\Sigma_{r \in \mathrm{dom}(h)} size(h(r))$. We consider the semantics in Fig. 3 where no GC is performed. Given a trace $t \equiv A_1; h_1 \rightsquigarrow^* A_n; h_n$, the *total memory allocation* of t is defined as $total(t) = size(h_n) - size(h_1)$.

In this section, we briefly overview the application of the cost analysis framework, originally proposed by Wegbreit [20], to total memory consumption inference of bytecode as proposed in [3]. The original analysis framework [1] takes as input a program and a cost model $\mathcal{M}$, and outputs a closed-form upper bound that describes its execution cost w.r.t. $\mathcal{M}$. The cost model $\mathcal{M}$ defines the cost that we want to accumulate. For instance, if the cost model is the number of executed instructions, $\mathcal{M}$ assigns cost 1 to all instructions. The application of this framework to total memory consumption of bytecode takes as input a bytecode program and the following cost model $\mathcal{M}^t$, which is a simplification for our language of the cost model for heap space usage of [3].

DEFINITION 3.1 (heap consumption cost model [3]). *Given a bytecode instruction b, the heap consumption cost model is defined as*

$$\mathcal{M}^t(b) = \begin{cases} size(c^i) & b \equiv x := \mathsf{new}\ c^i \\ 0 & \textit{otherwise} \end{cases}$$

For a sequence of instructions, $\mathcal{M}^t(b_1 \cdots b_n) = \mathcal{M}^t(b_1) + \cdots + \mathcal{M}^t(b_n)$. □

3.1 Inference of Size Relations

The aim of the analysis is to approximate the memory consumption of the program as an upper bound function in terms of its input *data size*s. As customary, the *size* of data is determined by its variable type: the size of an integer variable is its value; the size of an array is its length; and the size of a reference variable is the length of the longest path that can be traversed through the corresponding object (e.g., length of a list, depth of a tree, etc.). To keep the presentation simple, we use the original variable names (possible primed) to refer to the corresponding abstract (size) variables; but we write the size in *italic* font. For instance, let x be a reference to a tree, then x represents the depth of x. When we need to compute the sizes $\bar{v}$ of a given tuple of variables $\bar{x}$, we use the notation $\bar{v} = \alpha(\bar{\mathrm{x}}, tv, h)$, which means that the integer value v_i is the size of the variable $\mathrm{x_i}$ in the context of the variables table tv and the heap h. For instance, if x is the reference to a tree, we need to access the heap h where the tree is allocated to compute its depth and obtain v. If x is an integer variable, then its size (value) can be obtained from the variable table tv.

Standard *size analysis* is used in order to obtain relations between the sizes of the program variables at different program points. For instance, associated to procedure $\mathrm{f_c}$, we infer the size relation $n' = n/2$ which indicates that the value of n decreases by half when calling $\mathrm{f_c}$ recursively. We denote by φ_r the conjunction of linear constraints which describes the relations between the abstract variables of a rule r and refer to [9, 3] for more information.

$$(1)\quad \frac{b \equiv x := exp,\ \ v = eval(exp, tv)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto v]\rangle{\cdot}A; h}$$

$$(2)\quad \frac{b \equiv x := \mathsf{new}\ c^i,\quad o{=}\mathsf{newobject}(c^i),\quad r{\notin}dom(h)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto r]\rangle{\cdot}A; h[r \mapsto o]}$$

$$(3)\quad \frac{b \equiv x := y.f,\ \ tv(y) \neq \mathsf{null},\ \ o = h(tv(y))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto o.f]\rangle{\cdot}A; h}$$

$$(4)\quad \frac{b \equiv x.f := y,\ \ tv(x) \neq \mathsf{null},\ \ o = tv(x)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv\rangle{\cdot}A; h[o.f \mapsto tv(y)]}$$

$$(5)\quad \frac{\begin{array}{c} b \equiv q(\langle \bar{x}\rangle, \langle \bar{y}\rangle),\ \text{there is a program rule } q(\langle \bar{x}'\rangle, \langle \bar{y}'\rangle){:=}g, b_1, \cdots, b_k \\ \text{such that } tv'{=}\mathsf{newenv}(q),\ \forall i.tv'(x'_i) = tv(x_i),\ eval(g, tv') = true \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle q, b_1 \cdot \ldots \cdot b_k, tv'\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv\rangle{\cdot}A; h}$$

$$(6)\quad \frac{}{\langle q, \epsilon, tv\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv'\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv'[\bar{y} \mapsto tv(\bar{y}')]\rangle{\cdot}A; h}$$

Figure 3. Operational semantics of bytecode programs in rule-based form

(1)	$m(n){=}size(\mathrm{Tree}^1){+}m(s_1){+}m(s_2){+}$ $f(n){+}init(s_0, s_1, s_2, s_3)$	$\{n{>}0, s_0{=}1,$ $s_1{=}n{-}1, s_2{=}n{-}1\}$
(2)	$m(n){=}0$	$\{n{\leq}0\}$
(3)	$f(n){=}f_c(n, a) + f_d(i, a')$	$\{a{=}0, i{=}n\}$
(4)	$f_c(n, a){=}g(n){+}f_c(n', a')$	$\{n{>}1, n'{=}n/2\}$
(5)	$f_c(n, a){=}0$	$\{n{\leq}1\}$
(6)	$f_d(i, a){=}h(i){+}f_d(i', a')$	$\{i{>}1, i'{=}i/2\}$
(7)	$f_d(i, a){=}0$	$\{i{\leq}0\}$
(8)	$g(n){=}size(\mathrm{Integer}^2){+}size(\mathrm{Integer}^3)$	$\{x{=}1\}$
(9)	$h(n){=}size(\mathrm{Long}^4)$	$\{r{=}1, s_0{=}n{-}1\}$
(10)	$init(this, l, r, d){=}0$	$\{\}$

Figure 4. Total Allocation CR.

3.2 Generation of Cost Relations

In a nutshell, given a bytecode program P, the analysis of [3] proceeds in three steps: (1) it first transforms it into an equivalent rule-based program (our work directly starts from such rule-based form), (2) it infers size relations as explained above, (3) it generates a CR which describes the total memory consumption of the program as follows.

DEFINITION 3.2 (total allocation CR [3]). *Consider a rule* $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}}\rangle, \langle \bar{\mathrm{y}}\rangle){::=}\mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ *and the size relations* φ_r *computed for* r*. We distinguish the subsequence of all calls to procedures* $\mathrm{b}_{\mathrm{i}_1} \ldots \mathrm{b}_{\mathrm{i}_\mathrm{k}}$ *in* r*, with* $1 \leq i_1 \leq \cdots \leq i_k \leq n$ *and assume* $\mathrm{b}_{\mathrm{i}_\mathrm{j}} = \mathrm{q}_{\mathrm{i}_\mathrm{j}}(\langle \bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}\rangle, \langle \bar{\mathrm{y}}_{\mathrm{i}_\mathrm{j}}\rangle)$*. Then, the cost equation for* r *is:*

$$p(\bar{x}) = \mathcal{M}^t(g, b_1, \ldots, b_n) + \Sigma_{j=1}^{k}\ q_{i_j}(\bar{x}_{i_j}),\ \ \varphi_r$$

Given a program P*, we denote by* $\mathcal{S}_P$ *the cost relation generated for each rule in* P *w.r.t. the heap consumption cost model* $\mathcal{M}^t$*.* □

Note that each call in the rule $\mathrm{q}_{\mathrm{i}_\mathrm{j}}(\langle \bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}\rangle, \langle \bar{\mathrm{y}}_{\mathrm{i}_\mathrm{j}}\rangle)$ has a corresponding abstract version $b^{\alpha}_{i_j} = q_{i_j}(\bar{x}_{i_j})$ where $\bar{x}_{i_j}$ are the size abstractions of $\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}$. The output variables are ignored in the CR as the cost is a function of the *input* data sizes, however it should be noted that the possible effect of output variables on the cost has been already modeled by the size relation φ_r. For simplicity, the same procedure name is used to define its associated cost relation, but in *italic* font.

EXAMPLE 3.3. *The CR generated for the rule-based program in Fig. 2 w.r.t.* $\mathcal{M}^t$ *is depicted in Fig. 4. To simplify the presentation, we assume that the total heap consumption of all external methods (*init_1*,* $\mathrm{intValue}_1$*,* init_2 *and* $\mathrm{intValue}_2$*) is* 0 *and we do not show them in the equations from now on. Consider, for example, equation (4). It states that the memory consumption of executing* $\mathrm{f_c}(\langle \mathrm{n}, \mathrm{a}\rangle, \langle \mathrm{n}, \mathrm{a}\rangle)$ *is the total memory consumption of executing* $\mathrm{g}(\langle \mathrm{n}\rangle, \langle \mathrm{r}\rangle)$ *plus the one of* $\mathrm{f_c}(\langle \mathrm{n}', \mathrm{a}'\rangle, \langle \mathrm{n}', \mathrm{a}'\rangle)$*. The set of constraints attached to equation (4) includes information on: (1) how the sizes of the data change when the program moves from one rule to another, e.g., the constraint* $n' = n/2$ *indicates that the value of* n *decreases by half when calling* $\mathrm{f_c}$ *recursively; and (2) numeric conditions (obtained by abstracting the guards) under which the corresponding rule is applicable, e.g.,* $n > 1$ *indicates that the equation can be applied only when* n *is greater than* 1. □

An important observation is that, as discussed in Sec. 1, this analysis approach is intrinsically designed to infer the *total cost* (memory allocation in this case) of the program's execution and not to infer its peak consumption. This is because the equations accumulate the cost of all instructions and rules together as it can be observed in the CR for the example above.

3.3 Closed-Form Upper Bounds

Once the CR is generated, a cost analyzer has to use a CR solver in order obtain closed-form upper bounds, i.e., expressions without recurrences. The technical details of this process are not explained in the paper as our analysis does not require any modification to such part. In what follows, we rely on the CR solver of [3] (which can be accessed online through a web interface) to obtain closed-form upper bounds for our examples. The soundness of the overall analysis, as stated in the next theorem, requires that the equations generated as well as their closed-form upper bounds are sound.

THEOREM 3.4 (soundness [3]). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}}\rangle, \langle \bar{\mathrm{y}}\rangle) \cdot bc, tv_1\rangle \cdot A; h_1 \rightsquigarrow^* \langle \mathrm{q}, bc, tv_n\rangle{\cdot}A; h_n$*, then* $p(\bar{v}) \geq total(t)$ *where* $\bar{v} = \alpha(\bar{x}, tv_1, h_1)$*.* □

Observe that the trace t in the theorem represents an execution of procedure p for some specific input data (properly stored in tv_1 and h_1) where the first configuration corresponds to calling p and the last one to returning from that specific call. As already mentioned in Sec. 3.1, $\bar{v}$ denotes the size of the input data.

EXAMPLE 3.5. *Solving the equations of Fig. 4 results in the following closed-form upper bounds for* f, m, g *and* h*:*

$$\begin{aligned} m(n) &= (2^{\mathsf{nat}(n)}-1)*(f(n)+size(\mathrm{Tree}^1))\\ f(n) &= \log_2(\mathsf{nat}(n-1)+1)*(size(\mathrm{Integer}^2)+size(\mathrm{Integer}^3))+\\ &\quad \log_2(\mathsf{nat}(n-1)+1)*size(\mathrm{Long}^4))\\ f_c(n,a) &= \log_2(\mathsf{nat}(n-1)+1)*(size(\mathrm{Integer}^2)+size(\mathrm{Integer}^3))\\ f_d(i,a) &= \log_2(\mathsf{nat}(i-1)+1)*size(\mathrm{Long}^4))\\ g(n) &= size(\mathrm{Integer}^2)+size(\mathrm{Integer}^3)\\ h(n) &= size(\mathrm{Long}^4) \end{aligned}$$

where the expression $\mathsf{nat}(l)$ *is defined as* $\max(l,0)$ *to avoid negative evaluations. As expected, method* m *has an exponential memory consumption due to the two recursive calls, which in turn is multiplied by the allocation at each iteration (i.e., the consumption of* f *plus the creation of a* Tree *object). The solver indeed substitutes* $f(n)$ *by its upper bound shown below. The memory consumption of* f *has two logarithmic parts: the leftmost one corresponds to the first loop which accumulates the allocation performed along the execution of* $g(n)$*, the rightmost one corresponds to the second loop with the allocation of* $h(n)$*.* □

A fundamental observation is that the above upper bounds on the memory consumption can be tighter if one considers the effect of GC. For instance, a more precise upper bound for m can be inferred if we take into account that the memory allocated by f can be entirely garbage collected upon return from f. Likewise, the upper bound for f can be more precise if we take advantage of the fact that not all memory escapes from g. The goal of the rest of the paper is to provide automatic techniques to infer accurate bounds by taking into account the memory freed by scoped-GC.

4. Escaped Memory Upper Bounds

In a real language, GC removes objects which become *unreachable* along the program's execution. Given a configuration $A;h$, we say that an object $o = h(r)$ where $r \in \mathrm{dom}(h)$ is not reachable, if it cannot be accessed (directly or indirectly) through the variables table tv of any activation record in A. To develop our analysis, we assume a scoped-memory manager, which at the level of the source language, meets these conditions: (1) it reclaims memory *only* upon return from methods and, (2) it collects *all* unreachable objects which have been created *during* the execution of the corresponding method call.

In order to simulate the behavior of such garbage collector at the level of the corresponding rule-based bytecode, it is enough to assume that the memory manager reclaims memory only upon return from procedures that correspond to methods but not from procedures that correspond to intermediate states like $\mathrm{f_c}$ and $\mathrm{f_d}$. We use $\leadsto_{gc}$ to denote $\leadsto$-transitions with a scoped-memory manager which meets the two conditions above. In this context, the *escaped memory* of a procedure execution is defined as follows. Given a trace $t \equiv \langle q, p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)\cdot bc, tv_1\rangle\cdot A; h_1 \leadsto^*_{gc} \langle q, bc, tv_n\rangle\cdot A; h_n$ whose first configuration corresponds to calling p and the last one to returning from that specific call, the escaped memory of t is $escaped(t) = size(h_n) - size(h_1)$, which corresponds to the amount of memory allocated during the execution of p and still live in the memory upon exit from p. Our first contribution is an automatic technique to infer *escaped memory upper bounds*.

4.1 Inference of Escape Information

We say that an object *escapes* from a procedure p, in the context of a scoped-memory manager, if it is created during the execution of p, and still available in the heap upon exit from p. Note that if p corresponds to an intermediate procedure, such object might be unreachable but still has not been garbage collected because GC is applied only when exiting from procedures that correspond to methods in the original program. As a preprocessing phase, for each procedure p, we need to over-approximate the set of allocation instructions "new c^i" that might be executed when calling p and its transitive calls such that it is guaranteed that *all* objects they create are not in memory upon exit from p, i.e., they have been garbage collected. Recall that an allocation instruction "new c^i" is uniquely identified by the *tagged* class c^i. We use the notation $A \setminus B$ for the difference on sets.

DEFINITION 4.1 (collectable objects). *Given a procedure* p*, we denote by* $collectable(\mathrm{p})$ *the set of all allocation instructions, identified by their* tagged *classes, defined as follows.*

$c^i \in collectable(\mathrm{p})$ *iff the following conditions hold:*

1. *"*new c^i*" is a reachable instruction from* p*;*
2. *for any trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle\bar{\mathrm{x}}\rangle,\langle\bar{\mathrm{y}}\rangle)\cdot bc, tv_1\rangle\cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n\rangle\cdot A; h_n$*, it holds that* $\forall r \in \mathrm{dom}(h_n)\setminus\mathrm{dom}(h_1)$ *the object* $h_n(r)$ *is not an instance of* c^i*.* □

The set of collectable objects can be approximated from the information computed by escape analysis [15, 4]. The goal of escape analysis is to determine all program points where an object is reachable and whether the lifetime of the object can be proven to be restricted only to the current method. In our implementation, we use the approach described in [21] which, as our experiments show, behaves well in practice.

EXAMPLE 4.2. *The escape information is computed for all procedures (both methods and intermediate rules) defined in Fig. 2:*

$$\begin{aligned} &collectable(\mathrm{m}) = collectable(\mathrm{f}) = \{\mathrm{Integer}^2, \mathrm{Integer}^3, \mathrm{Long}^4\}\\ &collectable(\mathrm{f_c}) = collectable(\mathrm{g}) = \{\mathrm{Integer}^2\}\\ &collectable(\mathrm{f_d}) = collectable(\mathrm{h}) = \emptyset \end{aligned}$$

As an example, the information in the set $collectable(\mathrm{f})$ *states that the objects created with class tags* $\mathrm{Integer}^2$*,* $\mathrm{Integer}^3$ *and* Long^4 *during the execution of* f *by the transitive calls to* g *and* h*, do not escape from* f*. Also,* $collectable(\mathrm{f_d}) = \emptyset$ *means that the object* Long^4 *created in* h *might escape from* $\mathrm{f_d}$*. An important observation is that this object is not reachable upon exit from* $\mathrm{f_d}$*, but since GC is applied only upon exit from procedures that correspond to methods, it will be collected only upon exit from* f*. This issue will be further discussed in Sec. 6.* □

4.2 Upper Bounds on the Escaped Memory

Intuitively, our technique to infer upper bounds on the escaped memory consists of two steps. In the first step, we generate equations for the total allocation (exactly as stated in Def. 3.2) which accumulate *symbolic* expressions of the form $size(c^i)$ to represent the heap allocation for the instruction new c^i, rather than its concrete allocation size. From these equations, we obtain an upper bound for the total memory allocation as a symbolic expression which contains residual $size(c^i)$ sub-expressions. The main novelty is that, in a second step, we tighten up such total allocation upper bound to extract from it only its escaped memory as follows. Given a procedure p, and its total heap consumption upper bound $p(\bar{x})$, we obtain the upper bound on the escaped memory by replacing expressions of the form $size(c^i)$ by 0 if it is guaranteed that all corresponding objects are not available in the memory upon exit from p, namely $c^i \in collectable(\mathrm{p})$. Given an expression exp and a substitution σ from sub-expressions to values, $exp[\sigma]$ denotes the application of σ on exp.

DEFINITION 4.3 (escaped memory upper bound). *Given a procedure* p*, its escape information* $collectable(\mathrm{p})$*, and its (symbolic) upper-bound for the total memory allocation* $p(\bar{x}) = exp$*, the escaped memory upper-bound of* p *is defined as:* $\check{p}(\bar{x}) = exp[\forall c^i \in collectable(\mathrm{p}).size(c^i) \mapsto 0]$. □

Observe that, in the above definition, it is required that the set $collectable(\mathrm{p})$ contains the information for objects created in transitive calls from p, as stated in Def. 4.1, because escaped memory upper-bounds for a method p are obtained by using only the information in $collectable(\mathrm{p})$ and not in any other $collectable(\mathrm{q})$ with $\mathrm{q} \neq \mathrm{p}$. This is an essential difference w.r.t. existing work [3] which does not compute information for transitive calls, but instead computes the escape information only for the objects which are created inside each method (excluding its transitive calls). We obtain strictly more accurate bounds as the following example illustrates.

EXAMPLE 4.4. *Applying Def. 4.3 to the total heap allocation inferred in Ex. 3.5, by using the escape information of Ex. 4.2, results in the escaped memory upper bounds:*

$$\check{m}(n) = (2^{\mathsf{nat}(n)} - 1) * size(\mathrm{Tree}^1) \qquad \check{f}(n) = 0$$
$$\check{f}_c(n,a) = \log(\mathsf{nat}(n-1)+1) * size(\mathrm{Integer}^3) \qquad \check{g}(n) = size(\mathrm{Integer}^3)$$
$$\check{f}_d(i,a) = \log(\mathsf{nat}(i-1)+1) * size(\mathrm{Long}^4) \qquad \check{h}(n) = size(\mathrm{Long}^4)$$

We can see that the escaped memory upper bound for m *does not accumulate the allocations of* Long^4 *nor* $\mathrm{Integer}^2$ *and* $\mathrm{Integer}^3$ *objects because they do not escape from* f. *In [3], the allocations corresponding to* $\mathrm{Integer}^3$ *and* Long^4 *are accumulated because they escape from the method where these objects have been created. The problem is that in [3] they are accumulated in the CR and hence in all upper bounds for methods that transitively invoke* g *and* h. □

The following theorem states the soundness of our escaped memory upper bounds.

THEOREM 4.5 (soundness). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$*, then* $\check{p}(\bar{v}) \geq escaped(t)$ *where* $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_1, h_1)$.

Proof.
(sketch) First, by Theorem 3.4, we have the soundness of the total allocation upper bound $p(\bar{v}) \geq total(t)$. Second, by the soundness of escape analysis [4], we know that $collectable(\mathrm{p})$ gives a safe approximation of the objects that escape from t. Now, by combining both parts, we have that $\check{p}(\bar{v}) \geq escaped(t)$ and, hence, the soundness of $\check{p}(\bar{v})$ follows. □

5. Live Heap Space Analysis

This section presents a novel live heap space analysis for garbage-collected languages which obtains precise upper bounds including logarithmic, exponential, etc. complexity classes. Achieving accuracy is crucial because live heap bounds represent the minimum amount of memory required to execute a program.

5.1 The Notion of Peak Consumption

Essentially, given a trace $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$, the peak consumption can be defined as $peak(t) = max(size(h_2), \ldots, size(h_n)) - size(h_1)$. We decrement $size(h_1)$ because the objects created in an outer scope (i.e., those in h_1) cannot be collected during the execution t, as stated in condition (2) of scoped-GC in Sec. 4.

Let us illustrate this notion by means of this simple method "void r() {A; p(); B; q(); C;}" whose memory consumption is showed in Fig. 5. A, B and C are sequences of instructions that do not contain any method invocation. We use the notation $\hat{p}$ to the note the peak consumption of executing the method p. We can observe that the peak heap consumption $\hat{r}$ is the maximal of three possible scenarios: (1) In the leftmost column, we depict a scenario where we allocate A and then execute p, thus we add the peak heap consumption of p. (2) In the next alternative scenario, we still have A and then return from p's execution, thus we add the memory escaped upon return from p (i.e., $\check{p}$) and we continue until the execution of q. Hence we add B plus the peak of q. (3) In the next column, we have A, plus the memory escaped from p, plus B, plus the memory escaped from q, plus C. Observe that any of these scenarios may correspond to the actual peak and we need to infer upper bounds for all of them and then take the maximal. The rightmost column indicates the upper bound for total allocation which is clearly much less accurate.

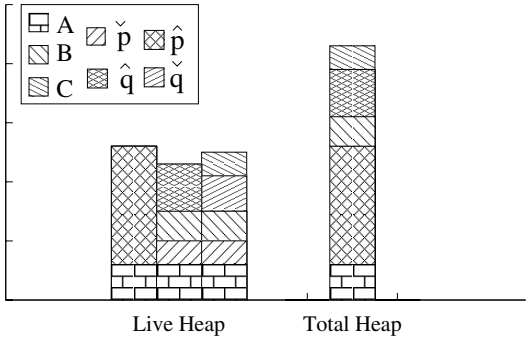


Figure 5. Memory Consumption of simple program

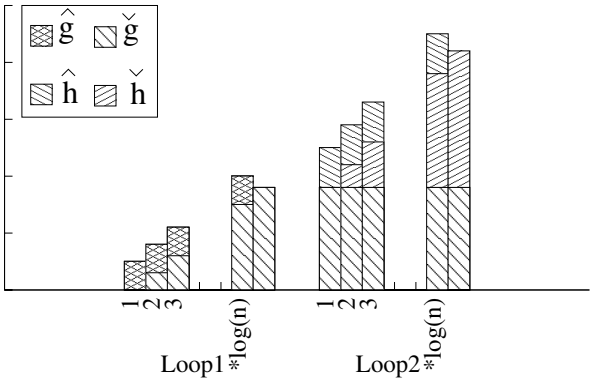


Figure 6. Memory Consumption of running example

In general the problem is more complicated, e.g., when method invocations occur within loops. Fig. 6 depicts the actual memory consumption of the execution of method f in our running example. Column 1 captures the heap allocation of executing g at the first iteration of the first loop (defined by procedure $\mathrm{f_c}$). Column 2 represents the escaped memory from g plus the next iteration of the loop where g allocates again $\hat{\mathrm{g}}$ memory and so on. As the loop in $\mathrm{f_c}$ is executed $\log(n)$ times we have all such possible scenarios over the tag Loop 1. Then, we start the execution of the second loop with an initial heap usage of $log(n)$ times the memory escaped from g. Similarly, at each iteration of the second loop, method h is invoked which allocates a maximal of memory $\hat{\mathrm{h}}$ and upon return, we need to consider the escaped memory from h plus the next execution. As the loop is executed $\log(n)$ times, we have all possible scenarios to the right grouped over the tag Loop 2. The peak heap allocation of executing f is the maximal of all such scenarios, namely the maximal between the two scenarios marked with $*$. The important point is that we need to infer upper bounds for $\hat{\mathrm{h}}$, $\hat{\mathrm{g}}$, $\check{\mathrm{h}}$, $\hat{\mathrm{g}}$ and generate as peak heap consumption the expression $\hat{\mathrm{f}} = \max(\hat{\mathrm{g}} + (\log(\mathrm{n}) - 1) * \check{\mathrm{g}}, \hat{\mathrm{h}} + (\log(\mathrm{n}) - 1) * \check{\mathrm{h}} + \log(\mathrm{n}) * \check{\mathrm{g}})$. Note that, in principle, it could happen that $\hat{\mathrm{g}} > (\log(\mathrm{n}) - 1) * \check{\mathrm{h}} + \hat{\mathrm{h}}$.

5.2 Peak Consumption Cost Relation

We now propose a novel approach for generating CR that, by relying on the escaped memory bounds, capture the peak heap consumption by considering all possible states of a program's execution. Our proposal is based on the following intuition: Let m_1 and m_2 be two methods, and let $\hat{m}_1(\bar{x}_1)$ and $\hat{m}_2(\bar{x}_2)$ be the peak heap consumption of executing m_1 and m_2 respectively, then the peak heap consumption of the two consecutive calls $\mathrm{m}_1;\mathrm{m}_2$ is $\max(\hat{m}_1(\bar{x}_1), \check{m}_1(\bar{x}_1) + \hat{m}_2(\bar{x}_2))$. The following definition generalizes this idea for an arbitrary sequence of statements.

DEFINITION 5.1 (peak consumption CR). *Consider a rule* $\mathrm{r} \equiv \mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle)::=\mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ *and its corresponding size relations* φ_r. *Then, its* peak consumption *equation is* $\hat{p}(\bar{x}) = \mathcal{T}(b_1, \ldots, b_n),\ \varphi_r$ *where* $\mathcal{T}$ *is defined as follows:*

$$\mathcal{T}(b_1,\ldots,b_n)::= \begin{cases} 0 & \text{if } n = 0 \\ \max(\hat{q}(\bar{x}_1), \check{q}(\bar{x}_1) + \mathcal{T}(b_2,\ldots,b_n)) & \text{if } b_1 = \mathrm{q}(\langle\bar{\mathrm{x}}_1\rangle, \langle\bar{\mathrm{y}}_1\rangle) \text{ is a call} \\ \mathcal{M}^t(b_1) + \mathcal{T}(b_2,\ldots,b_n) & \text{if } b_1 \text{ is an instruction} \end{cases}$$

Given a program P, *we denote by* $\hat{\mathcal{S}}_P$ *the* peak consumption cost relation *generated for each rule in* P. □

In the above definition, it can be observed that, in the second case, we generate two possible scenarios not only for methods, but also for intermediate procedures. These scenarios correspond to either the peak of the first procedure call or to the escaped memory from the first procedure call plus the peak of the rest of the instructions sequence. Considering the two scenarios at the level of procedures (no only of methods) allows us to gain further accuracy in situations, like in the method f, in which intermediate procedures correspond to loops which contain method invocations. The next example illustrates this point.

EXAMPLE 5.2. *The peak consumption CR* $\hat{\mathcal{S}}_P$ *of the rule-based program is different from the one in Fig. 4 in equations (1), (3), (4) and (6) which are now as follows:*

$$\begin{array}{lll} (1) & \hat{m}(n) = & size(\mathrm{Tree}^1) + \max(\hat{m}(s_1), \check{m}(s_1) + \max(\hat{m}(s_2), \\ & & \check{m}(s_2) + \max(\hat{f}(n), \check{f}(n) + \hat{init}(s_0,s_1,s_2,s_3)))) \\ (3) & \hat{f}(n) = & \max(\hat{f}_c(n,a), \check{f}_c(n,a) + \hat{f}_d(i,a')) \\ (4) & \hat{f}_c(n,a) = & \max(\hat{g}(n), \check{g}(n) + \hat{f}_c(n',a')) \\ (6) & \hat{f}_d(i,a) = & \max(\hat{h}(i), \check{h}(i) + \hat{f}_d(i',a')) \end{array}$$

with the same constraints as those of Fig. 4. We can now replace the escaped memory upper bounds $\check{g}$, $\check{h}$, $\check{m}$ *and* $\check{f}$ *by the ones in Ex. 4.4. As an optimization, we do not apply the transformation to the last call in the rules, for instance, to the call to* $init$ *in equation (1), since trivially* $\hat{init} \geq \check{init}$. *Observe that in equation (3) we have applied also two possible scenarios to the intermediate procedure* f_c *which does not correspond to a method by introducing the max operator. This is essential to keep the two possible peaks (marked with "*" in the figure) separate instead of accumulating both of them, which would lead to a larger, less accurate upper bound. Besides, it is sound w.r.t. scoped-GC because the corresponding escaped memory bounds for* $\hat{f}_c$ *and* $\hat{f}_d$ *are obtained by considering that GC takes place upon method's return only.*

The most important point is that equation (4) accurately captures the memory consumption of all scenarios in Loop 1 of Fig. 6 and equation (6) captures those in Loop 2 to the right of the figure, as it will become clear after solving the equations in Ex. 5.3. □

An important feature of our CR $\hat{\mathcal{S}}_P$ is that they can still be solved by relying on a standard upper bound solver for CR produced by cost analysis like the one in [3]. The only adjustment is that our CR use the $\max$ operator which is frequently not supported. This is handled by a further preprocessing which transforms one equation that uses $\max$ into an equivalent set of equations that do not use $\max$ by creating nondeterministic equations whenever we have $\max$. In particular, an equation of the form $p(\bar{x}) = A + max(B, C), \varphi$ is translated into the two equations $p(\bar{x}) = A + B, \varphi$ and $p(\bar{x}) = A + C, \varphi$. Since an upper bound solver looks for an upper bound for all possible paths, it is guaranteed that this transformation simulates the effect of the $\max$ operator. Nested $\max$ are translated iteratively. For instance, the translation of equation (1) in Ex. 5.2, results in the following equations:

$$\begin{array}{l} \hat{m}(n) = size(\mathrm{Tree}^1) + \hat{m}(s_1), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \hat{m}(s_2), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \check{m}(s_2) + \hat{f}(n), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \check{m}(s_2) + \check{f}(n) + \hat{init}(s_0,s_1,s_2,s_3), \varphi_1 \end{array}$$

EXAMPLE 5.3. *Solving the transformed equations results in the following closed-form upper bounds:*

$$\begin{array}{lll} \hat{m}(n) & = 2^{\mathsf{nat}(n)} * size(\mathrm{Tree}^1) + \hat{f}(n) \\ \hat{f}(n) & = \max(\hat{f}_c(n,a), \check{f}_c(n,a) + \hat{f}_d(n,a')) \\ \hat{f}_c(n,a) & = (\log(\mathsf{nat}(n-1)+1)+1) * size(\mathrm{Integer}^3) + size(\mathrm{Integer}^2) \\ \hat{f}_d(i,a) & = (\log(\mathsf{nat}(i-1)+1)+1) * size(\mathrm{Long}^4) \\ \hat{g}(n) & = size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3) \\ \hat{h}(n) & = size(\mathrm{Long}^4) \end{array}$$

We can observe that the peak bound for f *accurately captures the maximal of the two scenarios in the figure: (1)* $\hat{f}_c(n,a)$ *corresponds to the leftmost column of Fig. 6 (since* $\check{g}$ *is* $size(\mathrm{Integer}^3)$ *which is accumulated* $log(n)-1$ *times and* $\hat{g}(n)$ *is* $size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3)$ *and (2)* $\check{f}_c(n,a) + \hat{f}_d(n,a')$ *corresponds to the rightmost column where, as expected, we accumulate* $log(n) - 1$ *times the escaped* $size(\mathrm{Long}^4)$ *object plus an additional one which is the peak consumption of* h *(and nothing escapes from* $\mathrm{f_c}$*).*

It is fundamental to observe the difference between the above live heap space bound for m *and the total allocation computed in Ex. 3.5. In our live bound, since the allocation required by* f *can be entirely garbage collected upon exit from* f, *the required heap is not proportional to the number of times that* f *is invoked (i.e., exponential on* n*) but rather the memory required for a single execution of* f. □

The following theorem states that the upper bounds computed by our analysis are *sound*, i.e., for any input values, they evaluate to a larger value than the actual peak consumption.

THEOREM 5.4 (soundness). *Given a procedure* p, *and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle) \cdot bc, tv_1\rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle \mathrm{q}, bc, tv_n\rangle \cdot A; h_n$ *then* $\hat{p}(\bar{v}) \geq peak(t)$ *where* $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_1, h_1)$. □

6. Approximating the Ideal Garbage Collector

In this section, we show how the analysis of Sec. 5 can be refined to consider other GC schemes and, in particular, to get closer to the *ideal* GC manager where objects are collected as soon as they become unreachable. For instance, the peak consumption upper bound inferred in Ex. 5.3 for f is accurate when using a scope-GC scheme, since all objects created inside the loops are collected only upon exit from f. However, it is clearly inaccurate for an ideal GC scheme, since the lifetime of each object created in f is limited to one iteration of the corresponding loop, and therefore f can be executed in constant heap space.

Luckily, we can take advantage of scopes in the rule-based representation in order to infer accurate upper bounds for such GC schemes without modifying our analysis. In Def. 4.1 the effect of GC is considered only on exit from procedures that correspond to methods, this is essential in order to obtain safe upper bounds for scoped-GC, since in the original language GC is assumed to be applied upon exit from method scopes. However, the rule-based language distinguishes scopes that correspond to code fragments (in

the original program) smaller than methods, e.g., $\mathrm{f_c}$ and $\mathrm{f_d}$ respectively correspond to the first and second loop of f. Considering the effect of GC on exit from these (non-method) smaller scopes corresponds to applying more often GC than in the original language, and therefore getting closer to the ideal GC. In order to support this, we need to compute the set of collectable objects for blocks exactly as we do for methods in Def. 4.1. Let us see an example.

EXAMPLE 6.1. *If we apply GC upon exit from* $\mathrm{f_c}$*, then the collectable objects are* $collectable(\mathrm{f_c}) = \{\mathrm{Integer}^2, \mathrm{Integer}^3\}$*, and hence* $\check{f}_c(n,a) = 0$*. Observe that in Ex. 4.2* $collectable(\mathrm{f_c})$ *contains only* $\mathrm{Integer}^3$*. This in turn improves the peak consumption for* f *to* $\hat{f}(n) = \max(\hat{f}_c(n,a), \hat{f}_d(n,a'))$*, which is clearly more precise than the one in Ex. 5.3.* □

Interestingly, the above upper-bound can be even further improved in order to obtain one which is as close as possible to the ideal behavior. Consider Rule (6) in Fig. 2 which corresponds to the second loop in f. The object created in h, and escaped to the calling context, becomes unreachable immediately after executing $\mathrm{intValue_2}$. Thus, if we separate the loop's body into a separate procedure $\mathrm{f'_d}$, we make this behavior observable to our analysis. This can be done by transforming the rules associated to the loops as follows:

(4) $\mathrm{f_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle) ::=$
$\mathrm{f'_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle).$
$\mathrm{f_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle).$

$\mathrm{f'_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle) ::=$
$\mathbf{n > 1}$,
$\mathrm{g}(\langle \mathrm{n}\rangle, \langle \mathrm{s_0}\rangle)$,
$\mathrm{intValue_1}(\langle \mathrm{s_0}\rangle, \langle \mathrm{s_0}\rangle)$
$\mathrm{a := a + s_0}$,
$\mathrm{n := n/2}$.

(6) $\mathrm{f_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle) ::=$
$\mathrm{f'_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle).$
$\mathrm{f_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle).$

$\mathrm{f'_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle) ::=$
$\mathbf{i > 1}$,
$\mathrm{h}(\langle \mathrm{i}\rangle, \langle \mathrm{s_0}\rangle)$,
$\mathrm{intValue_2}(\langle \mathrm{s_0}\rangle, \langle \mathrm{s_0}\rangle)$
$\mathrm{a := a * s_0}$,
$\mathrm{i := i/2}$.

Now the peak consumption equations for $\mathrm{f_c}$ and $\mathrm{f_d}$ are:

$$\hat{f}_c(n,a) = \max(\hat{f}'_c(n,a), \check{f}'_c(n,a) + \hat{f}_c(n',a')) \quad \{n{>}1, n'{=}n/2\}$$
$$\hat{f}_c(n,a) = 0 \quad \{n{\leq}1\}$$
$$\hat{f}_d(i,a) = \max(\hat{f}'_d(i,a), \check{f}'_d(i,a) + \hat{f}_d(i',a')) \quad \{i{>}1, i'{=}i/2\}$$
$$\hat{f}_d(i,a) = 0 \quad \{i{\leq}1\}$$
$$\hat{f}'_c(n,a) = size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3)$$
$$\hat{f}'_d(i,a) = size(\mathrm{Long}^4)$$

and, since $\check{f}'_c(n,a){=}\check{f}'_d(i,a){=}0$, solving them results in

$$\hat{f}_c(n,a){=}size(\mathrm{Integer}^2){+}size(\mathrm{Integer}^3)$$
$$\hat{f}_d(i,a){=}size(\mathrm{Long}^4)$$

which in turn improves the upper bound of f to

$$\hat{f}(n) = max(size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3), size(\mathrm{Long}^4))$$

which is indeed the minimal amount of memory required in order to execute f in the presence of an ideal GC.

In order to support such transformations, one should guide the transformation from the bytecode to the rule-based program by the information previously computed on the lifetime of the different objects. Such analysis should give us indications about when it is profitable to make smaller scopes. Currently, we do this transformation only for scopes that correspond to loops. Also, it should be noted that there is an efficiency versus accuracy trade-off here, as we generate more equations in this case which thus will be more expensive to solve. Note that the same ideas are useful for supporting region-based memory management. The idea is to infer regions and use this information to separate the scopes, such that the exit from scopes coincides with the removal of the corresponding region.

7. Experiments

In this section, we assess the practicality of our proposal on realistic programs, the standardized set of benchmarks in the JOlden suite [12]. This benchmark suite was first used by [7] in the context of memory usage verification for a different purpose, namely for checking memory adequacy w.r.t. given specifications, but there is no inference of upper bounds as our analysis does. It has been also used by [5] for our same purpose, i.e., the inference of peak consumption. However, since [5] does not deal with memory-consuming recursive methods, the process is not fully automatic in their case and they have to provide manual annotations. Also, they require invariants which sometimes have to be manually provided. In contrast, our tool is able to infer accurate live heap upper bounds in a fully automatic way, including logarithmic and exponential complexities.

The first column of Table 1 contains the name of the benchmark. For most examples, we analyze the method `main` which transitively requires the analysis of the majority of the methods in the package. Only in those benchmarks whose name appears in two different rows, we do not analyze the `main` but rather all those methods invoked within the `main` that we succeed to analyze. In particular, benchmarks Health(cV), Health(gR), Bh(cTD), Bh(eB), Voronoi(cP), and Voronoi(b) correspond, respectively, to methods `createVertex`, `getResults`, `createTreeData`, `expandBox`, `createPoints`, and `buildDelaunayTriangulation` in the corresponding packages. In benchmark Bh, we cannot obtain an upper bound for the method `stepSystem` which is invoked within `main`. The reason is that this method contains a loop whose termination condition does not depend on the size of the data structure, but rather on the particular value stored at certain locations within the data structure. In general, it is complicated to bound the number of iterations of this kind of loops. Basically, the same situation happens in the method `simulate` of benchmark Health. In Voronoi, we are able to analyze all methods when they are not connected together. Unfortunately, we cannot analyze the `main` which, first invokes the method `createPoints` which returns an object `point` and then invokes the method `point.buildDelaunayTriangulation` on such object. The problem is that the upper bound of `buildDelaunayTriangulation` depends on the size of the object `point` returned by `createPoints` and the size analysis is not able to propagate such relation. It should be noted that, in these three cases, the limitations are not related to our proposal in this paper but to external components which can be independently improved.

The second and third columns in the table show, respectively, the upper bounds for total allocation and for live heap space usage. Note that the cost model we use for the experiments substitutes the symbolic expressions $size(Obj)$ by their corresponding numerical values, so that the system can perform mathematical simplifications. In particular, the size of primitive types is $1, 2, 4$, etc. bytes respectively for byte, char, int, etc.; the size of a reference is set to 4 bytes; and the size of an object is the sum of the sizes of all its fields (including those inherited).[1]

Let us first explain the examples Tsp, Bisort, Health, TreeAdd, Perimeter and Voronoi which follow a similar pattern. Basically, they contain methods (in rule-based form) which have this shape $p(X) ::= alloc(k), p(Y_1), \ldots, p(Y_n)$, i.e., a certain allocation k is accumulated by several recursive calls to the method. The size of the arguments in the recursive calls decrease by half in examples Tsp, Bisort and Voronoi and there are two recursive calls. Thus, their resulting upper bounds are linear. In benchmarks Health, Perimeter and TreeAdd, the size of the argument decreases by a constant; the first two examples contain 4 recursive calls and the

[1] This is just an estimation. The sizes depend on the particular JVM

Bench	Total Allocation Upper Bounds	Live Heap Space Upper Bounds
Mst	nat(A+1)*nat($\frac{A}{4}$) + 33*nat(A+1) + 8	nat(A+1) + 8 + max(nat(A+1)^2 + 18*nat(A+1) + nat($\frac{A}{4}$) + 72, nat(A+1)*nat($\frac{A}{4}$) + 25*nat(A+1) + 2*nat($\frac{A}{4}$) + 48)
Em3d	2*nat(D-1)*(32+nat(B)) + 2*nat(B) + 16*nat(C) + 2*nat(D) + 89	max(4*nat(B) + nat(C) + 2*nat(D) + 2*nat(D-1) + 153, 4*nat(B) + max(16,nat(C)) + 2*nat(D) + 2*nat(D-1) + 153), (34 + nat(B))*nat(D-1) + 6*nat(B) + 3*nat(D) + 313)
Bisort	4*nat(A) + 12*nat(B-1) + 52	max(4*nat(A),12*nat(B-1) + 36)
Tsp	46*nat(2*B-1) + 138	28*nat(2*B-1) + 84
Power	258544	5992
Health(cV)	104*4$^{nat(A)}$ + 416	104*4$^{nat(A)}$ + 416
Health(gR)	28*4$^{nat(A-1)}$ + 36	28*4$^{nat(A-1)}$ + 36
TreeAdd	40*2$^{nat(B-1)}$ + 4*nat(A) + 76	24*2$^{nat(B-1)}$ + 60
Bh(cTD)	96*nat(B) + 128	92*nat(B) + nat(B-1) + 308
Bh(eB)	96	92
Perimeter	56*4$^{nat(B)}$ + 4*nat(A) + 128	56*4$^{nat(B)}$ + 112
Voronoi(cP)	20*nat(2*A-1) + 60	20*nat(2*A-1) + 60
Voronoi(b)	88*2$^{nat(A-1)}$ + 8	88*2$^{nat(A-1)}$ + 8

Table 1. Upper bounds for Total Allocation and Live Heap Usage

latter one 2 recursive calls. Thus, their resulting upper bounds are exponential. The upper bounds for live heap and total heap for the methods in Health and Voronoi are the same. This happens because the analyzed methods are encharged of creating the data structures and there is no memory that can be garbage collected. In the remaining examples, the method `main` first calls the method `parseCmdLine` which creates a (linear) number of objects that do not escape to the `main` and, then, calls other methods that construct (and modify) a data structure which escapes to the `main`. The fact that some memory can be garbage collected explains that the live heap bounds are smaller than the total allocation. Tsp is interesting because some auxiliary `Double` objects are created during the execution of the methods `uniform` and `median` which do not escape from such methods and hence the difference between the live bound and the total allocation is bigger.

Benchmark Power has a constant memory consumption. Its live bound is much smaller than the total allocation because many objects are created by the constructor of `Lateral` which become unreachable and hence can be garbage collected. In the examples Mst and Em3d, most of the memory is allocated during the construction of a graph and all such memory cannot be garbage collected. As before, the live bound is slightly smaller because of the memory created by `parseCmdLine` which can be entirely garbage collected. Finally, the methods analysed for the benchmark Bh also create a number of auxiliary objects that can be garbage collected and the live heap bounds become tighter than the total allocation.

It is not easy to compare our upper bounds with those obtained by [5] since the cost models are different (we count sizes of objects as explained above while they count number of objects), they consider a region-based memory model while our analysis is developed for a scope-based model and, besides, for recursive methods (which occur in most benchmarks) [5] requires manual annotations that are not shown in their paper. In spite of these differences, as expected, our upper bounds coincide with those of [5] asymptotically (i.e., by ignoring the coefficients and constants).

An interesting experimentation that we plan to do for future work is to compare our upper bounds with actual observed values. This is however a rather complicated task. Note that it would require choosing particular inputs, and the memory consumption of the program could highly vary depending on such choice. We are confident about the positive results since, as we saw above, our UBs are coherent with those in [5], which in turn have already been compared to actual observed values.

8. Related Work

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis (see, e.g., [22]). Analysis of live heap space is different because it involves explicit analysis of all program states. Most of the work of memory estimation has been studied for functional languages. The work in [11] statically infers, by typing derivations and linear programming, linear expressions that depend on functional parameters while we are able to compute non-linear bounds (exponential, logarithmic, polynomial). The technique is developed for functional programs with an explicit deallocation mechanism while our technique is meant for imperative bytecode programs which are better suited for an automatic memory manager. The techniques proposed in [18, 17] consist in, given a function, constructing a new function that symbolically mimics the memory consumption of the former. Although these functions resemble our cost equations, their computed function has to be executed over a concrete valuation of parameters to obtain a memory bound for that assignment. Unlike our closed-form upper bounds, the evaluation of that function might not terminate, even if the original program does. Other differences with the work by Unnikrishnan et al. are that their analysis is developed for a functional language by relying on *reference counts* for the functional data constructed, which basically count the number of pointers to data and that they focus on particular aspects of functional languages such as tail call optimizations.

For imperative object-oriented languages, related techniques have been recently proposed. Previous work on heap space analysis [3] cannot be used to infer upper bounds on the maximum live memory as their cost relation systems are generated to accumulate cost, as explained in Sec. 3. Their refinement to infer escaped memory bounds is strictly less precise than ours as explained in Sec. 4, besides, there is no solution there to infer peak consumption. Later work improves [3] by taking garbage collection into account. In particular, for an assembly language, [8] infers memory resource bounds (both stack usage and heap usage) for low-level programs (assembly). The approach is limited to linear bounds, they rely on explicit disposal commands rather than on automatic memory management. In their system, dispose commands can be automatically generated only if alias annotations are provided. For a Java-like language, the approach of [5] infers upper bounds of the peak consumption by relying on an automatic memory manager as we do. They do not deal with recursive methods and are restricted to polynomial bounds. Besides, our approach is more flexible as regards its adaptation to other GC schemes (see Sec. 6). We believe that

our system is the first one to infer upper bounds on the live heap consumption which are not restricted to simple complexity classes.

9. Conclusions and Future Work

We have presented a general approach to automatic and accurate live heap space analysis for garbage-collected languages. As a first contribution, we propose how to obtain accurate bounds on the memory *escaped* from a method's execution by combining the total allocation performed by the method together with information obtained by means of escape analysis. Then, we introduce a novel form of *peak consumption cost relation* which uses the computed escaped memory bounds and precisely captures the actual heap consumption of programs' execution for garbage-collected languages. Such cost relations can be converted into closed-form upper bounds by relying on standard upper bound solvers. For the sake of concreteness, our analysis has been developed for object-oriented bytecode, though the same techniques can be applied to other languages with garbage collection. We first develop our analysis under a scoped-memory management which reclaims memory on method's return. The amount of memory required to run a method under such model can be used as an over-approximation of the amount required to run it in the context of an ideal garbage collection which frees objects as soon as they become dead. We also show how to approximate such ideal behavior with our analysis. For future work, we also plan to consider how to adapt our techniques to region based memory management [16, 6].

Finally, the idea developed in Sec. 5 can be used to estimate other (non accumulative) resources which require to consider the maximal consumption of several execution paths. For example, it can be used to estimate the maximal height of the frames stack as follows. Given a rule $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, where $b_{i_1} \ldots b_{i_k}$ are the calls in r, with $1 \leq i_1 \leq \cdots \leq i_k \leq n$ and $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, its corresponding equation would be

$$p(\bar{x}) = \max(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_1}(\bar{x}_{i_k})) \;\; \varphi_r$$

which takes the maximal height from all possible call chains. Each "1" corresponds to a single frame created for the corresponding call. Note that in this setting, tail call optimization can be also supported, by using an analysis that detects calls in tail position, and then replace their corresponding 1's by 0's. This is a subject for future work.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* ,TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.

[3] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[4] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.

[5] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.

[6] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In David F. Bacon and Amer Diwan, editors, *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 85–96. ACM, 2004.

[7] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 70–86, 2005.

[8] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.

[9] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM Press, 1978.

[10] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[12] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[13] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[15] Y. G. Park and B. Goldberg. Escape analysis on lists. In *PLDI*, pages 116–127, 1992.

[16] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[17] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In *Proc. of LCTES/OM*, pages 102–111. ACM, 2001.

[18] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85, 2003.

[19] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[20] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

[21] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

Constancy Analysis

Samir Genaim and Fausto Spoto

[1] CLIP, Technical University of Madrid (UPM), Spain
[2] Università di Verona, Italy
samir@clip.dia.fi.upm.es, fausto.spoto@univr.it

Abstract. A reference variable x is constant in a piece of code C if the execution of C does not modify the heap structure reachable from x. This information lets us infer purity of method arguments, an important ingredient during the analysis of programs dealing with dynamically allocated data structures. We define here an abstract domain expressing constancy as an abstract interpretation of concrete denotations. Then we define the induced abstract denotational semantics for Java-like programs and show how constancy information improves the precision of existing static analyses such as sharing, cyclicity and path-length.

1 Introduction

A major difference between pure functional/logic programming and imperative programming is that the latter uses destructive updates. That is, data structures are *mutable*: they are built *and later modified*. This can be both recognized as a superiority of imperative programming, since it allows one to write faster and simpler code, and as a drawback, since if two variables share a data structure then a destructive update to the data reachable from one variable may affect the data reachable from the other. This often leads to subtle programming bugs.

It is hence important to control what a method invocation modifies. Some methods do not modify the data structures reachable from their parameters. Others only modify those reachable from some but not all parameters. Namely, some parameters are *constant* or *read-only*, others may be modified. If all parameters of a method are constant, the method is *pure* [10]. Knowledge about purity is important since pure methods can be invoked in any order, which lets compilers apply aggressive optimizations; pure methods can be used in program assertions [7]; they can be skipped during many static analyses or more precisely approximated than other methods. This results in more efficient and more precise analyses. For instance, sharing analysis [11] can safely assume that sharing is not introduced during the execution of a pure method. In general, all static analyses tracking properties of the heap benefit from information about purity.

For these reasons, software specification has found ways of expressing purity of methods and constant parameters. The notable example is the Java Modeling Language [7], which uses the `assignable` clause to specify those heap positions that might be mutated during the execution of a method. Those clauses are manually provided and used by many static analyzers, such as ESC/Java [6]

and ChAsE [4]. However, those tools do not verify the correctness of the user-provided `assignable` clauses, or use potentially incorrect verification techniques. A formally correct verification technique is defined in [14], but has never been implemented. In [10] a formally correct analysis for purity is presented, it is based on a preliminary points-to and escape analysis, and an implementation exists and has been applied to some small size examples. In [8] a correct and precise algorithm for statically inferring the reference immutability qualifiers of the Javari language has been presented. The algorithm has been implemented in the Javarifier tool.

In this paper, we investigate an alternative technique aiming at determining which parameters of a method are constant. We use abstract interpretation [5] and perform a static analysis over the reduced product of the sharing domain in [11] (the *sharing component*) and a new abstract domain expressing the set of variables bound to data structures mutated during the execution of a piece of code (the *purity component*). The use of reduced product is justified since the sharing component helps the purity component during a destructive update, by identifying which variables share the updated data structure and hence lose their purity; conversely, the sharing component uses the purity component during method calls, since only variables sharing with non-pure parameters of a method m can be made to share during the execution of m.

Our technique is sometimes less precise than [10], since it does not use the field names (i.e., we do not keep information on which field has been updated, but rather that a field has been updated). However, it is implemented in a completely flow-sensitive and context-sensitive fashion, which improves its precision. Moreover, it is expressed in terms of Boolean formulas implemented through binary decision diagrams, resulting in fast analyses scaling to quite big programs. Our contributions are hence: (1) a definition of the reduced product of sharing and purity; (2) its application to large programs; (3) a comparison of the precision of sharing analysis alone with that of sharing analysis in reduced product with purity; and (4) an evaluation of the extra precision induced by the purity information during static analyses tracking properties of the heap, namely, possible cyclicity of data structures [9] and path-length of data structures [13].

The paper is organized as follows: Section 2 defines the syntax and semantics of a simple Object-Oriented language; Section 3 develops our constancy analysis for that language; Section 4 provides an experimental evaluation.

2 Our Simple Object-Oriented Language

This section presents syntax and denotational semantics of a simple Object-Oriented language that we use through the paper. Its commands are normalized versions of corresponding Java commands: the language supports reference and integer types; in method calls, only syntactically distinct variables can be actual parameters, which is a form of normalization and does not prevent them from being bound to shared data-structures at run-time; in assignments, the left hand side is either a variable or the field of a variable; Boolean conditions are kept

generic, they are conditions that are evaluated to either $true$ or $false$; iterative constructs, such as the `while` loop, are not supported since they can be implemented through recursion. These assumptions are only for the sake of clear and simple presentation and can be relaxed without affecting subsequent results. A program has a set of *variables* $\mathcal{V}$ (including *out* and *this*) and a finite poset of *classes* $\mathbb{K}$. The *commands* of the language are

$$\begin{aligned} com ::= \quad & v := c \mid v := w \mid v := \texttt{new}\ \kappa \mid v := w + z \mid v := w.\texttt{f} \mid v.\texttt{f} := w \mid \\ & v := v_0.\texttt{m}(v_1, \ldots, v_n) \mid \texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2 \mid com_1; com_2 \end{aligned}$$

$v, w, z, v_0, v_1, \ldots, v_n \in \mathcal{V}$ are distinct variables, $c \in \mathbb{Z} \cup \{\mathsf{null}\}$, $\kappa \in \mathbb{K}$ and e is a Boolean expression. The signature of a method $\kappa.\texttt{m}(t_1, \ldots, t_p){:}t$ refers to a method called `m` expecting p parameters of type $t_1, \ldots, t_p \in \mathbb{K} \cup \{\mathsf{int}\}$, respectively, returning a value of type t and *defined* in class κ with a statement

$$t\ \texttt{m}(w_1{:}t_1, \ldots, w_n{:}t_n)\ \texttt{with}\ \{w_{n+1}{:}t_{n+1}, \ldots, w_{n+m}{:}t_{n+m}\}\ \texttt{is}\ com,$$

where $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ are distinct, not in $\{out, this\}$ and have *type* $t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m} \in \mathbb{K} \cup \{\mathsf{int}\}$, respectively. Variables $w_1, \ldots, w_n$ are the *formal parameters* of the method and $w_{n+1}, \ldots, w_{n+m}$ are its *local variables*. The method also uses a variable *out* of type t to store its *return value*. For a given method signature $m = \kappa.\texttt{m}(t_1, \ldots, t_p) : t$, we define $m^b = com$, $m^i = \{this, w_1, \ldots, w_n\}$, $m^o = \{out\}$, $m^l = \{w_{n+1}, \ldots, w_{n+m}\}$ and $m^s = m^i \cup m^o \cup m^l$. Classes might declare fields of type $t \in \mathbb{K} \cup \{\mathsf{int}\}$.

We use a denotational semantics, hence compositional, in the style of [15]. However, we use a more complex notion of state, which assumes an infinite set of *locations*. Basically, a state is a pair which consists of a frame and a heap, where a frame maps variables to values and a heap maps locations to objects. Note that since we assume a denotational semantics, a state has a single frame, rather than an activation stack of frames as it is required in operational semantics. We let $\mathbb{L}$ denote an infinite set of *locations*, and let $\mathbb{V}$ denotes the set of *values* $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$. A *frame* over a finite set of variables V is a mapping that maps each variable in V into a value from $\mathbb{V}$; a *heap* is a partial map from $\mathbb{L}$ into *objects*. An object is a pair that consists of its class tag κ and a frame that maps its *fields* (identifiers) into values from $\mathbb{V}$; we say that it *belongs to* class κ or *has* class κ. Given a class κ, we assume that $newobj(\kappa)$ return a new object where its fields are initialized to 0 or depending on their types. If ϕ is a frame and $v \in V$, then $\phi(v)$ is the value of variable v. If μ is a heap and $\ell \in \mathbb{L}$, then $\mu(\ell)$ is the object bound in μ to ℓ. If o is an object, then $o.tag$ denotes its class and $o.\phi$ denotes its frame; if f is a field of o, then sometimes we use $o.f$ to refer to (or set) its value instead of going through its frame.

Definition 1 (computional state). *Let V denotes the set of variables in scope at a given program point p. The set of possible* states *at p is*

$$\Sigma_V = \left\{ \langle \phi, \mu \rangle \;\middle|\; \begin{array}{l} 1.\ \phi \text{ is a frame over } V \text{ and } \mu \text{ is a heap} \\ 2.\ \mathrm{rng}(\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \\ 3.\ \forall \ell \in \mathrm{dom}(\mu).\ \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \end{array} \right\}$$

Conditions 2 *and* 3 *guarantee the absence of dangling pointers. Given* $\sigma = \langle \phi, \mu \rangle \in \Sigma_V$*, we use* ϕ_σ *and* μ_σ *to refer to its frame and heap respectively.* □

Now we define the notion of *Denotations* which are the input/output semantics of a piece of code. Basically they are mappings from states to states which describe how the input state is changed when the corresponding code is executed. *Interpretations* are a special case of denotations which provide a denotation for each method in terms of its input and output variables.

Definition 2. *A* denotation δ *from* V *to* V' *is a partial function from* Σ_V *to* $\Sigma_{V'}$*. We often refer to* $\delta(\sigma) = \sigma'$ *as* $(\sigma, \sigma') \in \delta$*. The set of denotations from* V *to* V' *is* $\Delta(V,V')$*. An* interpretation ι *maps methods to denotations and is such that* $\iota(m) \in \Delta(m^i, m^i \cup m^o)$ *for each method* $m = \kappa.\mathtt{m}(t_1, \ldots, t_p) : t$ *in the given program. The set of all possible interpretations is written as* $\mathbb{I}$*.* □

The denotational semantics associates a denotation to each command of the language. Let V denotes a set of variables. Let $\iota \in \mathbb{I}$. We define the *denotation for commands* $\mathcal{C}^\iota_V[\![\_]\!] : com \mapsto \Delta(V,V)$, as their input/output behaviour:

$$
\begin{aligned}
\mathcal{C}^\iota_V[\![v{:=}c]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto c]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}\mathtt{new}\ \kappa]\!] &= \{(\sigma, \sigma[\mu_\sigma(\ell) \mapsto newobj(\kappa)]) \mid \sigma \in \Sigma_V,\ \ell \notin \mathrm{dom}(\mu_\sigma)\} \\
\mathcal{C}^\iota_V[\![v{:=}w + z]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w) + \phi_\sigma(z)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w.f]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \mu_\sigma\phi_\sigma(w).f]) \mid \sigma \in \Sigma_V, \phi_\sigma(w) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V[\![v.f{:=}w]\!] &= \{(\sigma, \sigma[\mu_\sigma\phi_\sigma(v).f \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V, \phi_\sigma(v) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V\left[\!\!\left[\begin{array}{r}\mathtt{if}\ e\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2\end{array}\right]\!\!\right] &= \begin{array}{l}\{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \mid \sigma \models e \approx true\} \cup \\ \{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_2]\!] \mid \sigma \models e \approx false\}\end{array} \\
\mathcal{C}^\iota_V[\![com_1; com_2]\!] &= \{(\sigma, \sigma'') \mid (\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \wedge (\sigma', \sigma'') \in \mathcal{C}^\iota_V[\![com_2]\!]\}
\end{aligned}
$$

The denotation for a method call $\mathcal{C}^\iota_V[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!]$ should consider the denotation $\iota(m)$ (where m is the called method) and extend it to fit in the calling scope and update the variable v. Assume the method signature is $\mathtt{m}(t_1, \ldots, t_p){:}t$, and that we have a lookup procedure $\mathcal{L}$ that, for any given $\sigma \in \Sigma_V$, fetches the actual method that is called depending on the run-time class of v_0. Then the method call denotation is defined as follows:

$$
\left\{ (\sigma, \langle \phi_\sigma[v \mapsto \phi''_\sigma(out)], \mu''_\sigma \rangle) \;\middle\|\; \begin{array}{l} 1.\ \sigma \in \Sigma_V, \phi_\sigma(v_0) \in \mathrm{dom}(\mu_\sigma); \\ 2.\ m = \mathcal{L}(v_0, \sigma, \mathtt{m}(t_1, \ldots, t_p){:}t); \\ 3.\ (\sigma', \sigma'') \in \iota(m); \\ 4.\ \mu_\sigma \equiv \mu'_\sigma, \forall 0 \leq i \leq p.\ \phi_\sigma(v_i) = \phi'_\sigma(w_i) \end{array} \right\}
$$

The concrete denotational semantics of a program is the least fixpoint of the following transformer of interpretations [3].

Definition 3 (Denotational semantics). *The denotational semantics of a program* P *is defined as* $\bigcup_{i \geq 0} T^i_P(\iota_0)$ *, i.e. the* least fixed point *of* T_P *where* T_P *is:*

$$T_P(\iota) = \left\{ (m, X) \;\middle\|\; \begin{array}{l} 1.\ m \in P \\ 2.\ \sigma \in \Sigma_{m^s}, \forall v \in m^l.\ \phi_\sigma(v) = 0 \text{ or } \phi_\sigma(v) = \texttt{null} \\ 3.\ X = \{(\sigma|_{m^i}, \sigma'|_{m^i \cup m^o}) \mid (\sigma, \sigma') \in \mathcal{C}^\iota_{m^s}[\![m^b]\!]\} \end{array} \right\}$$

and $\iota_0 = \{(m, \emptyset) \mid m \in P\}$ *and* $\forall \iota_1, \iota_2 \in \mathbb{I}$ *the union* $\iota_1 \cup \iota_2$ *is defined as* $\{(m, X_1 \cup X_2) \mid m \in P, (m, X_1) \in \iota_1, (m, X_2) \in \iota_2\}$ □

3 Constancy Analysis

We want to design an analysis to infer definite information about constant data structures. This can be done by tracking data structures that are not modified (definite information), or by tracking data structures that might be modified (may information). We follow the latter approach as we believe it easier. In addition, we want to analyze methods in a context independent way, and later adapt the result to any calling context.

Example 1. Consider the following method:

```
A m(x:A, y:A) with {} is y:=y.next; x.next:=y; out:=y;
```

The only command that might modify the heap structure is "`x.next:=y`". Note that "`y:=y.next`" does not affect the heap structure but rather changes the heap location stored in `y`. This method might be called in different contexts where the actual parameters: (1) do not have any common data structure; or (2) have a common data structure. In the first case, "`x.next:=y`" might modify only the data structure pointed by the first argument. In the second case, it might modify a common data structure for `x` and `y`, and therefore we say that both arguments might be modified. We describe this behaviour by the Boolean formula $\check{x} \wedge (\check{y} \leftrightarrow x\check{}y)$, which is interpreted as: (1) in any calling context, the data structure the first argument points to when the method is called might be modified by the method (expressed by $\check{x}$); and (2) the data structure that the second argument points to when the method is called, might be modified by the method (expressed by $\check{y}$) iff x and y might share a data structure when the method is called (expressed by $x\check{}y$).

□

We define now the set of reachable heap locations from a given reference variable, which we need to define the notion of *constant heap structure*.

Definition 4 (reachable heap locations). *Let* μ *be a heap. The set of* locations reachable from $\ell \in \mathrm{dom}(\mu)$ *is* $L(\mu, \ell) = \cup\{L^i(\mu, \ell) \mid i \geq 0\}$ *where* $L^0(\mu, \ell) = \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L}$ *and* $L^{i+1}(\mu, \ell) = \cup\{\mathrm{rng}(\mu(\ell')) \cap \mathbb{L} \mid \ell' \in L^i(\mu, \ell)\}$. *The set of reachable heap locations from* v *in* $\sigma \in \Sigma_V$*, denoted* $L_V(\sigma, v)$*, is* $\{\phi_\sigma(v)\} \cup L(\mu_\sigma, \phi_\sigma(v))$ *if* $\phi_\sigma(v) \in dom(\mu_\sigma)$*; and the empty set otherwise.* □

Definition 5 (constant reference variable). *A reference variables $v \in V$ is constant with respect to a denotation δ, denoted $\mathsf{c}(v, \delta)$, iff for any $(\sigma_1, \sigma_2) \in \delta$ all locations in $L_V(\sigma_1, v)$ are constant with across δ, namely $\forall \ell \in L_V(\sigma_1, v)$, $\mu_{\sigma_1}(\ell)$ and $\mu_{\sigma_2}(\ell)$ have the same class tag and agree on their* reference *field values.* □

The definition above considers modifications of fields of reference type only. The reason for concentrating on reference fields is that we have developed this analysis for a specific need which requires tracking updates only in the shape of the data structure (see Section 4). Tracking updates of integer fields can simply done by modifying the above definition to consider those updates. In what follows, a modification of a variable stands for a modification of the shape of the heap structure reachable from that variable.

Definition 6 (common heap location). *$x, y \in V$ have a common heap location (share) in a state $\sigma \in \Sigma_V$ if and only if $L_V(\sigma, x) \cap L_V(\sigma, y) \neq \emptyset$* □

We define now an abstract domain which captures a set of variables that *might* be modified by a concrete denotation.

Definition 7 (update abstract domain). *The update abstract domain U_V is a partial order $\langle \wp(V), \subseteq \rangle$. Its concretization function $\gamma_V{:}\mathsf{U}_V \to \Delta(V,V')$ is defined as $\gamma_V(X) = \{\delta \mid \forall v \in V.\ \neg \mathsf{c}(v, \delta) \to (v \in X)\}$.* □

As we have seen in Example 1, information about possible sharing between variables is important for a precise constancy analysis. There are many ways for inferring such information. Here, we use the pair-sharing domain [11]. Moreover, constancy information improves the precision of method calls in pair sharing analysis. This is because the execution of a method m can introduce sharing between non-constant parameters only. Hence we design an analysis over the (reduced) product of the update domain U_V and of the pair-sharing domain SH_V, denoted by $\mathsf{SH}{\times}\mathsf{U}_V$. Informally, the pair sharing domain abstracts an element $s \in \wp(\Sigma_V)$ to a set sh of symmetric pairs of the form (x, y) where $x, y \in V$. If $(x, y) \in sh$ then x and y *might* share in s, and if $(x, y) \notin sh$ then they cannot share, so that if $(x, x) \notin sh$ then x must be `null` in s. In what follows, instead of saying *might share* we simply say share.

Figure 1 defines abstract denotations for our simple language over $\mathsf{SH}{\times}\mathsf{U}_V$. They are Boolean functions corresponding to the elements of $\mathsf{SH}{\times}\mathsf{U}_V$. For a piece of code C, the Boolean variables:

- $x\check{}y$ and $x\hat{}y$ indicate if x and y *share* before and after executing C, respectively. Since pair sharing is symmetric, $x\check{}y$ and $y\check{}x$ are equivalent Boolean variables; and
- $\check{x}$ and $\hat{x}$ indicate if x is modified with respect to its value before and after C (by the program execution), respectively.

Each abstract denotation is defined in terms of a Boolean function $\varphi \wedge \psi$, where φ propagates (forward) *sharing* information and ψ propagates (backwards) *update* information. In what follows we explain the meaning of each abstract denotation:

$$
\begin{aligned}
\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \\
&\quad -\varphi_1 = (\wedge\{\neg x\hat{\cdot}v \mid x \in V\}) \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \wedge \varphi_2 \\
&\quad -\varphi_1 = \wedge\{x\hat{\cdot}v \leftrightarrow x\check{\cdot}w \mid x \in V\backslash\{v\}\} \\
&\quad -\varphi_2 = w\check{\cdot}w \leftrightarrow v\hat{\cdot}v \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{new}\ \kappa]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge v\hat{\cdot}v \wedge \varphi_1 \\
&\quad -\varphi_1 = (\wedge\{\neg x\hat{\cdot}v \mid x \in V \setminus \{v\}\}) \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}w.f]\!] &= \mathcal{A}^{\iota}_{V}[\![v{:=}w]\!] \\
\mathcal{A}^{\iota}_{V}[\![v.f{:=}w]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \wedge\{x\hat{\cdot}y \leftrightarrow x\check{\cdot}y \vee (x\check{\cdot}w \wedge y\check{\cdot}v) \mid x, y \in V\} \\
&\quad -\psi = \{\check{x} \leftrightarrow v\check{\cdot}x \vee \hat{x} \mid x \in V\} \\
\mathcal{A}^{\iota}_{V}[\![\texttt{if}\ e\ \ldots]\!] &= \mathcal{A}^{\iota}_{V}[\![c_1]\!] \vee \mathcal{A}^{\iota}_{V}[\![c_2]\!] \\
\mathcal{A}^{\iota}_{V}[\![c_1; c_2]\!] &= \mathcal{A}^{\iota}_{V}[\![c_1]\!] \circ \mathcal{A}^{\iota}_{V}[\![c_2]\!] \\
\mathcal{A}^{\iota}_{V}[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!] &= \phi \wedge \varphi \wedge \psi \\
&\quad \phi_m = \vee\{\iota(m) \mid m \text{ might be called }\} \\
&\quad \phi = \phi_m[s_i \mapsto v_i, \mathit{out} \mapsto v, \mathit{this} \mapsto v_0] \\
&\quad \varphi = \wedge\{x\hat{\cdot}y \leftrightarrow x\check{\cdot}y \vee \varphi_1 \mid x, y \in V\backslash\{v_0, \ldots, v_p\}\} \\
&\quad \varphi_1 = \vee\{(x\check{\cdot}v_i \wedge y\check{\cdot}v_j \wedge v_i\hat{\cdot}v_j \wedge (\check{v}_i \vee \check{v}_j)) \mid i, j \in \{0, \ldots, p\}\} \\
&\quad \psi = \psi_1 \wedge (\check{v} \leftrightarrow \psi_3 \vee \psi_2(v)) \\
&\quad \psi_1 = \wedge\{\check{x} \leftrightarrow \hat{x} \vee \psi_2(x) \mid x \in V\backslash\{v, v_0, \ldots, v_p\}\} \\
&\quad \psi_2(x) = \vee\{(x\check{\cdot}v_i \wedge \check{v}_i) \mid i \in \{0, \ldots, p\}\} \\
&\quad \psi_3 = \{x\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}
\end{aligned}
$$

Fig. 1. Abstract Denotations over $\mathsf{SH}{\times}\mathsf{U}_V$

- $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); and nothing can share with v after C (φ_1). (**U**) $x \in V\backslash\{v\}$ is modified before C iff it is modified after C, and v is modified before C iff it shares with some y before C and y is modified after C.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); since v becomes an alias for w then v can share with $x \in V\backslash\{v\}$ after C iff x shares with w before C (φ_1); and v can share with itself after C (i.e., not `null`) iff w shares with itself before C (φ_2). (**U**) the same as for "$v{:=}$`null`".
- $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{new}\ \kappa]\!]$: the same as $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!]$ except that v shares with itself after executing the statement.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}w.f]\!]$: the same as $\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!]$ since the analysis is field insensitive.
- $\mathcal{A}^{\iota}_{V}[\![v.f{:=}w]\!]$: (**SH**) $x, y \in V$ share after C iff before C, they shared or x shared with w and y with v; (**U**) $x \in V$ is modified before C, iff it shares with v before C or x is modified after C.
- $\mathcal{A}^{\iota}_{V}[\![\texttt{if}\ e\ \ldots]\!]$: combines the branches through logical or.

- $\mathcal{A}^\iota_V[\![c_1; c_2]\!]$: combines $\mathcal{A}^\iota_V[\![c_1]\!]$ and $\mathcal{A}^\iota_V[\![c_2]\!]$. This is simply done by matching the output variables of the first denotation with the input variables of the second denotation.
- $\mathcal{A}^\iota_V[\![v{:=}v_0.\mathtt{m}(v_1,\ldots,v_p)]\!]$: (1) First we fetch the abstract denotations of all methods that might be called, and we combine them through logical or into ϕ_m; (2) Assuming that the method denotations use $s_i \neq v_i$ for the i-th formal parameter, we rename all sharing information by changing each s_i into v_i and *out* into v. We get ϕ. (3) We add sharing information for variables which are not in $V\backslash\{v, v_0, \ldots, v_p\}$. The sharing component φ states that x and y might share after the call iff they shared before (i.e. $x\check{\cdot}y$) or they shared with arguments v_i and v_j where v_i and v_j share after the call, and either v_i or v_j has been modified (expressed by φ_1); (4) We add the constancy information which states that $x \in V\backslash\{v\}$ is modified before iff it is modified after, or if it shares with a variable that is modified by the method. For v it is a bit different since we exclude the case that if v is modified after then it is modified before, since we possibly assign to it a new reference.

The abstract denotation for a method:

$$t\ \mathtt{m}(w_1{:}t_1,\ldots,w_n{:}t_n)\ \mathtt{with}\ w_{n+1}{:}t_{n+1},\ldots,w_{n+m}{:}t_{n+m}\ \mathtt{is}\ com,$$

is then defined as $\phi_m = \exists V'.\ \mathcal{A}^\iota_V[\![com]\!] \wedge \varphi_1 \wedge \varphi_2$ where:

- $S = \{s_1, \ldots, s_n\}$ such that $S \cap m^s = \emptyset$, and $V = m^s \cup S$
- $\varphi_1 = \{\neg x\check{\cdot}y \mid x \in m^l \cup \{out\}, y \in m^s\}$
- $\varphi_2 = \{s_i\check{\cdot}x \leftrightarrow w_i\check{\cdot}x \mid 1 \leq i \leq n, x \in m^i\}$
- $V' = \{x\check{\cdot}y, x\hat{\cdot}y, \check{x}, \hat{x} \mid x \notin S \cup \{this, out\}, y \in V\} \cup \{o\check{u}t\}$

The idea is that we: (1) extend m^l to V in order to include shallow variable s_i for each method argument w_i; (2) compute $\mathcal{A}^\iota_V[\![com]\!]$; (3) add φ_1 which indicates that local variables are initialized to `null`; (4) add φ_2 which creates the connection between the shallow variables and the actual parameters; (5) eliminate all local information by removing the Boolean variables V'. The abstract denotational semantics can be then defined similar to the concrete one in Definition 3, where the initial method summaries are $false$ ans summaries are combined (during the fixpoint iterations) using the logical or $\vee$.

Example 2. Applying the above abstract semantics to the method defined in Example 1 results in a Boolean formula whose constancy component is $(t\check{h}is \leftrightarrow t\hat{h}is) \wedge \check{x} \wedge (\check{y} \leftrightarrow (x\check{\cdot}y \vee \hat{y}))$. For simplicity we ignore the part of ϕ_m that talks about sharing.

4 Experiments

We show here some experiments with our domain for sharing and constancy analysis. They have been performed with the Julia analyzer [12] on a Linux

Program	M	Sharing		Non-Cyclicity	
		T	P	T	P
`JLex`	446	1595 (2324)	34.30% (34.84%)	506 (415)	34.03% (35.21%)
`JavaCup`	933	5707 (6486)	22.24% (23.76%)	853 (953)	59.23% (76.13%)
`Kitten`	2131	20976 (27824)	17.90% (19.11%)	2538 (3177)	36.34% (41.13%)
`jEdit`	3206	47408 (49356)	21.12% (21.28%)	4969 (5963)	43.49% (47.50%)
`Julia`	4028	79199 (129562)	9.71% (10.25%)	8014 (12018)	33.40% (38.17%)

Fig. 2. The effect of the purity component on Sharing and Non-Cyclicity. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

machine based on a 64 bits dual core AMD Opteron processor 280 running at 2.4Ghz, with 2 gigabytes of RAM and 1 megabyte of cache, by using Sun Java Development Kit version 1.5. All programs have been analyzed including all library methods that they use inside the `java.lang.*` and `java.util.*` hierarchies.

Figure 2 compares sharing analysis alone with sharing analysis in reduced product with constancy (Section 3), and its effect on non-cyclicity analysis [9]. In each column, numbers in parentheses correspond to the analysis using the reduced product. For each program, it reports the number of methods analyzed, including the libraries, and time and precision of the corresponding analysis with and without constancy. For sharing, the precision is the amount of pairs of variables of reference type that are proved not to share at the program points preceding the update of an instance field, the update of an array element or a method call. This is sensible since there is where sharing analysis is used by subsequent analyses. That figure suggests that the constancy component slightly improves the precision of sharing analysis. However, the importance of constancy is shown when we consider its effects on a static analysis that uses constancy information. This is the case of non-cyclicity analysis, which finds variables bound to non-cyclical data structures [9]. Figure 2 shows that the computation of cyclicity analysis after a simple sharing analysis leads to less precise results than the same computation after a sharing and constancy analysis. Here, precision is the number of field accesses that read the field of a non-cyclical object. This is sensible since there is where non-cyclicity is typically used.

The importance of constancy analysis becomes more apparent when it supports a static analysis that uses constancy, sharing and cyclicity information. This is the case of *path-length* [13]. It approximates the length of the maximal path of pointers one can follow from each variable. This information is the basis of a termination [1] and resource bound analyses [2] for programs dealing with dynamic data structures. Figure 3 shows the effects of constancy on path-length and termination analysis (available in [12]) of a set of small programs that do not use libraries except for `java.lang.Object`. Times are in milliseconds and precision is the number of methods proved to terminate. Constancy information

Program	M	T	P
`Init`	10	102 (140)	8 (8)
`List`	11	624 (512)	6 (11)
`Diff`	5	6668 (9040)	5 (5)
`Hanoi`	7	548 (868)	7 (7)
`BTree`	7	306 (415)	6 (7)
`BSTree`	10	234 (273)	9 (10)
`Virtual`	11	357 (418)	10 (11)
`ListInt`	11	767 (507)	6 (11)

Program	M	T	P
`Nested`	4	324 (447)	4 (4)
`Double`	5	270 (268)	5 (5)
`FactSum`	6	169 (178)	6 (6)
`Sharing`	7	309 (501)	6 (7)
`Factorial`	5	102 (196)	5 (5)
`Ackermann`	5	1308 (1732)	5 (5)
`BubbleSort`	5	871 (951)	5 (5)
`FactSumList`	8	278 (703)	7 (8)

Fig. 3. The effect of the purity information on Termination analysis. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

results in proving that all terminating methods terminate (only 2 methods of `Init` are not proved to terminate: they actually diverge). Without constancy information, many terminating methods are not proved to terminate.

These experiments suggest that constancy information contributes to the precision of sharing, cyclicity, path-length and hence termination analysis. Computing constancy information with sharing requires more time than computing sharing alone (Figure 2). Performing other analyses by using the constancy information increases the times further (Figures 2 and 3). Nevertheless, this is justified by the extra precision of the results.

5 Acknowledgments

This work of Samir Genaim was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project, and a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science, Oslo, Norway, June 2008. Springer-Verlag, Berlin. To appear.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19-20:149–197, 1994.
4. N. Cataño and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2003.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center, 1998.
7. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML. Technical Report 96-06p, Iowa State University, 2001.
8. Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.
9. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006.
10. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In R. Cousot, editor, *Proc. of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215, Paris, France, 2005. Springer.
11. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. of Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005.
12. F. Spoto. The JULIA Static Analyser. `profs.sci.univr.it/~spoto/julia`, 2008.
13. F. Spoto, P. M. Hill, and E. Payet. Path-Length Analysis for Object-Oriented Programs. In *Proc. of Emerging Applications of Abstract Interpretation*, Vienna, Austria, March 2006. `profs.sci.univr.it/~spoto/papers.html`.
14. F. Spoto and E. Poll. Static Analysis for JML's assignable Clauses. In G. Ghelli, editor, *Proc. of FOOL-10, the 10th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New Orleans, Louisiana, USA, January 2003. ACM Press. Available at `www.sci.univr.it/~spoto/papers.html`.
15. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models

Mark Marron[1], Manuel Hermenegildo[1,2], Deepak Kapur[1], and Darko Stefanovic[1]

[1]University of New Mexico
{marron,kapur,darko}@cs.unm.edu
[2] Technical University of Madrid and IMDEA-Software
herme@fi.upm.es

Abstract. The performance of heap analysis techniques has a significant impact on their utility in an optimizing compiler. Most shape analysis techniques perform interprocedural dataflow analysis in a context-sensitive manner, which can result in analyzing each procedure body many times (causing significant increases in runtime even if the analysis results are memoized). To improve the effectiveness of memoization (and thus speed up the analysis) *project*/*extend* operations are used to remove portions of the heap model that cannot be affected by the called procedure (effectively reducing the number of different contexts that a procedure needs to be analyzed with). This paper introduces *project*/*extend* operations that are capable of accurately modeling properties that are important when analyzing non-trivial programs (sharing, nullity information, destructive recursive functions, and composite data structures). The techniques we introduce are able to handle these features while significantly improving the effectiveness of memoizing analysis results (and thus improving analysis performance). Using a range of well known benchmarks (many of which have not been successfully analyzed using other existing shape analysis methods) we demonstrate that our approach results in significant improvements in both accuracy and efficiency over a baseline analysis.

1 Introduction

Recent work on shape analysis techniques [25,28,1,14,15,9,8] has resulted in a number of techniques that are capable of accurately representing the properties (connectivity, interference, and shape) that are needed for a range of optimization and parallelization applications. However, the computational cost of performing these analyses has limited their applicability. A significant component of the analysis runtime is due to the need to perform a context-sensitive interprocedural analysis, where each procedure body may be analyzed multiple times (once for each different calling context).

The practice of using a memo-table to avoid recomputing analysis results and the use of a *project* operation to remove portions of the heap that cannot affect or be affected by the called procedure are standard techniques for minimizing the number of times each function needs to be analyzed during interprocedural dataflow analysis [2,17,16,19]. The two major goals of the *project* operation are improving the effectiveness of memoizing analysis results by removing portions of the heap that could cause spurious inequalities

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 245–259, 2008.

between calling contexts and preventing the loss of precision that occurs when recursive procedures use a summary representation for multiple out-of-scope references (e.g. local reference variables with the same name but that exist in different call frames).

The *project* operation for heap models and the utility of locality axioms have been analyzed in a number of papers [22,21,7,12,4]. These techniques use variations on the notion of a *frame rule* as presented in [11,20] and identify a number of features of the *project* operation that are of particular importance for interprocedural analysis using heap domains. A major distinction is made between the projection operation in *cutpoint*-free cases, where there are no pointers that cross from a section of the heap that is *unreachable* from the procedure arguments into a section of the heap that is *reachable* from the procedure arguments, and cases where such pointers may exist.

This paper presents a method for using cutpoints to support interprocedural heap analysis. We then use the technique to quickly analyze (10's of seconds) programs that are larger (by a factor of 2-4) and more varied (in terms of data structures and algorithms) than any other analysis technique to date. Our first contribution is the reformulation of the project/extend operations in [21] so that they can be used in a graph based (as opposed to an access path based) heap model which allows us to use a very compact and efficient representation of heap connectivity. Our second contribution is the extension of the original approach to handle two classes of programatic events that are critical to analyzing real world programs, analyzing programs that involve non-trivial sharing and composite data structures [1,15] and propagating nullity test information from callee to caller scope. Finally we use the results of the heap analysis to drive the parallelization of a range of benchmarks (several of which have not been successfully analyzed/parallelized using shape information) achieving an average parallel speedup of 1.69 on a dual-core machine.

2 Example Code

To develop intuition about the mechanism and purpose of *project*/*extend* operations we look at a simple function (Figure 1) that illustrates the basic functioning of the *project*/*extend* operations and the propagation of nullity information from the callee to the caller scope. Our lists are made of objects of *type* `LNode`, each `LNode` object has two fields, a `nx` field which refers to the next element in the list and a field `f` which stores a boolean.

```
LNode LInit(LNode l)
  if(l == null)
    return;

  tin = l.nx;
  LInit(tin);
  l.f = true;
```

Fig. 1. Recursive List Initialize

Accurately analyzing the initialization method (LInit) requires the analysis to propagate information inferred about cutpoints in the callee scope back into the caller scope. If the analysis is unable to use the `l == null` test in the callee scope to infer that `l.nx` is `null` in the caller scope then the analysis will not be able to infer that after the method returns the argument list is either `null` or must have the `true` value in all the `f` fields.

3 Heap Model

We model the concrete heap as a labeled, directed multi-graph (V,E) where each vertex $v \in V$ is an object in the store or a variable in the environment, and each labeled directed edge $e \in E$ represents a pointer between objects or a reference from a variable to an object. Each edge is given a label that is an identifier from the program, an edge $(a,b) \in E$ labeled with p, we use the notation $a \xrightarrow{p} b$ to indicate that a points to the object b via the field name (or identifier) p.

A *region* of memory $\mathfrak{R}$ is a subset of the objects in memory, with all the pointers that connect these objects and all the cross-region pointers that start or end at an object in this region. Formally, let $C \subseteq V$ be a subset of objects, and let $P_i = \{p \mid \exists a,b \in C, a \xrightarrow{p} b\}$ and $P_c = \{p \mid \exists a \in C, x \notin C, a \xrightarrow{p} x \vee x \xrightarrow{p} a\}$ be respectively the set of internal and cross-region pointers for C. Then a region is the tuple (C,P_i,P_c). For a region $\mathfrak{R} = (C,P_i,P_c)$ and objects $a,b \in C$, we say a and b are *connected* in $\mathfrak{R}$ if they are in the same weakly-connected component of the graph (C,P_i). Objects a and b are *disjoint* in $\mathfrak{R}$ if they are in different weakly-connected components of the graph.

3.1 Abstract Heap Model

The underlying abstract heap domain is a graph where each node represents a region of the heap or a variable and each edge represents a set of pointers or a variable target. The nodes and edges are augmented with additional instrumentation predicates. The abstract domain evaluates the predicates using a *3-valued* semantics: predicates are either definitely true, definitely false, or unknown [25]. Our analysis tracks the following set of instrumentation predicates. Our choice of predicates is influenced by common predicates tracked in previous papers on shape analysis [5,24,28,20].

Types. For each type t in the program, there is an instrumentation predicate (also written t) that is true at a concrete heap node if any concrete object represented by the node may have type t.

Linearity. Each abstract node has a *linearity* that represents whether it represents at most one concrete node (linearity 1) or any set of 0 or more concrete nodes (written #).

Abstract Layout. To track the connectivity and shape of the region a node abstracts, the analysis uses *abstract layout* predicates *Singleton*, *List*, *Tree*, *MultiPath*, or *Cycle*. The *Singleton* predicate states that there are no pointers between any of the objects represented by an abstract node. The *List* predicate is similar to the inductive *List* predicate

in separation logic [20]. The other predicates correspond to the definitions for Trees, Dags, and Cycles in the literature, for the formal definitions see [14].

Interference. The heap model uses two properties to track the potential that two references can reach the same memory location in the region that a node represents.

The first property is for references that are represented by different edges in the heap model. Given the concretization function γ and two edges e_1, e_2 that are incoming edges to the node n, the predicate that defines *inConnected* in the abstract domain is: e_1, e_2 are *inConnected* with respect to n if it is possible that $\exists r_1 \in \gamma(e_1) \wedge \exists r_2 \in \gamma(e_2) \wedge \exists a, b \in \gamma(n)$ s.t. (r_1 refers to a) $\wedge$ (r_2 refers to b)$\wedge$ (a, b connected). For improved precision we also track *may* and *must* aliasing (e_1, e_2 are *inConnected* and $a = b$) between the references the edges abstract (*must* aliasing is only meaningful if the edge represents a single references, see [15] for an approach that generalizes *must-aliasing* to sets of references).

The second property is for the case where the references are represented by the same edge. To model this the *interfere* property is introduced. An edge e represents interfering references if there may exist references $r_1, r_2 \in \gamma(e)$ such that the objects that r_1, r_2 refer to are connected/aliased. A three-element lattice, $np < ip < ap$, *np* for edges with all non-interfering references and *ip* for potentially interfering references and *ap* for potentially aliasing references, is used to represent the interference property.

The Heap Graph. Each node in the graph either represents a region of the heap or a variable. The variable nodes are labeled with the variable that they represent. Nodes representing the concrete heap regions contain a record that tracks the types of the concrete objects that the node represents (*types*), the number of objects (either 1 or #) that may be in the region (*count*), and the abstract layout of a node (*layout*). Each node also tracks the connectivity relation between pairs of incoming edges. A binary relation *connR* is used to track the *inConnected* relation. Although the connectivity relation is a property of the nodes, for readability in the figures we associate the information with the edges. Thus, each node is represented as a record of the form `[types layout count]`.

As in the case of the nodes, each edge contains a record that tracks additional information about the edge. The *offset* component indicates the offsets (labels) of the references that are abstracted by the edge. The number of references that the edge may represent is tracked with the *maxCut* property. The *interfere* property tracks the possibility that the edge represents references that interfere. Finally, we have a field *connto* which is a list of all the other edges/variables that the edge may be connected to according to the *connR* relation (we add a (`!`) for the edges in the list that represent references which *may* alias and a ($\sim$) if the edges represent single references that *must* alias). To simplify the figures if the *connto* field is empty we omit it entirely from the record in the figure. Since the variable edges always represent single references and the offset label is implicitly the name of the variable the record simply contains the *connR* information or is omitted entirely if the *connR* relation is empty. To simplify the discussion of the examples each edge also has a unique label. The pointer edges in the figures are represented as records {`label offset maxCut interfere connto`}.

The abstract heap domain is restricted via a normal form [14,15]. The normal form ensures that the heap graph remains finite, and that equality comparisons are efficient. The local data flow analysis is performed using a *Hoare (Partially Disjunctive) Power Domain* [13,26] over these graphs. Interprocedural analysis is performed in a context-sensitive manner and the procedure analysis results are memoized. At each call/return site the portion of the heap graphs passed to the call are joined into a single graph. The design of the join operation is such that, in general, information lost in the join can be recovered when needed later in the program. The decision to perform joins at call sites (programs tend to have uniform expectations of the portion of the heap passed to and returned from calls) and to perform the join only on the portion of the heap passed to the called method results in very little loss of precision while ensuring the abstract model remains compact.

Abstract Call Stack. Our concrete model for the *call stack* is a function $S_m : (LV \times \mathbb{N}) \mapsto O$, where LV is the set of local variable names and $\mathbb{N}$ represents the depth in the call sequence (main is at depth 1) and O is the set of all live objects. Thus, the pair $(\mathtt{v}, 4)$ refers to the value of the variable $\mathtt{v}$ in the scope of the 4^{th} call frame.

To represent the concrete call stack we introduce *stack variables* which represent the values of local variables on the stack (for a variation on this approach see [22]). In our extension each *stack variable* summarizes all the possible targets (in a given graph) for a given variable name on the stack. Given a variable name $\mathtt{v}$ and a heap graph G we define a variable name $\mathtt{v'}$ for use in the abstract domain (we will select a better naming scheme in Section 4) where: $\mathtt{v'}$ is the abstraction of all the variables in the call stack, $\exists i \in \mathbb{N},$ node $n \in G,$ object o_n s.t. $o_n \in \gamma(n) \wedge S_m(\mathtt{v}, i) = o_n$.

By associating the set of stack locations that are abstracted with the set of targets in a given abstract heap graph, we can naturally partition the *stack variables* along with the heap graphs. Since each *stack variable* is associated with only the values on the stack that point into a region of the heap represented by the given heap graph, it is straightforward to partition and join them when partitioning the heap graphs.

Thus, during the local analysis the heap graph represents the portion of the program heap that is visible from the local variables and is augmented with some number of *stack variables* and *cutpoint variables* which relate variable values and the heap in the caller scope to the portions of the heap reachable from callee scope local variables.

For efficiency and in order to ensure analysis termination the naming scheme we choose will result in situations where multiple cutpoint (or stack) edges are given the same name. This may result in some amount of information loss (particularly with respect to reachability and aliasing). To minimize the loss that occurs we introduce an instrumentation domain for the stack/cutpoint variable edges, *nameColl* = {*pdj*, *pua*, *pa*}. Where *pdj* indicates a cutpoint/stack name representing (a single edge) or edges where the edges do not represent any pairwise *connected* references, *pua* indicates a name representing multiple edges where there are no pairwise *aliases*, while *pa* is the indicates the name represents edges that they may have pairwise *aliasing*. Thus, the cutpoint variable edges are represented with records {`maxCut interfere connto nameColl`} (stack variables are not used in this example).

4 Stack Variables, Cutpoint Labels

When performing the project operation in heaps with cutpoints we need to name the *stack variables* as well as the *cutpoint* edges. We use a simple technique for the stack variables: given a variable name `v` defined in the caller function `fcaller` we use the name `$fcaller*v` to represent this variable in the callee scope. This naming scheme can create false dependencies on the local scope names unless the variable information is normalized during the comparisons of entries in the memo-table.

Naming edges that cross the cutpoints is more complex since we need to balance the accuracy of the analysis with the potential of introducing spurious differences resulting from isomorphic (or nearly so) cutpoint edges being given different names. For the renaming of the cutpoint edges we assume that special names for the arguments to the function have been introduced. The first pointer parameter is referred to by the special variable name `p1` and the i^{th} pointer argument is referred to by the variable `pi`.

Figure 2(c) shows a recursive call to `LInit` where the special argument name `p1` has been added to represent the value of the first argument to the function. In this figure the edge *e1* is a cutpoint edge since it starts in the portion of the heap that is unreachable from the argument variables and ends in a portion of the heap that is reachable from the argument variables (this differs slightly from the definition for cutpoints in [21] but allows us to handle edges uniformly).

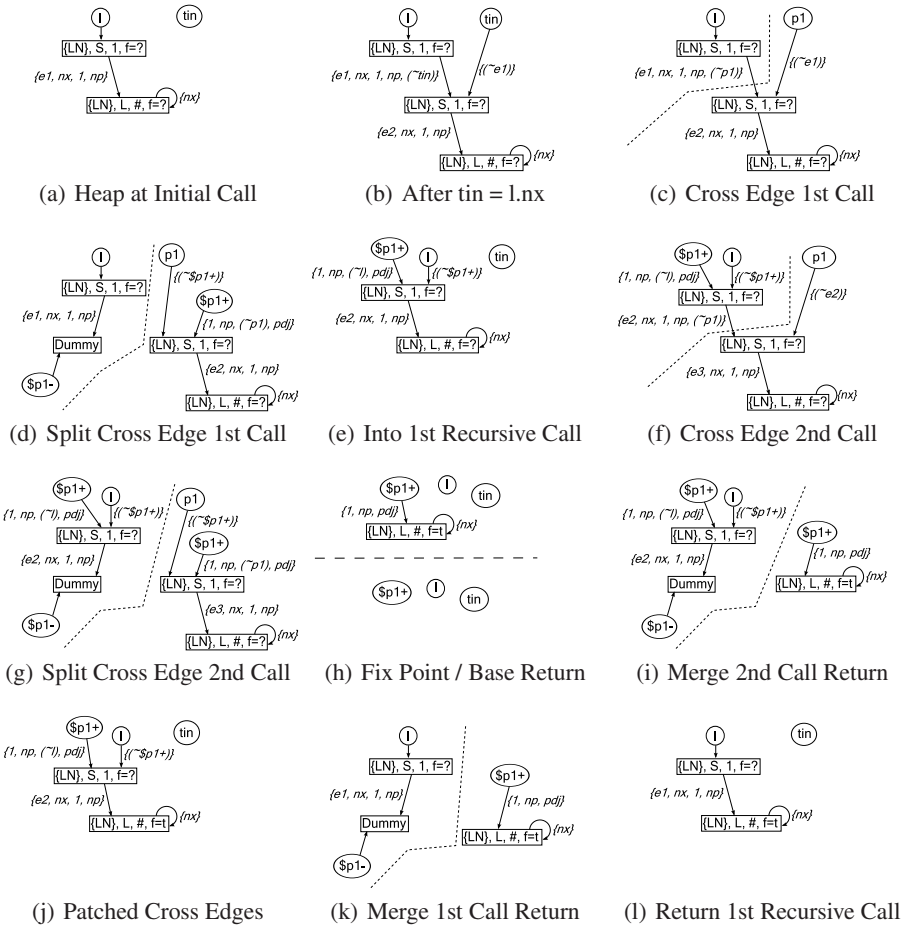
For each cutpoint edge we generate a pair of names: one is used in the unreachable section of the heap graph and one in the reachable section, which allows an abstract heap model to represent both incoming and outgoing cutpoint edges that are isomorphic and exist in the same abstract heap component without loss of precision.

If we are adding a cutpoint for the method call `fcaller` and the edge *e*, which is a cutpoint, starting at *n* and ending at n', and has edge label `fe`. We can find the shortest path (`f1 ... fk`) from any of the `pi` variables to n' (using lexographic comparison on the path names to break ties). Using the `pi` argument variable and the path (`f1 ... fk`) we derive the cutpoint `basename` = `fcaller*pi*f1*...*fk*fe` We compute a pair of static names (*unreachN*, *reachN*) where *unreachN* = `$basename-` and *reachN* = `$basename+`. In Figure 2(d) the cutpoint name `$p1+` (for brevity we simply label the cutpoint with the `pi` variable) is used to represent the endpoint of the cutpoint edge in the reachable component of the heap and `$p1-` to track a dummy node associated with the cutpoint edge in the unreachable component of the heap.

5 Example

The example program, Figure 1, recursively initializes the `f` fields in a linked list to the value `true`. Figure 2(a) shows the abstract heap model at the entry of the first call to the procedure (for simplicity we ignore any caller scope variables).

In Figure 2(a), variable `l` refers to a node that represents `LNode` objects (*types* = {`LNode`}, abbreviated to `LN`), that represents a region with no internal connections (*Layout* = *S*), which contains a single object (*count* = 1), and where all the incoming edges represent disjoint pointers (the `connto` lists on the edges are omitted). In this figure we also have that the elements in the list have unknown truth values in the `f`



(a) Heap at Initial Call (b) After tin = l.nx (c) Cross Edge 1st Call

(d) Split Cross Edge 1st Call (e) Into 1st Recursive Call (f) Cross Edge 2nd Call

(g) Split Cross Edge 2nd Call (h) Fix Point / Base Return (i) Merge 2nd Call Return

(j) Patched Cross Edges (k) Merge 1st Call Return (l) Return 1st Recursive Call

Fig. 2. Recursive Calls

fields (`f=?`). There is a single edge out of this node representing pointers stored in the `nx` field of the object represented by the node. This edge represents a single pointer (*maxCut* = 1) and all the pointers are non-interfering (*interfere* = *np*). Finally, this edge refers to a node that also represents `LNode` objects but may represent many of these objects (*count* = #) and, since the *Layout* value is *List*, we know that the objects may be connected in a list-like shape. Since there is a single incoming edge and it represents a single pointer, we can safely assume that this edge refers to the head of the list structure.

Figure 2(b) shows the abstract heap model just after executing the statement `tin = l.nx`. Since we know that *e1* refers to the head element of the list from Figure 2(a) we replaced the single *List*-shaped node with a node representing the unique head element and a node representing the tail of the list. Since the head element is unique we set the *count* of this new node to 1. Additionally, the only possible layout for a node of *count* 1 is *Singleton*. Finally, if a node represents a single object then all the outgoing field edges

can each represent a single pointer. Thus, we set the outgoing edge to have a *maxCut* = 1. Also note that after the load the analysis has determined that `tin` and `e1` must alias (indicated by the ∼`e1` and ∼`tin` entries in the connectivity lists).

Figure 2(c) shows the state of the abstract heap at the entry of the *project* procedure. The special name *p1* has been added to represent the value of the first pointer argument to the function and we have added a dotted line to indicate the reachable and unreachable portions of the heap. Note that the edge *e1* is a cutpoint edge according to our definition.

The result of the project operation is shown in Figure 2(d). The *e1* edge, which was a cutpoint edge for the call, has been remapped to a dummy node and the static cutpoint names `$p1-` and `$p1+` (for brevity we omit the procedure name and edge labels from the static names) have been introduced at the dummy node and at the target of this edge in the reachable section. Since this cutpoint edge only represents the single cutpoint edge generated in this call frame `nameColl = pdj`. Also note that the analysis has determined that the formal parameter `p1` must alias the cutpoint edge `$p1+`.

Figure 2(e) shows the resulting abstract heap that is passed into the callee scope for analysis. Since all the local variables in the caller scope either did not refer to nodes in the callee reachable section or are dead after the call return we do not have to give them stack names and can remove them entirely from the heap model. Figure 2(f) shows the abstract heap at the entry to the project function for the second recursive call. Again we have a cutpoint edge *e2*. Note that the reachable cutpoint label, `$p1+` introduced in the previous call is now in the unreachable portion of the heap, thus (`$p1+`) does not conflict with the unreachable name added in this call (`$p1-`). The result of the project operation is shown in Figure 2(g).

Figure 2(h) shows the eventual fixpoint approximation (above the dotted line) of the analysis of this function and also the base case return value (below the dotted line). Notice in the base case return value we were able to determine that the test `l == null` implies that `l` must be null and since we preserved must alias information through the cutpoint introduction we can infer that `l` must alias `$p1+`, which implies the cutpoint edge (`$p1+`) must also be null. Thus, the analysis can infer that on return the cutpoint edge is either `null` or is non-null and refers to some list in which all the `f` fields have been set to `true` (`f=t` in the figure).

In Figure 2(i) we show how the fixpoint approximation for the reachable section of the heap is recombined with the unreachable section of the heap using the *extend* operation. After the recombination we get the abstract heap model shown in Figure 2(j). In Figure 2(i) we have unioned the graphs and are ready to patch up the cutpoint cross edge information. The static name `$p1+` in the reachable portion of the heap has been used to compute the associated unreachable name (`$p1-`). Then the algorithm identifies the edge associated with the dummy node referred to by `$p1-` (*e2*) and remapped this edge to end at the target of `$p1+` (`tin` has been nullified since it is dead).

Figure 2(k) shows the *extend* operation at the return from the first recursive call which is similar to the situation in the second recursive call. The resulting abstract heap is shown in Figure 2(l) which can be joined with the result of the base case test and then completes the analysis of the method. As desired, the analysis has determined that the recursive list initialize procedure preserves the list shape of the argument list and that all of the `f` fields in the list have been set to `true` (`f=t` in the figures).

6 Project and Extend Algorithms

Project. We assume that before the *projectHeap* function is invoked all of the special argument variable names have been added to the heap model. This allows *projectHeap* (Algorithm 1 below) to easily compute the section of the heap model that is reachable in the callee procedure and then compute the set of nodes that comprise the unreachable portion of the heap model.

Algorithm 1. projectHeap

input : h: the heap model to be partitioned
output: h_r, h_u: the reachable and unreachable partitions, *snu*, *ncs*: the static names used and newly created
reachNodes ← set of nodes reachable from args;
unreachNodes ← set of nodes unreachable from args;
crossEdges ← set of edges that start in *unreachNodes* and end in *reachNodes*;
snu ← ∅;
ncs ← ∅;
foreach edge *e* in *crossEdges* **do**
    (sn, isnew) ← procCrossEdge(*h*, *e*, *reachNodes*);
    snu.add(*sn*);
    if *isnew* **then** *ncs*.add(*sn*);

h_u ← subgraph of h on the nodes *unreachNodes* ∪ {dummy nodes from procCrossEdge};
h_r ← subgraph of h on the nodes *reachNodes*;
return *(*h_r*,* h_u*,* snu, ncs*)*;

For each edge that crosses from the unreachable section into the reachable section we add a pair of static names to represent the edge (Algorithm 2). Since the heap model stores a number of domain properties in each edge, we create a dummy node and remap the edge to end at this node. Then, the *unreachN* static name is set to refer to this dummy node. In the reachable portion of the heap graph we simply set the *reachN* static name to refer to the target of the cross edge.

When adding the *reachN* static name to the reachable section of the heap graph the name may or may not already be present in the heap graph. If the name is not present then we add it to the static name map and for later use we note that this is the call where the name is introduced. Otherwise a name collision has occurred and we must mark the edges representing the possible cutpoints appropriately (for simplicity we mark all the edges). If there may be aliasing we note that the cutpoints from different frames may have aliasing targets (*pa*) and similarly if the new cutpoint edge may be connected with an existing cutpoint edge we mark them as being pairwise connected (*pua*). The functions *makeEdgeForUnreachCutpoint* and *makeEdgeForReachCutpoint* are used to produce edges to represent the cutpoint (based on the static name and the cutpoint edge properties) in the unreachable and reachable portions of the heap.

Once all of the cutpoint edges have been replaced by the required static names, the heap can be transformed into the unreachable version (where all the nodes in the reachable section and all the variables/static names that only refer to reachable nodes have

been removed) and the reachable version (where the nodes in the unreachable section and the associated names have been removed).

Algorithm 2. procCrossEdge

input : h: the heap, e: the cross edge, *reachNodes*: set of reachable nodes
output: *rsn*: the name used, *isnew*: true if *rsn* a new name
$n_e \leftarrow$ the node e ends at;
$n_i \leftarrow$ new dummy node;
(ursn, rsn) $\leftarrow$ genStaticNamePairForEdge(h, e);
$e_u \leftarrow$ makeEdgeForUnreachCutpoint(e, *ursn*);
set endpoint of e_u to n_i;
add e_u as an edge for *ursn*;
$e_r \leftarrow$ makeEdgeForReachCutpoint(e, *rsn*);
set endpoint of e_r to n_e;
remap the endpoint of e to n_i;
if the name *rsn* exists and has edges pointing to a node in *reachNodes* **then**
    rsnes $\leftarrow \{e' | e'$ is an edge for the cutpoint var *rsn*$\}$;
    add e_r as an edge for *rsn*;
    if e_r *is inConnected with an edge in* rsnes **then** set edges in *rsnes* and e_r to *pua*;
    if e_r *may alias with an edge in* rsnes **then** set edges in *rsnes* and e_r to *pa*;
    return *(*rsn*,* false*)*;
else
    add the name *rsn* to h;
    add e_r as an edge for *rsn*;
    return *(*rsn*,* true*)*;

Extend. After the call return we need to rejoin the unreachable portion of the heap that we extracted before the procedure call entry with the result we obtained from analyzing the callee procedure. This is done by looking at each of the static names that was used to represent a cutpoint edge and reconnecting as required. Then, each of the newly introduced cutpoint names can be removed from the heap model. The pseudo-code to do this is shown in Algorithm 3.

This algorithm merges all edges with the same reachable cutpoint name so that there is at most one target edge for a given cutpoint name in the reachable heap h_r (this simplifies the algorithm and is in our experience is quite accurate). The algorithm then pairs up the two cutpoint names and remaps the edge we saved in the unreachable section to the target node in the reachable section subject to a number of tests to propagate sharing information (the nullity information is propagated due to the fact that the dummy node and all incoming edges are always removed but the foreach loop on the targets of *ursn* does not execute since the target set is empty). The e_r.*nameColl* = *pua* test is true if this edge represents sets of pointers that do not have pairwise aliases. Thus, we mark the newly remapped edge and e_r as pairwise unaliased. Similarly, the e_r.*nameColl* = *pdj* test is true if this edge represents cutpoint/stack edges that are pairwise disjoint. Thus, we mark the newly remapped edge and e_r as pairwise disjoint.

Algorithm 3. extendHeap

input : h_r, h_u: the reachable and unreachable partitions, *snu*, *ncs*: the static names used and newly created
output: h: the joined heap model
$h \leftarrow$ new *heap*();
h.heapGraph $\leftarrow$ mergeGraphs(h_r.heapGraph, h_u.heapGraph);
foreach static name *sn* in *snu* **do**
  ursn $\leftarrow$ reachNameToUnreachName(*sn*);
  $n_r \leftarrow$ the target of *sn* in h_r.nameMap;
  foreach node n_u that is a target of *ursn in* h_u.*nameMap* **do**
    $e_r \leftarrow$ the single incoming edge to n_u;
    remap e_r to end at the target of n_r;
    e_r.interfere = e_r.interfere $\sqcup$ n_r.interfere;
    if e_r.nameColl = *pua* **then** set e_r and n_r as unaliased;
    if e_r.nameColl = *pdj* **then** set e_r and n_r as disjoint;

  h_u.removeNodeAllEdges(target of *ursn*);
  h_u.unmapStaticName(*ursn*);
  if sn *in* ncs **then** h_r.unmapStaticName(*sn*);

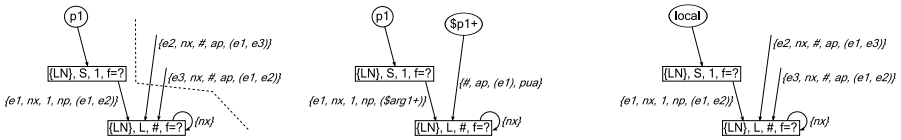
h.nameMap $\leftarrow$ mergeNameMaps(h_r.nameMap, h_u.nameMap);
return h

The major components of this algorithm are the separation of the *mergeGraphs* action from the *mergeNameMaps* action and the elimination of the static cutpoint edge names that were introduced for this call.

The *mergeGraphs* function computes the union of the graph structures that represent the abstract heap objects, while the *mergeNameMaps* function computes the union of the name maps (which are maps from the stack/variable/cutpoint names to the nodes in the graph structure that represent them). This separation allows the algorithm to nullify the names created for this call which prevents the propagation of unneeded cutpoint edge targets to the caller scope. The function *unmapStaticName* is used to eliminate a given static name from the abstract heap model name map.

Example Name Collision. The introduction of the *nameColl* domain minimizes the precision loss that occurs when a cutpoint or stack variable name collision occurs. Figure 3 shows an example of such a situation. In this figure we show part of a heap where the edges *e2* and *e3* are both cutpoint edges and they do not represent any pairwise aliasing pointers (no ! in the *connTo* lists) although they each represent sets of pointers that may alias, *interfere* = *ap*.

In this example our naming scheme will result in *e2* and *e3* being represented with the same cutpoint name. However, our method will mark this cutpoint edge as *nameColl* = *pua* (Figure 3(b)). This means that on return the *extend* algorithm will set the edges that are mapped to this cutpoint as being pairwise unaliased (Figure 3(c)) as desired. Thus, even though there was a name collision for the cutpoints we avoided (in this case completely) the loss of sharing information about the heap.



(a) Colliding Names (b) To Same Cutpoint (c) PUA on Return

Fig. 3. Name Collision

7 Experimental Results

The proposed approach has been implemented and the effectiveness and efficiency of the analysis have been evaluated on the source code for programs from a variation of the Jolden [3,18] suite and several programs from SPEC JVM98 [27] (raytrace, modified to be single threaded, db and compress). The analysis algorithm is written in C++ and was compiled using MSVC 8.0. The parallelization benchmarks were run using the Sun 1.6 JVM. All runs are from our 2.8 GHz PentiumD machine with 1 GB of RAM.

We ran the analysis with the project/extend operations enabled (the *Project* column) and disabled (the *No-Project* column) and recorded the analysis time, the average number of times a method needed to be analyzed, and used the resulting shape information to parallelize the programs, shown in Figure 4. The results indicate that the project/extend operations have a significant impact on the performance of the analysis, reducing the number of contexts that each function needs to be analyzed in (on average reducing the number of contexts by a factor of 4.3) which results in a substantial decrease in analysis times (by a factor of 18.4). As expected this reduction becomes more pronounced as the size and complexity of the benchmarks increases, in the case of raytrace the analysis time without the project/extend operation is impractically large (772.6 seconds) but when we use the project/extend operations the analysis time is reduced to 35.11 seconds.

We used the shape information from the analysis to drive the parallelization of the benchmarks by using multiple threads in loops and calls, resulting in the speedup columns in Figure 4. Given the shape information produced by the analysis it is straight forward to compute what parts of the heap are read and written by a loop body or method call and thus which loops and calls can be executed in parallel (in raytrace we treated the memoization of intersect computations as spurious dependencies). Once the analysis identified locations that could be parallelized we inserted calls to a simple thread pool (since our current work is focused on the analysis this is done by hand but can be fully automated [6,23,10]). In 8 of 9 benchmarks that are suitable for shape driven parallelization (compress, db and mst do not have any data structure operations that are amenable to shape driven parallelization) we achieve a promising speedup, averaging a factor of 1.69 over the benchmarks.

Our experimental results show that the information provided by the analysis can be effectively used (in conjunction with existing techniques) to drive the parallelization of programs. To the best of our knowledge this analysis is the only shape analysis that is able to provide the information required to perform shape driven parallelization for five of these benchmarks (em3d, health, voronoi, bh and raytrace). Given the speed with

Benchmark Info			No-Project			Project		
Benchmark	Stmt	Method	Time	Avg Cont.	Speedup	Time	Avg Cont.	Speedup
bisort	260	13	0.86s	10.6	1.00	0.28s	1.9	1.72
em3d	333	13	0.12s	2.5	1.75	0.08s	1.8	1.75
mst	457	22	0.06s	3.2	NA	0.04s	3.0	NA
tsp	510	13	1.51s	22.4	1.84	0.17s	7.0	1.84
perimeter	621	36	54.57s	105.9	1.00	2.97s	50.2	1.00
health	643	16	3.24s	12.9	1.00	2.26s	4.2	1.76
voronoi	981	63	20.89s	61.4	1.00	2.67s	37.2	1.68
power	1352	29	5.71s	26.8	1.93	0.17s	1.3	1.93
bh	1616	51	8.64s	32.8	1.75	2.68s	7.3	1.75
compress	1102	41	0.29s	2.9	NA	0.18s	2.2	NA
db	1214	30	0.94s	3.7	NA	0.68s	2.8	NA
raytrace	3705	173	772.60s	293.1	1.00	35.11s	15.6	1.76
Overall	12794	523	869.43s	48.2	1.36	47.29s	11.2	1.69

Fig. 4. The Stmt and Method columns list the number of statements and methods for each benchmark. The columns for the No-Project and Project variations of the analysis list: the analysis time in seconds, the average number of times each method was analyzed and parallel speedup achieved on a 2 core 2.8 GHz PentiumD processor.

which the analysis is able to produce the information needed for the parallelization and the consistent parallel speedup that is obtained in the benchmarks (1.69 over all of the benchmarks and 1.77 if we exclude the benchmark mst), we find the results encouraging.

Of particular interest is the raytrace benchmark. This program is 2-4 times larger than any benchmarks used in the related work, builds and traverses several heap structures that have significant sharing between components. It also makes heavy use of virtual methods and recursion. This benchmark presents significant challenges in terms of the complexity and size of the program as well as in terms of the range of heap structures that need to be represented in order to accurately and efficiently analyze the program. Our analysis is able to manage all of these aspects and is able to produce a precise model of the heap (allowing us to obtain a speedup of 1.76 using heap based parallelization techniques). Further, the analysis is able to produce this result while maintaining a tractable analysis runtime.

8 Conclusion

We presented and benchmarked project/extend operations for a store-based heap model that is capable of precisely representing a range of shape, connectivity and sharing properties. The project and extend operations we introduced are designed to minimize the analysis time by reducing the number of unique calling contexts for each function and to minimize the imprecision introduced by the collisions that occur between stack/cutpoint names.

Our experimental results using the project/extend operations are very positive. The analysis was able to efficiently analyze benchmarks that build and manipulate a variety

of data structures. Our benchmark set includes a number of kernels that were originally designed as challenge problems for automatic parallelization (the Jolden suite) and several benchmarks from the SPEC JVM98 suite (including a single threaded version of raytrace). Our experimental results demonstrate that the project/extend operations are effective in minimizing the number of contexts that need to be analyzed (on average a factor of 4.3 reduction), improving analysis accuracy (seen as improved parallelization results, in 4 out of 12 benchmarks) and substantially reducing the analysis runtime (by a factor of nearly 20). Our heap analysis was also able to provide sufficient information to successfully parallelize the majority of benchmarks we examined, including several that cannot be successfully analyzed/parallelized using other proposed shape analysis methods.

Acknowledgments

This work is supported under subcontract R7A824-79200004 from the Los Alamos Computer Science Institute and Rice University and by the National Science Foundation (grant 0540600). Manuel Hermenegildo is also supported by the Prince of Asturias Chair at UNM, and projects MEC-MERIT, CAM-PROMESAS, and EU-MOBIUS.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Bruynooghe, M.: A Practical Framework for the Abstract Interpretation of Logic Programs. J. Log. Program 10, 91–124 (1991)
3. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in Java. In: PACT (2001)
4. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 463–482. Springer, Heidelberg (2003)
5. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL (1996)
6. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting parallelism in C programs with recursive data structures. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 159–173. Springer, Heidelberg (1998)
7. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
8. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 379–392. Springer, Heidelberg (2007)
9. Guo, B., Vachharajani, N., August, D.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
10. Hendren, L.J., Nicolau, A.: Parallelizing programs with recursive data structures. IEEE TPDS 1(1) (1990)
11. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)

12. Jeannet, B., Loginov, A., Reps, T.W., Sagiv, S.: A relational approach to interprocedural shape analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 246–264. Springer, Heidelberg (2004)
13. Manevich, R., Sagiv, S., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
14. Marron, M., Kapur, D., Stefanovic, D., Hermenegildo, M.: A static heap analysis for shape and connectivity. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 345–363. Springer, Heidelberg (2007)
15. Marron, M., Majumdar, R., Stefanovic, D., Kapur, D.: Dominance: Modeling heap structures with sharing. Tech. report, CS Dept., Univ. of New Mexico (August 2007)
16. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL (2004)
17. Muthukumar, K., Hermenegildo, M.V.: Compile-time derivation of variable dependency using abstract interpretation. J. Log. Program (1992)
18. Modified Jolden Benchmarks (August 2007), `http://www.cs.unm.edu/~marron`
19. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
20. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
21. Rinetzky, N., Bauer, J., Reps, T.W., Sagiv, S., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL (2005)
22. Rinetzky, N., Sagiv, S.: Interprocedural shape analysis for recursive programs. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 133–149. Springer, Heidelberg (2001)
23. Rugina, R., Rinard, M.C.: Automatic parallelization of divide and conquer algorithms. In: PPOPP (1999)
24. Sagiv, S., Reps, T.W., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: POPL (1996)
25. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL (1999)
26. Smyth, M.B.: Power domains and predicate transformers: A topological view. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 662–675. Springer, Heidelberg (1983)
27. Standard Performance Evaluation Corporation. JVM98 Version 1.04 (August 1998), `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`
28. Wilhelm, R., Sagiv, S., Reps, T.W.: Shape analysis. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 1–17. Springer, Heidelberg (2000)

Identification of Logically Related Heap Regions

Mark Marron[1] Deepak Kapur[2] Manuel Hermenegildo[1]

[1]IMDEA-Software (Madrid, Spain)
[2]University of New Mexico (New Mexico, USA)
{mark.marron, manuel.hermenegildo}@imdea.org, kapur@cs.unm.edu

Abstract

This paper introduces a novel set of heuristics for identifying logically related sections of the heap such as recursive data structures, objects that are part of the same multi-component structure, and related groups of objects stored in the same collection/array. When combined with lifetime properties of these structures, this information can be used to drive a range of program optimizations including pool allocation, object co-location, static deallocation, and region-based garbage collection. The technique outlined in this paper also improves the efficiency of the static analysis by providing a compact normal form for the abstract models (speeding the convergence of the static analysis).

We focus on two techniques for grouping parts of the heap. The first is a technique for identifying recursive data structures in object-oriented programs based on connectivity and type information. The second technique is a method for grouping objects that make up the same composite structure and that allows us to partition the objects stored in a collection/array into groups based on a *similarity* relation. We provide a parametric component in the *similarity* relation to support specialized analysis applications (e.g. numeric analysis of object fields). Using the Em3d and Barnes-Hut benchmarks from the JOlden suite we show how these grouping methods can be used to identify various types of logical structures and enable the application of many region-based optimizations.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages (program analysis)

General Terms Languages, Performance, Verification

1. Introduction

Identifying and grouping logically related parts of the program heap in an abstract program model is useful both to client optimization applications (which can use the information to perform pool allocation, object co-location, static deallocation, etc.) and in improving the performance of the static data flow analysis (providing a normal form which speeds the convergence of the analysis). This paper presents a novel set of grouping heuristics for identifying and grouping these regions in a manner that supports a wide range of client applications and that can be used in practice to produce an efficient static analysis.

ISMM'09 June 19–20, 2009, Dublin, Ireland.

Research on object allocation and memory layout has used the notions of logically related structures to improve the spatial locality of objects with similar temporal accesses via techniques such as pool allocation [16, 4] and object co-location [12, 8]. Other applications which use logically related sections of the heap have focused on improving the efficiency of garbage collection. The most direct application is static deallocation of regions or data structures [16, 5, 13]. There has also been work [15] on using region information to reduce the pause times of garbage collection by only performing the collection on portions of heap that are likely to contain many dead objects. Similar approaches (when combined with heap based read/write information) can also be used to support parallel garbage collection by statically identifying which parts of the heap can be safely collected without concern for the mutator.

The techniques listed above use a variety of approaches for identifying region information that is later used in the optimization phase. The techniques range from simple grouping via the points-to partitions computed using a Steensgaard style analysis [26, 14] to more sophisticated approaches as done in [16, 17, 13]. However, as these techniques are based on points-to style analyses or use limited amounts of context/flow sensitivity they cannot precisely model many properties (sharing and shape) of data structures that are used extensively in object oriented programs. The technique in this paper offers a significantly higher degree of precision for identifying regions than these approaches and can be used directly to improve the effectiveness of many region based memory optimizations.

In addition to being useful for a range of optimization techniques the region identification technique we present in this paper can be used to improve the performance of various static analysis techniques. This is achieved by using the region identification to define a normal form for the abstract models, reducing the height of the abstract lattice. This use of a normal form can be seen as a pseudo-widening operation used to transform a domain of infinite height (e.g, linked lists of size $0, 1 \ldots \infty$) into a finite height lattice (e.g, linked lists that are of size $0, 1, 2,$ or some unknown length ω). There are two parts to this normalization that we address in this paper. The first is the compression of recursive structures of potentially unbounded size, such as lists or trees, into finite representations. The second is the grouping of objects that make up composite data structures or partitioning objects stored in a collection/array based on a *similarity* relation.

While the concept of computing normal forms for heap representations is not novel to this paper —symbolic access paths in [7], normalization/merge in [19, 18, 3, 6, 21], and the append left/right rules in [1, 28] or similar rules for inductive synthesis in [11]— the heuristics we use to accomplish this are significantly more general than what has been used in previous work. In particular the formalization applies to any type of recursive data structure (as opposed to just lists or trees [1, 28, 19, 11]) and recursive data structures that are part of larger composite structures (such as in [1, 21]). The heuristics in this work can precisely model multi-

component structures with shared components which cannot be handled by [1, 28, 19]). Finally these heuristics support more effective grouping of the contents of collections (arrays or collections from `java.util`) than is possible with the methods described in [3, 23, 10] (and which are left out in most other approaches).

We begin with a brief introduction of the parametric labeled storage shape graph (*lssg*) model, Section 2, that we use to illustrate the main contributions of this paper. These contributions as described in Sections 3 and 4 are:

- A method for identifying and grouping recursive data structures.
- A method for grouping objects that form multi-object composite structures.
- A parametric approach to grouping the contents of arrays/collections.

Finally, in Section 6 we use the Em3d and Barnes-Hut benchmarks from the well known JOlden suite to illustrate the results of the region analysis and how this information can be used to support some of the optimizations mentioned above.

2. Concrete Heap and Labeled Shape Graph

We begin by reviewing the abstract graph model that we build on in this work (although the concepts presented in this paper can be applied in other approaches such as those that rely on separation logic [1, 11, 28]). In previous work [21, 22, 23] this model is used to precisely perform shape and sharing analysis on a range of Java programs. While the properties discussed therein are critical to precisely analyzing these programs (and similarly the region identification method presented in this paper is critical to the results in these papers), we do not need all of this information in order to perform region identification and grouping. Thus, to simplify the discussion and to focus on the novel concepts in this paper we present a simplified version of the model.

2.1 Concrete Semantics

To analyze a program we first transform (via a modified compiler frontend) the Java 1.4 source into a semantically equivalent program in a simpler to analyze core language. This intermediate language is statically typed, has method invocations, conditional constructs, exception handling and the standard looping statements. The state modification and expressions cover the standard range of program operations: load, store, and assignment along with logical, arithmetic, and comparison operators. During this transformation step we also load in our specialized standard library implementations, so we can analyze programs that use classes from `java.util`, `java.lang`, and `java.io`.

The semantics of memory are defined in the usual way, using an environment, mapping variables into values, and a store, mapping addresses into values. We refer to the environment and the store together as the concrete heap, which is treated as a labeled, directed multi-graph (V,O,R) where each $v \in V$ is a variable, each $o \in O$ is an object on the heap and each $r \in R$ is a reference (either a variable reference or a pointer between objects). The set of references $R \subseteq (V \cup O) \times O \times L$ where L is the set of storage location identifiers (a variable name in the environment, a field identifier for references stored in objects, or an integer offset for references stored in arrays/collections).

A *region* of memory $\Re = (C,P,R_{in},R_{out})$ consists of a subset $C \subseteq O$ of the objects in the heap, all the pointers $P = \{(a,b,p) \in R \mid a,b \in C \wedge p \in L\}$ that connect these objects, the references that enter the region $R_{in} = \{(a,b,r) \in R \mid a \in (V \cup O) \setminus C \wedge b \in C \wedge r \in L\}$ and references exiting the region $R_{out} = \{(a,b,r) \in R \mid a \in C \wedge b \in O \setminus C \wedge r \in L\}$.

2.2 Storage Shape Graph Abstraction

Our abstract heap domain is based on the *storage shape graph* [3] approach. An *abstract storage graph* is a tuple of the form $(\hat{V},\hat{N},\hat{E})$, where $\hat{V}$ is a set of abstract nodes representing the variables, $\hat{N}$ is a set of abstract nodes (each of which abstracts a region $\Re$ of the heap), and $\hat{E} \subseteq (\hat{V} \cup \hat{N}) \times \hat{N} \times \hat{L}$ are the graph edges, each of which abstracts a set of pointers, and $\hat{L}$ is a set of abstract storage *offsets* (variable names, field offsets or the special offset *?* for references stored in arrays/collections). We extend this definition with a set of additional relations $\hat{U}$ that further restrict the set of concrete heaps that each shape graph abstracts. The *labeled storage shape graphs (lssg)*, which we refer to simply as *abstract graphs*, are tuples of the form $(\hat{V},\hat{N},\hat{E},\hat{U})$.

DEFINITION 1 (Valid Concretization of a *lssg*). *A given concrete heap h is a* valid concretization *of a* labeled storage shape graph *g if there are functions Π_v,Π_o,Π_r such that the following hold:*

- *$\Pi_v : V \mapsto \hat{V}$, $\Pi_o : O \mapsto \hat{N}$ and $\Pi_r : R \mapsto \hat{E}$ are functions (and Π_v is 1–1).*
- *h,Π_v,Π_o, and Π_r satisfy all the relations in $\hat{U}$.*
- *h,Π_v,Π_o, and Π_r are* connectively consistent *with g.*

Where h,Π_v,Π_o,Π_r are connectively consistent *with g if:*

- *$\forall\, o_1,o_2 \in O$ s.t. $(o_1,o_2,p) \in R$, $\exists\, e \in \hat{E}$ s.t. $e = \Pi_r((o_1,o_2,p))$, e starts at $\Pi_o(o_1)$, ends at $\Pi_o(o_2)$, and e.offset $= p$.*
- *$\forall\, v \in V$, $o \in O$ s.t. $(v,o,v) \in R$, $\exists\, e \in \hat{E}$ s.t. $e = \Pi_r((v,o,v))$, e starts at $\Pi_v(v)$, ends at $\Pi_o(o)$, and e.offset $= v$.*

To check if a given concrete heap h and maps Π_v,Π_o,Π_r satisfy a given relation in $\hat{U}$ we need to look at the pre-images of the nodes and edges in the abstract graph g under the maps Π_v,Π_o,Π_r. We use the notation $h \downarrow_g e$ to indicate the set of references in the concrete heap h that are in the pre-image of e under the maps. Similarly, we use $h \downarrow_g n$, to indicate the region of the heap that is the pre-image of n under the maps.

2.3 Label Relations (in $\hat{U}$)

Type. For the *type* relation, we add a relation $(n,\{\tau_1,\ldots,\tau_k\})$ (we use the shorthand $n.type = \{\tau_1,\ldots,\tau_k\}$) to $\hat{U}$ for each node in $\hat{N}$, where τ_j are types in the program and say: h,Π_v,Π_o,Π_r satisfies $(n,\{\tau_1,\ldots,\tau_k\})$ iff $\{\texttt{typeof}(o) \mid \text{object } o \in h \downarrow_g n\} \subseteq \{\tau_1,\ldots,\tau_k\}$.

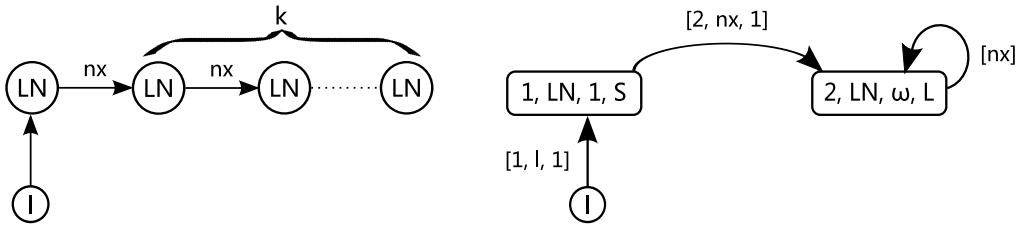
Linearity. The *linearity* relation is used to track the number of objects in the region abstracted by a given node or the number of references abstracted by a given edge. The *linearity* property has 2 values: 1 indicating a cardinality of $[0,1]$ or ω indicating a cardinality of $[0,\infty)$. Given a node n where $h \downarrow_g n = (C,P,R_{in},R_{out})$ then:

$$|C| \in \begin{cases} [0,1] & \text{if } n.linearity = 1 \\ [0,\infty) & \text{if } n.linearity = \omega \end{cases}$$

Similarly for an edge e where $h \downarrow_g e = \{r_1,\ldots,r_j\}$ then:

$$|\{r_1,\ldots,r_j\}| \in \begin{cases} [0,1] & \text{if } e.linearity = 1 \\ [0,\infty) & \text{if } e.linearity = \omega \end{cases}$$

Abstract Layout. To approximate the shape of the structures present in the region that a node abstracts, the analysis uses *abstract layout* properties *{(S)ingleton, (L)ist, (T)ree, (D)ag,* and *(C)ycle}*. The *(S)ingleton* property states that there are no pointers between any of the objects abstracted by the node, given a node n where $h \downarrow_g n = (C,P,R_{in},R_{out})$ then $P = \emptyset$. The other properties correspond to the standard definitions for list, tree, DAG, and cyclic structures in the literature [9, 21, 1].



(a) Concrete List, Length 3 or More (b) Abstraction with Regions Identified

Figure 1: A Linked List and Desired Abstraction with Regions Identified

2.4 Sample Heap and Abstract Graph Model.

Figure 1 shows a linked list of length 3 or more (left) and the representation of this list in the abstract domain with the objects that represent it grouped into regions (right). In the abstract domain each edge is labeled with a unique identifier, an abstract storage *offset*, and a *linearity* label. The nodes are labeled with a unique identifier, a *type* label, a *linearity* label and a *layout* label.

In Figure 1b we see that the variable `l` refers to node 1 which represents a single (*linearity* is 1) ListNode (`LN`) object at the head of the linked list. There is a single edge (edge 2) out of the node representing the single (again *linearity* 1) `nx` (next) pointer, which ends at node 2. This node represents the tail of the list (the self `n` edge and *(L)ist layout*) which may contain many objects (*linearity* is ω).

Partitioning the list into these two nodes captures several important attributes. First we have kept the head of the list (which may be modified though the variable `l`) distinct, giving more opportunities to the analysis for precisely modeling the effects of later program statements. Next, the grouping has produced a compact representation for the list structure which has a substantial impact on the efficiency of the analysis. Finally, we have grouped all of the objects that make up the list into two nodes (the head and the tail, nodes 1 and 2) and as we will see later if there are other unrelated lists in the program (and the analysis can determine that they are unrelated) the abstraction will generate separate nodes for each of these lists. Thus, the information needed by the various optimization techniques we are interested in is preserved (objects in the same structures are grouped together while disjoint structures in the concrete heap are kept separate in the abstract model).

3. Recursive Components

The first contribution in this paper is a generalized method for identifying parts of the abstract heap graph that may represent a single recursive data structure and how these parts should be grouped together (e.g. using multiple nodes to represent the head and tail sections of the linked list). The basic approach of identifying potentially recursive structures is a straightforward examination of the type information and connectivity properties of the program based on recursive field paths [7, 21, 1, 19]. However, there are a number of subtle but important modifications that are needed to maintain the desired level of precision in the results when dealing with non-trivial object-oriented programs.

3.1 Statically Recursive Types

We can identify the types in a program that may be recursive by looking at the type graph for the program. This *static program type graph* has a node for each type that is declared and for each pair of types τ, τ' there is an edge from τ to τ' if τ has a field of type (or supertype) τ'. From this construction we can identify types that are recursive (based on the static type information) as follows:

DEFINITION 2 (Statically Recursive Types). *For a given program and types τ, τ':*

1. *τ, τ' are* statically recursive *iff in the* static program type graph *($\tau \neq \tau' \wedge \tau, \tau'$ are in the same strongly connected component) $\vee$ ($\tau = \tau'$ and there is a self edge).*
2. *τ is a* statically recursive *type iff $\exists \tau'$ s.t. τ, τ' are* statically recursive.

In much of the past work on region identification [21, 7, 19, 1, 11] this static type information has been used (in various ways) to determine if two objects are part of the same recursive data structure. However, this can result in overly approximate region identification in three important classes of heap structures. Below we describe these and how we can modify our concept of recursive structures to characterize them.

3.2 Safe Nodes

In order to accurately simulate the effects of various program statements it is critical to precisely model the targets of variable references. Consider removing an element from a linked list where we have multiple variables pointing into the same list structure. In order to preserve the listness property after the removal we must keep track of the relative positions of the variable references into the list structure and the effects of the assignment statements on the objects referred to by the variables. Thus, even though all of these objects make up the same recursive list structure, we want to use multiple nodes to represent it (one for each location in the list that is being modified in addition to nodes for the tail or other segments).

To identify these important objects which need to be modeled independently we introduce the notion of *safe nodes* (which is similar to the notion of *interrupting nodes* in [19]). We say a node is safe if it represents an interesting point in a recursive data structure (a point where the program is accessing a specific node in the data structure via a variable, as in the above example, or a non-recursive data structure pointing into specific locations in the recursive structure) and we keep these nodes distinct from any other recursive components.

If we have a recursive data structure and we store references to important points in it via another data structure we want to be able to maintain the relations between these specific points in a data structure. This is a generalization of maintaining the precise locations of variable references into a recursive data structure. This is important to analyzing situations of the form: a method returns a `Pair` object containing two references to `ListNode` objects and we want to remove all the elements in the list between the `first` and `second` entries of the `Pair`. If the analysis does not maintain the order relation between the targets of the `first` and `second` reference fields in the list structure we cannot accurately model the effects of the remove operation (e.g., we would conservatively assume that the target of the `second` field could come before the target of the `first` field in the list).

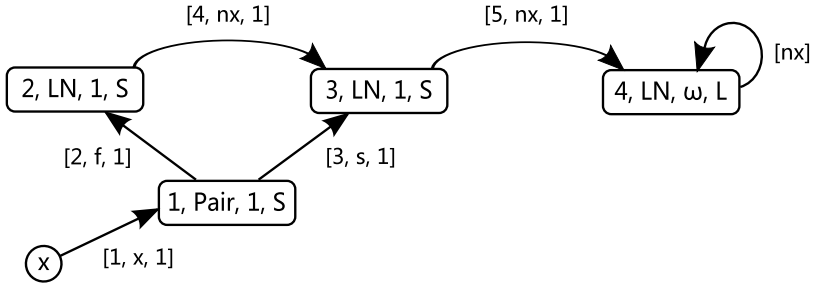
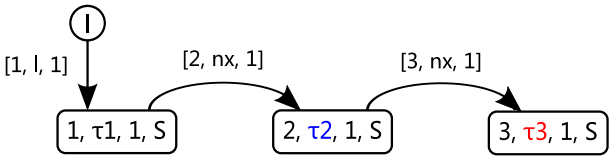
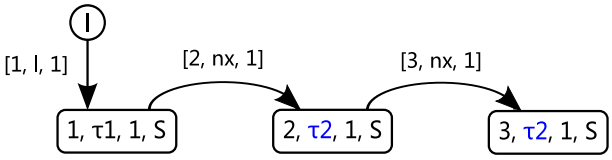


Figure 2: Safe Nodes Example



(a) Recursive Types But No Complete Structure



(b) Recursive Types With Complete Structure

Figure 3: Recursive Types and Complete Structures

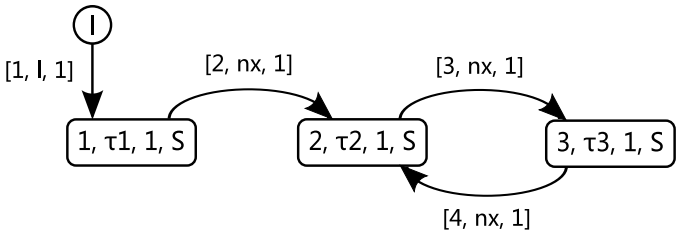


Figure 4: Recursive Cycle

DEFINITION 3 (Safe Node). *A node n is* safe *if it is a node with the* (S)ingleton layout *and either of the following hold:*

1. *$\exists$ variable v that refers to n.*
2. *$\exists$ edge e s.t. e starts at a node n_s where $\forall \tau_s \in n_s.\text{type}, \tau \in n.\text{type}$, τ_s, τ are not* statically recursive.

Figure 2 shows a simple example of the two ways a node is considered safe (represents an interesting point in the heap). In this figure we have node 1 which is *safe* since it is referred to directly by a variable. More interestingly we have nodes 2, 3 which both represent `LN` objects and are *statically recursive* but are also pointed to by the `Pair` object which is *not statically recursive* with the `LN` type. Thus according to our definition of safe nodes, nodes 1, 2 are considered safe and will not be merged.

3.3 Connectivity Awareness

Consider a program with the object types $\tau 1$, $\tau 2$, $\tau 3$ which are mutually recursive on the `nx` field. If we have the abstract heap graph in Figure 3a we can see that the 2^{nd} and 3^{rd} nodes in the list are statically recursive according to the definitions above but it is not complete. That is although types $\tau 2$ and $\tau 3$ are recursive each no object of a given type appears multiple times. Figure 3b shows a similar structure but in this case the 2^{nd} and 3^{rd} nodes in the list are statically recursive. Since the type $\tau 2$ appears multiple times (in node 2 and 3) these two nodes form a complete structure thus we want to replace this set of nodes with a single summary node.

To distinguish between these two cases we perform a connectivity aware detection of the recursive structures which takes connectivity and multiplicity into account ensuring that we only consider two nodes as being recursive if they are part of a *complete recursive structure*. This ensures only nodes that are in repeating and uninteresting parts of a recursive data structure are grouped together.

DEFINITION 4 (Complete Recursive Structure). *Two nodes n, n' are part of a* complete recursive structure *if:*

$\exists$ edge e from n to n', $\exists n_\tau$ and a path from n' to n_τ s.t. none of n, n', n_τ or the nodes on the path are safe, *and $n.\text{type} \cap n_\tau.\text{type} \neq \emptyset$.*

3.4 Recursive vs. Back Pointers.

Many programs use back pointers causing the above definition to identify any cyclic structure as recursive, since trivially every node can reach itself and thus every type appears multiple times. This causes the grouping of cycles in the graph into single nodes with the *layout (C)ycle*, which can lead to substantial imprecision. Figure 4 shows an example of such a heap. We can see that even though the abstract heap structure is finite, the back edge will cause our recursive component definition to group the 2^{nd} and 3^{rd} nodes into the same recursive component. To address this and similar problems that arise when distinguishing between bounded and unbounded structures when cyclic structures are present, we modify the recursive definition to ignore back edges.

3.5 Recursive Node Definition

Given the above scenarios and the proposed solutions for handling them we get the following final definition for determining if two nodes are recursive (that is they represent part of the same potentially recursive data structure on the heap).

DEFINITION 5 (Recursive Nodes). *Given the function* depth *which returns the depth of a node in the abstract heap graph, nodes n, n' (where $n \neq n'$) are* recursive *if:*

$\exists$ edge e from n to n', neither of n, n' are safe *and $\exists n_\tau$ s.t. there is a (possibly empty) path $\psi_r = \langle (n_1^s, n_1^e) \dots (n_k^s, n_k^e) \rangle$ from n' to n_τ s.t. $\forall (n_i^s, n_i^e) \in \psi_r, \text{depth}(n_i^s) < \text{depth}(n_i^e)$ (where* depth *is the depth of the node in the graph), $\forall (n_i^s, n_i^e)$, neither n_i^s or n_i^e is safe and, $n.\text{type} \cap n_\tau.\text{type} \neq \emptyset$.*

4. Composite Components and Array/Collection Grouping

The second contribution of this paper is a method to identify composite structures and equivalence classes of the objects stored in arrays or collections, which has not been studied as extensively as the problem of identifying recursive structures. The approach presented in this paper is based on the definition of a parametric predicate for determining if two nodes represent *equivalent* regions of the heap. The method presented in this section is based on the identification of heap regions based on connectivity information (and is sufficient for most optimization applications) as well as a parametric component which allows for the predicate to be tailored to support other applications as well (for example if we are using a numeric domain we can extend it to keep objects in an array with non-zero values in a given field distinct from objects that must have a zero in this field).

We introduce a notion of *equivalence* of two nodes that captures our intuition of when two nodes n, n' abstract similar regions of the concrete heap. Since the *equivalence* predicate is used to determine the maximum number of out edges each node may have, we can improve efficiency by minimizing the number of equivalence classes created by this relation. The tradeoff between precision and performance that we have found to be acceptable is determined by the following conditions: (1) are all the types represented by the nodes non-recursive (or may both nodes represent recursive types) and (2) what variables can access the objects in the regions abstracted by the nodes?

4.1 Recursive Similarity

Two nodes are *recursive similar* if they both abstract all non-recursive types or they both may abstract objects with recursive types. An example of why this is important is the common construction of k-ary trees using arrays/collections to hold either a recursive subtree or a non-recursive (with respect to the internal tree) leaf object.

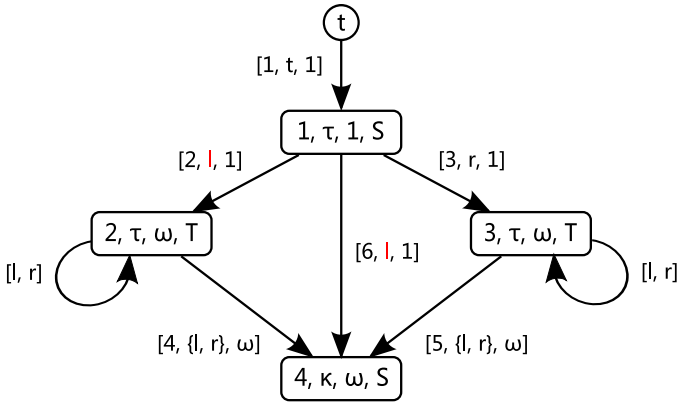
DEFINITION 6 (Recursive Similarity). *Given nodes n, n' and the* statically recursive *type information, n, n' are* recursive similar *iff either of the following holds:*

1. $\exists \tau \in n.\text{type}, \tau' \in n'.\text{type}$ *s.t.* τ, τ' *are statically recursive.*
2. *(*$\nexists \tau \in n.\text{type}, \tau$ *is statically recursive)* $\wedge$ *(*$\nexists \tau' \in n'.\text{type}, \tau'$ *is statically recursive)*

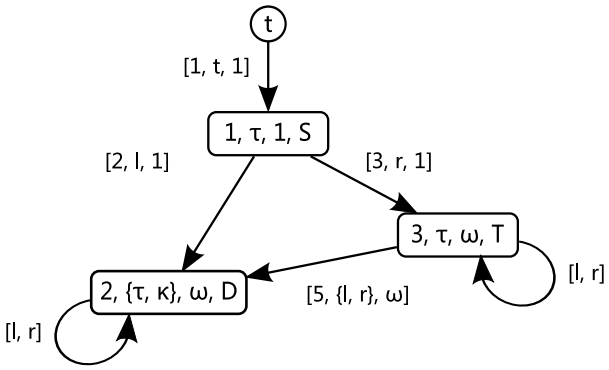
An example of where this heuristic applies is shown in Figure 5a. In this figure we have two types of objects τ, κ which both inherit from the superclass μ (a common way to build a tree structure in Object-Oriented Programing). The class τ is specialized to represent the internal tree structure (via the fields `l` and `r`) which point to objects of type μ. The class κ is the non-recursive leaf class which contains some value and may be referred to by multiple τ tree nodes.

In this case we want to make sure that not only do we distinguish the root node of the tree as well as the left and right sub-trees (which are preserved by the recursive structure identification heuristics in Section 3) but we also want to make sure that the analysis keeps the objects representing the internal tree structure in a disjoint region from the objects representing the leaf objects. Otherwise we would end up merging nodes 2 and 4, as they are both pointed to by an edge with *offset* `l` (highlighted in red if color is available) that starts at node 1 (Def. 8). This would result in a *DAG* region in node 2 and a loss of the overall tree structure as shown in Figure 5b.

At the other end of the range of possible similarity relations, if we were to ensure that regions with differing types were always kept separate the analysis would build unacceptably large tree



(a) Internal Tree and Leaves In Disjoint Regions



(b) Without Use of Recursive Similarity

Figure 5: Recursive Similarity

structures for many programs. For example a compiler may have a large number of classes that inherit from an `Expression` base class which appear in the parse tree structure and are treated uniformly by the program. If we maintained an abstract graph node for each of these types the tree would have a very large branching factor (and potentially depth) causing substantial performance degradation in the analysis.

4.2 Reference Similarity

If we have two nodes n, n' and the objects abstracted in the region by n are all stored in an array A and all the objects in the region abstracted by n' are stored in array A and a second array B then it is reasonable to assume that the programmer has partitioned these objects differently for some reason. Thus, we want to preserve this information by keeping the nodes distinct, we show this situation in Figure 6. We can ensure that the information on which collections and variables refer to which sets of objects is maintained by using the following definition of *reference similarity*.

DEFINITION 7 (Reference Similarity). *We say two nodes n, n' are reference similar if given the set of in edges to n, $E_{\text{in}} = \{e_1^n \ldots e_k^n\}$, the set of in edges to n', $E'_{\text{in}} = \{e_1^{n'} \ldots e_k^{n'}\}$, and the set of variables that can reach node n, $V_r = \{v_1^n \ldots v_r^n\}$, the set of variables that can reach node n', $V'_r = \{v_1^{n'} \ldots v_s^{n'}\}$, the following holds:*

$$(\{e.\text{offset} \mid e \in E_{\text{in}}\} = \{e'.\text{offset} \mid e' \in E'_{\text{in}}\}) \wedge (V_r = V'_r)$$

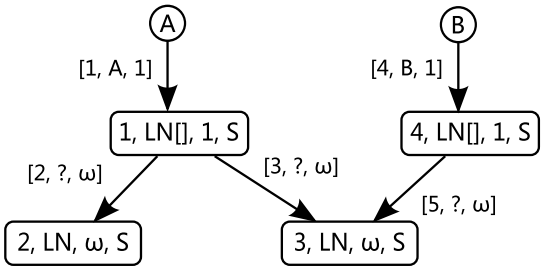


Figure 6: Nodes 2, 3 Not Reference Similar (based on variable reachability)

This definition ensures that if two nodes are treated differently with respect to the types of objects they are stored in or the variables that reach them then they are kept separate. In Figure 6 nodes 2 and 3 are not *reference similar* since node 2 is reachable from variable A while node 3 is reachable from both variables A and B.

4.3 Parametric Node Equivalence

In addition to using the structural information provided by the *recursive similar* and *reference similar* relations we can also provide a parametric component to the grouping operation to support the needs of more specific types of analysis. For example if we are checking a program to ensure that all file reads are exception free we want to distinguish `InputStream` objects that are open from those that are closed even if we have an array of such objects. Similarly if we are interested in checking locking properties we always want to distinguish between objects that are locked and those that are unlocked. Thus our definition allows parametric similarity properties to support specialized analyses that depend on precisely tracking differences of specific properties of interest for the objects in the program.

DEFINITION 8 (Equivalent Nodes/Edges). *Given the above definitions we define* edge equivalence. *Given a node n and two out edges e, e' which start at node n and end at nodes n_e and $n_{e'}$ respectively we say e, e' are* equivalent *if:*

1. e.offset $= e'$.offset
2. $n_e, n_{e'}$ *are* recursive similar
3. $n_e, n_{e'}$ *are* reference similar
4. $n_e, n_{e'}$ *are equivalent for all parametric similarity relations*

5. Region Identification and Grouping

Using the above definitions for identifying recursive structures, composite structures and grouping the contents of collections/arrays we define the method for constructing the logically related regions. Once we have identified a set of nodes that represent a logically related region, based on our region predicates, we need to replace them with a single node that *safely* approximates the properties of the nodes in the set.

5.1 Component Summarization

Before we present the complete region identification/normalization algorithm we describe how the summary nodes are computed. To simplify the computation we perform the summarization in a pairwise manner. When summarizing two nodes, n and n', there are three possibilities. The first is that there are no edges between the nodes, there are only edges in one direction between nodes (from n to n' or n' to n, but not both) and when there are edges from n to n' and from n' to n.

If there are no edges between the nodes we use the *mergeNoEdge* method to compute the summary representation. This method is a simple component-wise operation where the updated *type* label is the union of the two *type* sets, the *linearity* value is ω and the *layout* is the max (the most general) of the two *layout* labels. The case where there are edges from n to n' and from n' to n (*mergeBothWay*) is similar except we always assume the *layout* of the summary node is *(C)ycle* (while this is in general a significant over approximation we have found that the infrequency with which it is used makes this an acceptable definition).

The *mergeOneWay* operation (Algorithm 1) on a pair of nodes that have connecting edges is more complicated. In particular we need to account for the fact that the edge(s) connecting nodes n and n' will affect the *layout* of the new summary node.

Algorithm 1: mergeOneWay

input : graph g, n, n' nodes, *ebt* set of edges from n to n'
n.types $\leftarrow$ n.types $\cup$ n'.types;
n.linearity $\leftarrow$ ω;
n.layout $\leftarrow$ combineLayout(n.layout, n'.layout, *ebt*);
remap all edges incident to n' to be incident to n;
deleteNode(g, n');

The algorithm *combineLayout*(l, l', ebt), is based on a case analysis of the *layout* that results from the possible combinations of the *layouts* for n, n' along with the total number of pointers represented by *ebt* [20]. We enumerate the possible combinations of the *ebt* edges and the *layout* labels and then for each case we use the semantics of the edge and *layout* properties to determine the most general *layout* type that may result from this particular case. For example if we have two *(S)ingleton* nodes connected by an edge of *linearity* 1 then the most general *layout* for a node that summarizes these nodes and the edge is a *(L)ist*.

To merge two arbitrary nodes n, n' we use Algorithm 2 which selects the appropriate method for merging two nodes based on the existence of edges between them.

Algorithm 2: mergeNode

input : node n, n', graph g
if $\exists$ *edges from n to n' and n' to n* **then**
    mergeBothWay(g, n, n');
else if $\exists$ *edges from n to n'* **then**
    mergeOneWay(g, n, n', $\{e \mid e$ from n to $n'\}$);
else if $\exists$ *edges from n' to n* **then**
    mergeOneWay(g, n', n, $\{e \mid e$ from n' to $n\}$);
else
    mergeNoEdge(g, n', n);

5.2 Region Identification/Normalization Algorithm

Once we have the above methods for computing summary nodes for a pair of nodes in the graph we can define the final region identification algorithm. The resulting region grouped model is also a convenient normal form ensuring that the static analysis terminates as the infinite set of *labeled storage shape graphs* is a finite set under the normal form (recursive structures are represented by a bounded number of nodes and each node has a bounded number of out edges, for space we omit a formal proof).

The algorithm is a straightforward iterative identification of pairs of nodes/edges that should be grouped and the replacement of these structures by a summary representation until a fixpoint is reached. After this method terminates the abstract graph model will have all the logically related regions identified and grouped according to the characterizations in Sections 3 and 4.

Algorithm 3: groupRegions

```
input : graph g
while g is changing do
    while ∃ node n with edges e,e' s.t. e ≠ e' ∧ e,e' are
    equivalent edges do
        mergeNode(target of e, target of e', g);
        e.linearity ← ω;
        deleteEdge(g, e');
    while ∃ nodes n,n' that are recursive do
        mergeNode(g, n, n');
```

6. Case Study and Experimental Evaluation

In this section we look at two case studies that illustrate how the heuristics presented above allow the analysis to group heap objects into regions and how this information can be used to drive a range of memory management optimizations. Both benchmarks are taken from a version of the JOlden [2] suite.

6.1 Em3d

The first program we look at is Em3d which computes electromagnetic field values in a 3–dimensional space by constructing a list of `ENode` objects, each representing an electric field value and a second list of `ENode` objects, each of which represents a magnetic field value. To compute how the electric/magnetic field value for a given `ENode` object is updated at each time step the `computeNewValue` method uses an array of `ENode` objects from the opposite field and performs a convolution of these field values and a scaling vector, updating the current field value with the result. The main computation code is shown below:

```
void compute() {
   for(int i = 0; i < this.eNodes.size(); ++i)
     eNodes.get(i).computeNewValue();

   for(int i = 0; i < this.hNodes.size(); ++i)
     hNodes.get(i).computeNewValue();
}

void computeNewValue() {
   for(int i = 0; i < fromCount; i++)
     value -= coeffs[i] * fromNodes[i].value;
}
```

Figure 7 shows the heap structure that is constructed by the program and that is used in the main computation algorithm. To aid clarity we placed dashed lines around the composite structures that represent the magnetic field (in blue if color is available) and the electric field (in green). Variable `this` points to a single object of type `BiGrph`, which is the data structure that encapsulates all the objects of interest. The `BiGrph` object has 2 fields, the `hNodes` field pointing to a `Vector` of `ENode` objects that make up the magnetic field and, the `eNodes` field pointing to a `Vector` of `ENode` objects that make up the electric field. Each of these `ENode` objects has an array of `floats` and an array of `Enode` objects from the opposite field that are used to update the value of the field on each iteration of the field value computation loop. The region analysis identification techniques have precisely grouped all of the heap components in the program into the composite electric/magnetic field structures and even though the overall heap structure is cyclic the analysis has precisely resolved the bipartite graph structure. We note that in this example the definition of *safe nodes* due to non-recursive in edges is critical to ensuring the analysis resolves the heap into a bi-partite structure instead of merging many of the nodes into a single cyclic region.

While the heap is not further modified after construction, and thus there are no opportunities for improved memory collection, the above computation loop is an excellent candidate for altering memory layout to improve spatial locality of the memory accesses. This can be done statically by determining that the lifetimes of the `ENode` objects are bounded by the lifetime of the `Vector` they are stored in. Then at allocation time we can co-locate the `ENode` objects with the `Vector` [8]. Or we can use this information to provide support for the runtime reallocation of the `ENode` (and perhaps `ENode[]` or `float[]`) objects into contiguous memory pools based on the electric/magnetic structures they are in [12]. Our simple hand implementation of these optimizations on this benchmark resulted in approximately a 7-10% performance improvement, indicating that the information provided by the analysis is able to support sophisticated program transformations resulting in non-trivial performance improvements.

6.2 Barnes-Hut

The bh program performs a gravitational interaction simulation on a set of bodies (the `Body` objects) using a *fast-multipole* technique with a space decomposition tree. The tree is represented using `Cell` objects each of which has a `Vector` containing references to other `Cell` objects or references to the `Body` objects. The program also keeps two `Vector` objects for accessing the bodies, `bodyTab` and `bodyTabRev`. The positions (`pos`), velocities (`vel`) and acceleration (`acc`) values of the bodies are represented with composite structures consisting of a `MathVector` object and a `double[]`.

Using a common OOP idiom the `Cell` and `Body` objects both inherit from an abstract `Node` class. Thus, if we did not use the concept of *recursive similarity* to distinguish between references in the `Vector` collection to the recursive `Cell` objects which make up the tree structure and the non-recursive leaf `Body` objects the analysis would end up grouping the tree and the leaf objects into the same region. However, by distinguishing regions based on their *recursive similarity* the analysis has ensured that the tree structure and the leaf objects are grouped into different regions.

Figure 8 shows the abstract heap model built and used in the `stepSystem` method of the benchmark (the listing below), where the space decomposition tree is recomputed (the `makeTree` method), the body-body interactions are computed (the loop with the `hackGravity` method), and the new acceleration information is propagated (the `vprop` method).

```
public void stepSystem() {
   root = null;
   makeTree(nstep);

   Iterator<Body> bi = bodyTabRev.iterator();
   while (bi.hasNext())
     bi.next().hackGravity(rsize, root);

   vprop(bodyTabRev, nstep);
}
```

As we can see in Figure 8 the region identification algorithm is able to correctly identify and group all the major components in the overall heap structure. The space decomposition tree is grouped into the region represented by node 17 (although the analysis has overly conservatively assumed the structure may have a *(C)yclic* layout) while the leaf `Body` objects are represented separately by node 14. The analysis has also grouped the composite `MathVector`/`double[]` structures and has maintained the separation of these structures when they abstract distinct structures and are stored in different types or in different fields.

The bh program has many opportunities to apply the optimizations discussed in the introduction. In particular the information computed by the analysis in this paper enables opportunities that could not be previously exploited due to a lack of sufficiently precise region identification.

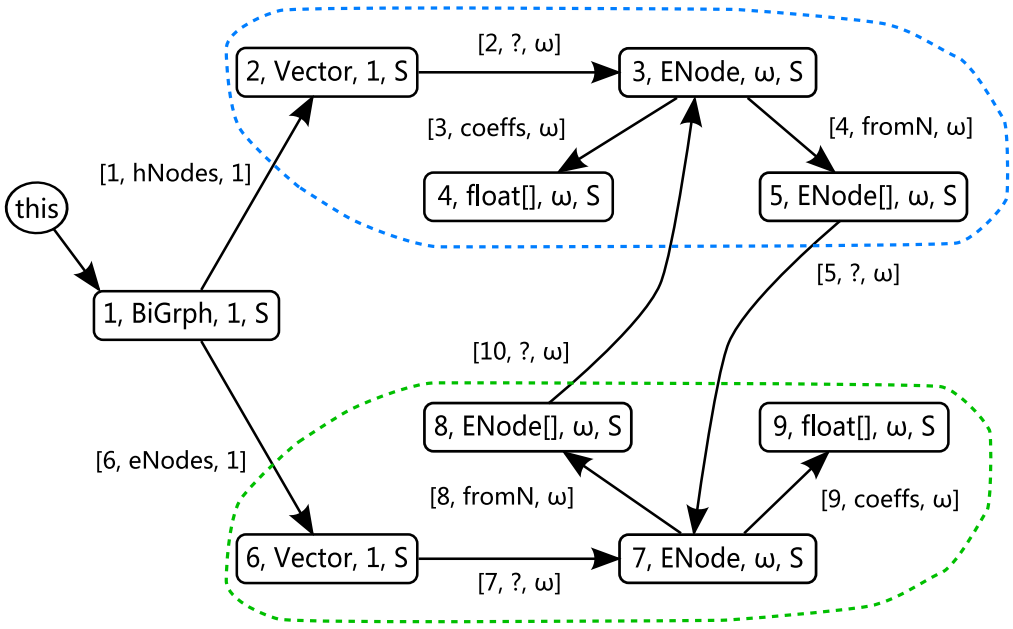


Figure 7: Abstract Heap in Em3d

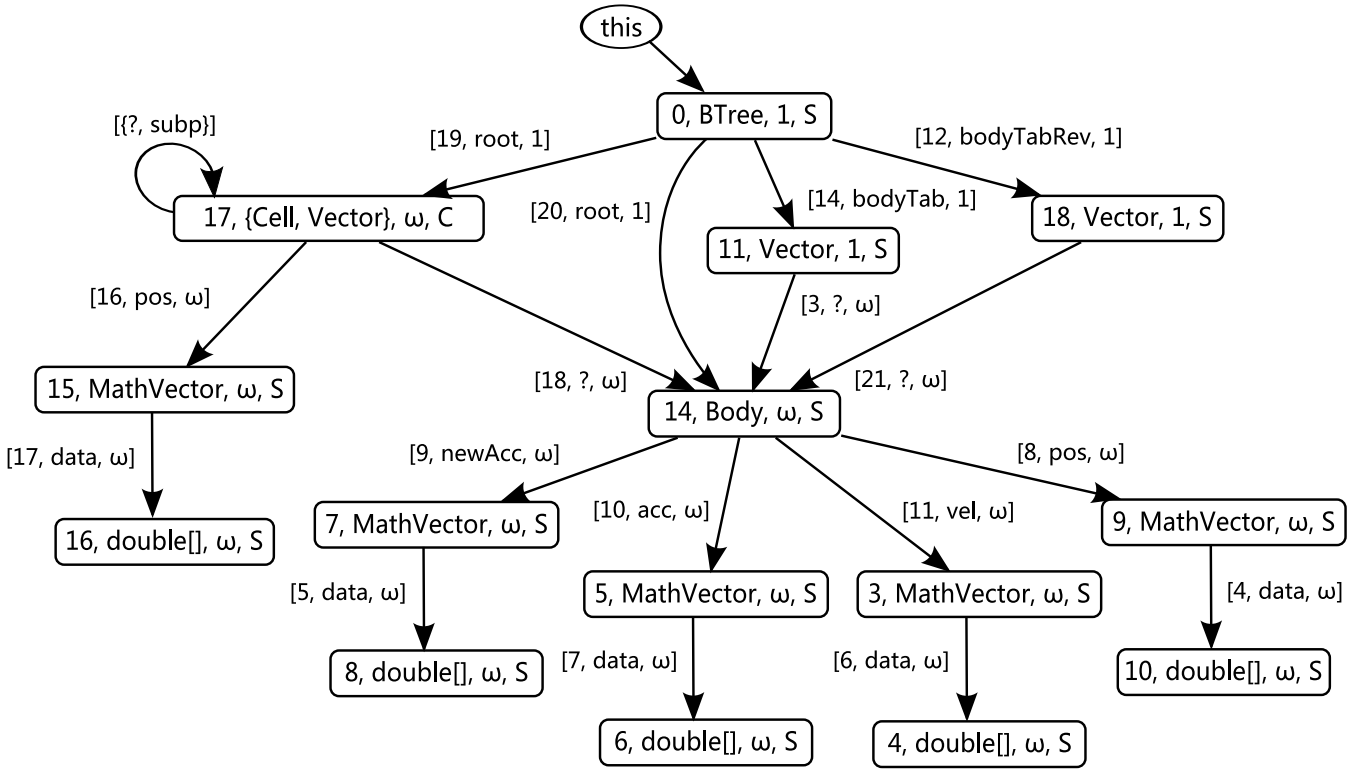


Figure 8: Abstract Heap in bh

The first possible optimization is the use of pool allocation [16] for the space decomposition tree (node 17) which is allocated in the `makeTree` method and then becomes dead at the `root = null` assignment on the next loop iteration. By pool allocating this we can collect the entire tree as one block instead of requiring the GC algorithm to traverse and collect each node in the tree one object at a time (reducing the number of objects that the garbage collector needs to reclaim by about 11%) and increasing the spatial locality of the accesses to the tree (which improves the performance of the program by 3-4% percent).

Given the structure of the heap, the two phases of computation and the limited pointer writes during the `hackGravity` method we can profitably apply parallel and region based collection [15]. This allows us to reduce the GC overhead by collecting dead `MathVector` objects in the regions for the `acc`, `vel`, and `pos` fields while the `newAcc` values are being computed in the `hackGravity` method. Similarly we can collect objects in the space decomposition tree and `newAcc` field regions while the mutator is in the `vprop` method. This parallel, region-specific collection greatly reduces the GC pause times while only requiring the collector/mutator to lock once on entry to these methods.

If we include *sharing* information as described in [22] we can determine that the `double[]` (where the size of the arrays is a small compile time constant) stored in the `MathVector` objects are never shared between `MathVector` objects and thus are good candidates for co-location [12, 8]. This has the beneficial effect of increasing the data locality and removing many redundant loads resulting in a 12% reduction in the runtime of the single threaded program, as well as reducing the size of the `MathVector`/Array composite structure object by a pointer (and the overhead of an array), resulting in a 37% reduction in memory usage.

Finally, if we again use the sharing information in [22] we can statically determine when each of the `MathVector`/`double[]` objects becomes dead and can insert explicit collection code for them [13]. This transformation reduces the number of objects that the GC needs to collect by a factor of about 52% (since these objects are immutable there are many of these created for each body object). If we perform this optimization with the pool allocation of the space decomposition tree then all of the objects can be collected statically eliminating the need for the collector entirely.

These transformations allow for the efficient collection (by collecting individual objects or entire pools) of all the dead objects created during this main computation portion and for the location of temporally related objects into contiguous parts of memory. Thus, this benchmark demonstrates how the precision of the region analysis presented in this paper enables the application of a number of powerful program optimizations that reduce the memory requirements, reduce garbage collection costs, and to improve the performance of the program.

6.3 Experimental Evaluation.

We have implemented a shape analyzer based on the region identification methods and instrumentation properties presented in this paper and evaluated the effectiveness and efficiency of the analysis on programs from SPECjvm98 [25] and a version of the JOlden suite. The JOlden suite contains pointer-intensive kernels that make use of recursive procedures, inheritance, and virtual methods. We modified the suite to use modern Java programming idioms. The benchmarks raytrace and db are taken from SPECjvm98.

The analysis algorithm was written in C++ and compiled using MSVC 8.0. The analysis was run on a 2.6 GHz Intel quad-core machine with 4 GB of RAM (although memory consumption never exceeded 120 MB).

For each of the benchmarks we provide a brief description of some of the major structures/features that are in the program.

Benchmark	LOC	Description	Analysis Time
bisort	560	Tree w/ Mod	0.26s
mst	668	Cycle w/ Struct.	0.12s
tsp	910	Tree to Cycle	0.15s
em3d	1103	Bipartite Graph	0.31s
perimeter	1114	Tree w/ Parent Ptr	0.91s
health	1269	Tree w/ Mod	1.25s
voronoi	1324	Cycle w/ Struct	1.80s
power	1752	Lists of Lists	0.36s
bh	2304	N-Body Sim w/ Mod	1.84s
db	1985	Shared/Mod Arrays	1.42s
raytrace	5809	Shared/Cycle/Tree	37.09s

Figure 9: LOC is for the normalized program representation including library stubs required by the analysis. Analysis Time is the analysis time for the analysis in seconds.

We mention the major data structures used (Trees, Lists of Lists, Cycles, etc.) and if the program heavily modifies the data structures (w/ Mod). Some of the benchmarks have slightly more nuanced structures — mst and voronoi which build globally cyclic structures that have significant local structure, bh which has a complex space-decomposition tree and sharing relations, and raytrace which builds a large multi-component structure which has cyclic structures, tree structures, and substantial sharing throughout. We also note that tsp and voronoi begin with tree structures and process them building up a final cyclic structure during the program. These benchmarks thus exercise a wide range of features in the analysis based on the types of structures built, modification of these structures, sharing of the structures, use of multi-component structures, and the use of arrays/collections.[1]

As our interest in this paper is primarily in the development of a heap analysis that can support a range of memory management and optimization techniques rather than in the performance of a specific GC method we focus on the cost of running the analysis to produce the region information. We note that the region information produced for all of the benchmarks is similar in quality to the results in the case studies (thus many of the same optimizations could be applied) and that the runtimes are on the order of seconds even for programs like bh and raytrace which make use of complex data structures, a number of classes from `java.util`/`java.io` and have nontrivial amounts of sharing between data structures.

7. Related Work

There has been a significant amount of work on developing static techniques to improve the allocation [16, 4], layout [12] and collection [16, 5, 13, 15] of memory in object oriented programs. These techniques have introduced a variety of methods for computing region information based on static partitions computed using a range of points-to analyses and are capable of scaling to large programs. However, the imprecision of fixed partitioning and flow insensitivity in parts of the analysis limits their ability to precisely analyze many programs that destructively rearrange regions and limits the ability to disambiguate components of larger composite structures (i.e. the 2 distinct regions of `ENode` objects in the overall cyclic heap structure in Em3d or disambiguating the `Body` objects from the space decomposition tree in Barnes-Hut). Thus the performance improvement achieved by the optimizations proposed in these papers, while good, is limited by the precision of the analysis results.

Other recent heap analysis work has focused on the precise modeling of destructive updates and their effect on the structure

[1] See `www.software.imdea.org/~marron/` for benchmark code, examples of the analysis results, and an executable analysis demo.

of the heap, TVLA [19, 18, 27, 24], separation logic based approaches [1, 28, 11]. While these techniques can model, with a very high degree of precision, many complex heap operations they currently impose limitations that make region analysis infeasible for many programs. In particular the current formulations are restricted to programs that manipulate lists (or trees) and restrict the amount of sharing between regions. As Separation Logic and TVLA are general purpose frameworks/logics the work in these papers could be extended as described in this work. However, to the best of our knowledge this extension has not been done. Thus, many of the benchmarks examined in this paper currently cannot be analyzed with these methods, including bh, em3d, voronoi, and raytrace, all of which have substantial opportunities for the application of various region based optimizations.

8. Conclusion

The analysis presented in this paper presents an important development in applying shape analysis techniques to real world programs as it can precisely and efficiently deal with the types of data structures and programmatic events that occur in realistic programs. In particular the formalization applies to any type of recursive data structures (as opposed to just lists or trees, and it supports composite data structures that have non-trivial sharing between them), it can precisely model many types of structures which are merged in simpler points-to style approaches, and it supports more precise grouping of the contents of collections (arrays or collections from `java.util`) than is possible with other methods.

Our experiments demonstrate that the proposed region identification method can be used to precisely and efficiently identify and group logically related regions of the heap (recursive data structures, composite structures composed of multiple objects and the contents of arrays/collections). Further our case studies demonstrate that the results of the analysis can be effectively used to support memory allocation/layout/collection optimization applications. Based on these results we believe that the proposed approach presents a basis for a heap analysis that can be used in practice to provide detailed heap information for a range of optimization applications that rely on region information and we are currently working on improving the practicality of the analysis by developing on techniques to scale it to larger programs.

Acknowledgements. We would like to thank the reviewers for their comments and suggestions. This work was funded in part by EU projects FET *HATS* and 06042-ESPASS, Spanish Ministry of Science and Industry projects TIN-2008-05624 *DOVES* and FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*. This work was also funded via NSF awards CCF-0541315, CNS-0831462, and CCF-0540600.

References

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[2] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.

[3] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[4] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, 2004.

[5] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *ISMM*, 2006.

[6] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *PLDI*, 1994.

[8] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI*, 2000.

[9] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[10] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.

[11] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.

[12] S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA*, 2004.

[13] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *PLDI*, 2006.

[14] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.

[15] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA*, 2003.

[16] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.

[17] C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[18] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.

[19] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *VMCAI*, 2005.

[20] M. Marron. Modeling the heap: A practical approach. Phd. thesis, University of New Mexico, 2008.

[21] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.

[22] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.

[23] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.

[24] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[25] Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[27] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.

[28] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *CAV*, 2008.

Precise Set Sharing Analysis for Java-Style Programs

Mario Méndez-Lojo[1] and Manuel V. Hermenegildo[1,2]

[1] University of New Mexico (USA)
[2] Technical University of Madrid (Spain)

Abstract. Finding useful *sharing* information between instances in object-oriented programs has recently been the focus of much research. The applications of such static analysis are multiple: by knowing which variables definitely do not share in memory we can apply conventional compiler optimizations, find coarse-grained parallelism opportunities, or, more importantly, verify certain correctness aspects of programs even in the absence of annotations. In this paper we introduce a framework for deriving precise sharing information based on abstract interpretation for a Java-like language. Our analysis achieves precision in various ways, including supporting multivariance, which allows separating different contexts. We propose a combined *Set Sharing + Nullity + Classes* domain which captures which instances do not share and which ones are definitively null, and which uses the classes to refine the static information when inheritance is present. The use of a set sharing abstraction allows a more precise representation of the existing sharings and is crucial in achieving precision during interprocedural analysis. Carrying the domains in a combined way facilitates the interaction among them in the presence of multivariance in the analysis. We show through examples and experimentally that both the set sharing part of the domain as well as the combined domain provide more accurate information than previous work based on pair sharing domains, at reasonable cost.

1 Introduction

The technique of Abstract Interpretation [8] has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to high- and low-level optimizations during program compilation, including program transformations. More recently, promising applications of such semantic approximations have been demonstrated in the more general context of program development, such as verification and static debugging.

Sharing analysis [14,20,26] aims to detect which variables do not share in memory, i.e., do not point (transitively) to the same location. It can be viewed as an abstraction of the graph-based representations of memory used by certain classes of alias analyses (see, e.g., [31,5,13,15]). Obtaining a safe (over-) approximation of which instances might share allows parallelizing segments of code,

F. Logozzo et al. (Eds.): VMCAI 2008, LNCS 4905, pp. 172–187, 2008.

improving garbage collection, reordering execution, etc. Also, sharing information can improve the precision of other analyses.

Nullity analysis is aimed at keeping track of null variables. This allows for example verifying properties such as the absence of null-pointer exceptions at compile time. In addition, by combining sharing and null information it is possible to obtain more precise descriptions of the state of the heap.

In type-safe, object-oriented languages *class* analysis [1,3,10,22], (sometimes called *type* analysis) focuses on determining, in the presence of polymorphic calls, which particular implementation of a given method will be executed at runtime, i.e., what is the specific class of the called object in the hierarchy. Multiple compilation optimizations benefit from having precise class descriptions: inlining, dead code elimination, etc. In addition, class information may allow analyzing only a subset of the classes in the hierarchy, which may result in additional precision.

We propose a novel analysis which infers in a combined way *set sharing*, *nullity*, and *class* information for a subset of Java that takes into account most of its important features: inheritance, polymorphism, visibility of methods, etc. The analysis is multivariant, based on the algorithm of [21], which allows separating different contexts, thus increasing precision. The additional precision obtained from context sensitivity has been shown to be important in practice in the analysis of object-oriented programs [30].

The objective of using a reduced cardinal product [9] of these three abstract domains is to achieve a good balance between precision and performance, since the information tracked by each component helps refine that of the others. While in principle these three analyses could be run separately, because they interact (we provide some examples of this), this would result in a loss of precision or require an expensive iteration over the different analyses until an overall fixpoint is reached [6,9]. In addition note that since our analysis is multivariant, and given the different nature of the properties being tracked, performing analyses separately may result in different sets of abstract values (contexts) for each analysis for each program point. This makes it difficult to relate which abstract value of a given analysis corresponds to a given abstract value of another analysis at a given point. At the other end of things, we prefer for clarity and simplicity reasons to develop directly this three-component domain and the operations on it, rather than resorting to the development of a more unified domain through (semi-)automatic (but complex) techniques [6,7]. The final objectives of our analysis include verification, static debugging, and optimization.

The closest related work is that of [26] which develops a *pair*-sharing [27] analysis for object-oriented languages and, in particular, Java. Our description of the (set-)sharing part of our domain is in fact based on their elegant formalization. The fundamental difference is that we track *set* sharing instead of *pair* sharing, which provides increased accuracy in many situations and can be more appropriate for certain applications, such as detecting independence for program parallelization. Also, our domain and abstract semantics track additionally nullity and classes in a combined fashion which, as we have argued above, is

$$
\begin{array}{lll}
\mathit{prog} & ::= & \mathit{class\_decl}^* \\
\mathit{class\_decl} & ::= & \texttt{class}\ \mathrm{k}_1\ [\texttt{extends}\ \mathrm{k}_2]\ \mathit{decl}^*\ \mathit{meth\_decl}^* \\
\mathit{meth\_decl} & ::= & \mathit{vbty}\ (\mathrm{t}_{ret}|\texttt{void})\ \mathrm{meth}\ \mathit{decl}^*\ \mathit{com} \\
\mathit{vbty} & ::= & \texttt{public} \mid \texttt{private} \\
\mathit{com} & ::= & \mathrm{v} = \mathit{expr} \mid \mathrm{v}.f = \mathit{expr} \\
 & & \mid \mathit{decl} \mid \texttt{skip} \\
 & & \mid \texttt{return}\ \mathit{expr} \mid \mathit{com}\,;\mathit{com} \\
 & & \mid \texttt{if}\ \mathrm{v}\ (== \mid !=)\ (\texttt{null}|\mathrm{w})\ \mathit{com}\ \texttt{else}\ \mathit{com} \\
\mathit{decl} & ::= & \mathrm{v{:}t} \\
\mathit{var\_lit} & ::= & \mathrm{v} \mid \mathrm{a} \\
\mathit{expr} & ::= & \texttt{null} \mid \texttt{new}\ \mathrm{k} \mid \mathrm{v}.f \mid \mathrm{v.m}(\mathrm{v}_1, \ldots \mathrm{v}_n) \mid \mathit{var\_lit}
\end{array}
$$

Fig. 1. Grammar for the language

particularly useful in the presence of multivariance. In addition, we deal directly with a larger set of object features such as inheritance and visibility. Finally, we have implemented our domains (as well as the pair sharing domain of [26]), integrated them in our multivariant analysis and verification framework [17], and benchmarked the system. Our experimental results are encouraging in the sense that they seem to support that our contributions improve the analysis precision at reasonable cost.

In [23,24], the authors use a *distinctness* domain in the context of an abstract interpretation framework that resembles our sharing domain: if two variables point to different abstract locations, they do not share at the concrete level. Their approach is closer to shape analysis [25] than to sharing analysis, which can be inferred from the former. Although information retrieved in this way is generally more precise, it is also more computationally demanding and the abstract operations are more difficult to design. We also support some language constructs (e.g., visibility of methods) and provide detailed experimental results, which are not provided in their work.

Most recent work [28,18,30] has focused on context-sensitive approaches to the points-to problem for Java. These solutions are quite scalable, but flow-insensitive and overly conservative. Therefore, a verification tool based on the results of those algorithms may raise spurious warnings. In our case, we are able to express sharing information in a safe manner, as invariants that all program executions verify at the given program point.

2 Standard Semantics

The source language used is defined as a subset of Java which includes most of its object-oriented (inheritance, polymorphism, object creation) and specific (e.g., access control) features, but at the same time simplifies the syntax, and does not deal with interfaces, concurrency, packages, and static methods or variables. Although we support primitive types in our semantics and implementation, they will be omitted from the paper for simplicity.

```
class Element {
   int value;
   Element next;}

class Vector {
   Element first;

    public void add(Element el) {
        Vector v  = new Vector();
        el.next  = null;
        v.first = el;
        append(v);
    }
}
```

```
public void append(Vector v) {

  if (this != v) {
      Element e = first;
      if (e == null)
          first = v.first;
      else {
         while (e.next != null)
            e = e.next;
         e.next = v.first;
      }
  }
}
```

Fig. 2. Vector example

The rules for the grammar of this language are listed in Fig. 1. The `skip` statement, not present in the Java standard specification [11], has the expected semantics. Fig. 2 shows an example program in the supported language, an alternative implementation for the `java.util.Vector` class of the JDK in which vectors are represented as linked lists. Space constraints prevent us from showing the full code here,[1] although the figure does include the relevant parts.

2.1 Basic Notation

We first introduce some notation and auxiliary functions used in the rest of the paper. By $\mapsto$ we refer to total functions; for partial ones we use $\rightarrow$. The powerset of a set s is $\mathcal{P}(s)$; $\mathcal{P}^+(s)$ is an abbreviation for $\mathcal{P}(s) \setminus \{\emptyset\}$. The *dom* function returns all the elements for which a function is defined; for the codomain we will use *rng*. A substitution $f[k_1 \mapsto v_1, \ldots, k_n, \mapsto v_n]$ is equivalent to $f(k_1) = v_1, \ldots, f(k_n) = v_n$. We will overload the operator for lists so that $f[K \mapsto V]$ assigns $f(k_i) = v_i$, $i = 1, \ldots, m$, assuming $|K| = |V| = m$. By $f|_{\_S}$ we denote removing S from $dom(f)$. Conversely, $f|_S$ restricts $dom(f)$ to S. For tuples $(f_1, \ldots, f_m)|_S = (f_1|_S, \ldots, f_m|_S)$. Renaming in the set s of every variable in S by the one in the same position in T ($|S| = |T|$) is written as $s|_S^T$. This operator can also be applied for renaming single variables. We denote by $\mathcal{B}$ the set of Booleans.

2.2 Program State and Sharing

With $\mathcal{M}$ we designate the set of all method names defined in the program. For the set of distinct identifiers (variables and fields) we use $\mathcal{V}$. We assume that $\mathcal{V}$ also includes the elements *this* (instance where the current method is executed),

[1] Full source code for the example can be found in
`http://www.clip.dia.fi.upm.es/~mario`

and *res* (for the return value of the method). In the same way, $\mathcal{K}$ represents the program-defined classes. We do not allow `import` declarations but assume as member of $\mathcal{K}$ the predefined class `Object`.

$\mathcal{K}$ forms a lattice implied by a subclass relation $\downarrow$: $\mathcal{K} \rightarrow \mathcal{P}(\mathcal{K})$ such that if $t_2 \in \downarrow t_1$ then $t_2 \leq_{\mathcal{K}} t_1$. The semantics of the language implies $\downarrow\texttt{Object} = \mathcal{K}$. Given $def : \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$, that determines whether a particular class provides its own implementation for a method, the Boolean function $redef : \mathcal{K} \times \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$ checks if a class k_1 redefines a method existing in the ancestor k_2: $redef(k_1, k_2, \mathrm{m}) = true$ iff $\exists k$ s.t. $def(k, \mathrm{m})$, $k_1 \leq_{\mathcal{K}} k <_{\mathcal{K}} k_2$.

Static types are accessed by means of a function $\pi : \mathcal{V} \mapsto \mathcal{K}$ that maps variables to their declared types. The purpose of an *environment* π is twofold: it indicates the set of variables accessible at a given program point and stores their declared types. Additionally, we will use the auxiliary functions $F(k)$ (which maps the fields of $k \in \mathcal{K}$ to their declared type), and $type_\pi(expr)$, which maps expressions to types, according to π.

The description of the memory state is based on the formalization in [26,12]. We define a frame as any element of $Fr_\pi = \{\, \phi \mid \phi \in dom(\pi) \mapsto Loc \cup \{\mathrm{null}\}\}$, where $Loc = \mathbb{I}^+$ is the set of memory locations. A frame represents the first level of indirection and maps variable names to locations except if they are null. The set of all objects is $Obj = \left\{\, k \star \phi \mid k \in \mathcal{K}, \phi \in Fr_{F(k)} \right\}$. Locations and objects are linked together through the memory $Mem = \{\, \mu \mid \mu \in Loc \mapsto Obj\}$. A new object of class k is created as $new(k) = k \star \phi$ where $\phi(f) = null \;\; \forall f \in F(k)$. The object pointed to by v in the frame ϕ and memory μ can be retrieved via the partial function $obj(\phi \star \mu, v) = \mu(\phi(v))$. A valid heap configuration (concrete state $\phi \star \mu$) is any element of $\Sigma_\pi = \{\, (\phi \star \mu) \mid \phi \in Fr_\pi, \mu \in Mem\}$. We will sometimes refer to a pair $(\phi \star \mu)$ with δ.

The set of locations $R_\pi(\phi \star \mu, v)$ reachable from $v \in dom(\pi)$ in the particular state $\phi \star \mu \in \Sigma_\pi$ is calculated as $R_\pi(\phi \star \mu, v) = \cup \left\{\, R^i_\pi(\phi \star \mu, v) \;\middle|\; i \geq 0 \right\}$, the base case being $R^0_\pi(\phi \star \mu, v) = \{(\phi(v))|_{Loc}\}$ and the inductive one $R^{i+1}_\pi(\phi \star \mu, v) = \cup \left\{\, rng(\mu(l).\phi))|_{Loc} \mid l \in R^i_\pi(\phi \star \mu, v) \right\}$. Reachability is the basis of two fundamental concepts: sharing and nullity. Distinct variables $V = \{v_1, \ldots, v_n\}$ *share* in the actual memory configuration δ if there is at least one common location in their reachability sets, i.e., $share_\pi(\delta, V)$ is true iff $\cap_{i=1}^n R_\pi(\delta, v_i) \neq \emptyset$. A variable $v \in dom(\pi)$ is *null* in state δ if $R_\pi(\delta, v) = \emptyset$. Nullity is checked by means of $nil_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{B}$, defined as $nil_\pi(\phi \star \mu, v) = true$ iff $\phi(v) = null$.

The *run-time type* of a variable in scope is returned by $\psi_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{K}$, which associates variables with their dynamic type, based on the information contained in the heap state: $\psi_\pi(\delta, v) = obj(\delta, v).k$ if $\overline{nil_\pi}(\delta, v)$ and $\psi_\pi(\delta, v) = \pi(v)$ otherwise. In a type-safe language like Java runtime types are congruent with declared types, i.e., $\psi_\pi(\delta, v) \leq_{\mathcal{K}} \pi(v) \;\; \forall v \in dom(\pi), \forall \delta \in \Sigma_\pi$. Therefore, a correct approximation of ψ_π can always be derived from π. Note that at the same program point we might have different run-time type states ψ^1_π and ψ^2_π depending on the particular program path executed, but the static type state is unique.

Denotational (compositional) semantics of sequential Java has been the subject of previous work (e.g., [2]). In our case we define a simpler version of that semantics for the subset defined in Sect. 2, described as transformations in the frame-memory state. The descriptions are similar to [26]. Expression functions $\mathcal{E}^I_\pi[\![]\!] : expr \mapsto (\Sigma_\pi \mapsto \Sigma_{\pi'})$ define the meaning of Java expressions, augmenting the actual scope $\pi' = \pi[res \mapsto type_\pi(exp)]$ with the temporal variable res. Command functions $\mathcal{C}^I_\pi[\![]\!] : com \mapsto (\Sigma_\pi \mapsto \Sigma_\pi)$ do the same for commands; semantics of a method `m` defined in class k is returned by the function $I(k.\mathtt{m}) : \Sigma_{input(k.m)} \to \Sigma_{output(k.m)}$. The definition of the respective environments, given a declaration in class k as t_{ret} `m`$(this : k, p_1 : t_1 \ldots p_n : t_n)$ `com`, is $input(k.m) = \{this \mapsto k, p_1 \mapsto t_1, \ldots, p_n \mapsto t_n\}$ and $output(k.m) = input(k.m)[out \mapsto t_{ret}]$.

Example 1. Assume that, in Figure 2, after entering in the method `add` of the class `Vector` we have an initial state $(\phi_0 \star \mu_0)$ s.t. $loc_1 = \phi_0(el) \neq null$. After executing `Vector v = new Vector()` the state is $(\phi_1 \star \mu_1)$, with $\phi_1(v) = loc_2$, and $\mu_1(loc_2).\phi(first) = null$. The field assignment `el.next = null` results in $(\phi_2 \star \mu_2)$, verifying $\mu_2(loc_1).\phi(next) = null$. In the third line, `v.first = el` links loc_1 and loc_2 since now $\mu_3(loc_2).\phi(first) = loc_1$. Now v and el share, since their reachability sets intersect *at least* in $\{loc_1\}$. Finally, assume that `append` attaches v to the end of the current instance $this$ resulting in a memory layout $(\phi_4 \star \mu_4)$. Given $loc_3 = obj((\phi_4 \star \mu_4)(this)).\phi(first)$, it should hold that $\mu_4(\ldots \mu_4(loc_3).\phi(next) \ldots).\phi(next) = loc_2$. Now $this$ shares with v and therefore with el, because loc_1 is reachable from loc_2.

3 Abstract Semantics

An abstract state $\sigma \in D_\pi$ in an environment π approximates the sharing, nullity, and run-time type characteristics (as described in Sect. 2.2) of set of concrete states in Σ_π. Every abstract state combines three abstractions: a sharing set $sh \in \mathcal{DS}_\pi$, a nullity set $nl \in \mathcal{DN}_\pi$, and a type member $\tau \in \mathcal{DT}_\pi$, i.e., $D_\pi = \mathcal{DS}_\pi \times \mathcal{DN}_\pi \times \mathcal{DT}_\pi$.

The sharing abstract domain $\mathcal{DS}_\pi = \{\{v_1, \ldots, v_n\} \mid \{v_1, \ldots, v_n\} \in \mathcal{P}(dom(\pi)), \cap_{i=1}^n C_\pi(v_i) \neq \emptyset\}$ is constrained by a class reachability function which retrieves those classes that are reachable from a particular variable: $C_\pi(v) = \cup\{C^i_\pi(v) \mid i \geq 0\}$, given $C^0_\pi(v) = \downarrow\pi(v)$ and $C^{i+1}_\pi(v) = \cup\{rng(F(k)) \mid k \in C^i_\pi(v)\}$. By using class reachability, we avoid including in the sharing domain sets of variables which cannot share in practice because of the language semantics. The partial order $\leq_{\mathcal{DS}_\pi}$ is set inclusion.

We define several operators over sharing sets, standard in the sharing literature [14,19]. The binary union $\uplus : \mathcal{DS}_\pi \times \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$, calculated as $S_1 \uplus S_2 = \{Sh_1 \cup Sh_2 \mid Sh_1 \in S_1, Sh_2 \in S_2\}$ and the closure under union $* : \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$ operators, defined as $S^* = \{\cup SSh \mid SSh \in \mathcal{P}^+(S)\}$; we later filter their results using class reachability. The relevant sharing with respect to v is $sh_v = \{s \in sh \mid v \in s\}$, which we overloaded for sets. Similarly, $sh_{-v} = \{s \in sh \mid v \notin s\}$. The projection $sh|_V$ is equivalent to $\{S \mid S = S' \cap V, S' \in sh\}$.

$$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](sh, nl, \tau) = (sh, nl', \tau')$$
$$nl' = nl[res \mapsto null]$$
$$\tau' = \tau[res \mapsto \downarrow object]$$

$$\mathcal{SE}^I_\pi[\![\texttt{new } k]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = sh \cup \{\{res\}\}$$
$$nl' = nl[res \mapsto nnull]$$
$$\tau' = \tau[res \mapsto \{\kappa\}]$$

$$\mathcal{SE}^I_\pi[\![v]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = (\{\{res\}\} \uplus sh_v) \cup sh_{-v}$$
$$nl' = nl[res \mapsto nl(v)]$$
$$\tau' = \tau[res \mapsto \tau(v)]$$

$$\mathcal{SE}^I_\pi[\![v\,.\,\texttt{f}]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ (sh', nl', \tau') \text{ otherwise} \end{cases}$$
$$sh' = sh_{-v} \cup \bigcup\{\mathcal{P}^+(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh_v\}$$
$$nl' = nl[res \mapsto unk, v \mapsto nnull]$$
$$\tau' = \tau[res \mapsto\downarrow F(\pi(v)(\texttt{f}))]$$

$$\mathcal{SE}^I_\pi[\![v\,.\,\texttt{m}(v_1, \ldots, v_n)]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ \sigma' \text{ otherwise} \end{cases}$$
$$\sigma' = \mathcal{SE}^I_\pi[\![\texttt{call}(v, m(v_1, \ldots, v_n))]\!](sh, nl', \tau)$$
$$nl' = nl[v \mapsto nnull]$$

Fig. 3. Abstract semantics for the expressions

The nullity domain is $\mathcal{DN}_\pi = \mathcal{P}(dom(\pi) \mapsto \mathcal{NV})$, where $\mathcal{NV} = \{null, nnull, unk\}$. The order $\leq_{\mathcal{NV}}$ of the nullity values ($null \leq_{\mathcal{NV}} unk$, $nnull \leq_{\mathcal{NV}} unk$) induces a partial order in $\mathcal{DN}_\pi$ s.t. $nl_1 \leq_{\mathcal{DN}_\pi} nl_2$ if $nl_1(v) \leq_{\mathcal{NV}} nl_2(v)$ $\forall v \in dom(\pi)$. Finally, the domain of types maps variables to sets of types congruent with π: $\mathcal{DT}_\pi = \{(v, \{t_1, \ldots, t_n\}) \in dom(\pi) \mapsto \mathcal{P}(\mathcal{K}) \mid \{t_1, \ldots, t_n\} \subseteq \downarrow\pi(v)\}$.

We assume the standard framework of abstract interpretation as defined in [8] in terms of Galois insertions. The concretization function $\gamma_\pi : D_\pi \mapsto \mathcal{P}(\Sigma_\pi)$ is $\gamma_\pi(sh, nl, \tau) = \{\delta \in \Sigma_\pi |\ \forall V \subseteq dom(\pi), share_\pi(\delta, V)$ and $\nexists W, V \subset W \subseteq dom(\pi)$ s.t. $share_\pi(\delta, W) \Rightarrow V \in sh$, and $R_\pi(\delta, v) = \emptyset$ if $nl(v) = null$, and $R_\pi(\delta, v) \neq \emptyset$ if $nl(v) = nnull$, and $\psi_\pi(\delta, v) \in \tau(v)$, $\forall v \in dom(\pi)\}$.

The abstract semantics of expressions and commands is listed in Figs. 3 and 4. They correctly approximate the standard semantics, as proved in [16]. As their concrete counterparts, they take an expression or command and map an input state $\sigma \in D_\pi$ to an output state $\sigma' \in D^{\sigma}_{\pi'}$ where $\pi = \pi'$ in commands and $\pi' = \pi[res \mapsto type_\pi(expr)]$ in expression $expr$. The semantics of a method call is explained in Sect. 3.1. The use of set sharing (rather than pair sharing) in the semantics prevents possible losses of precision, as shown in Example 2.

Example 2. In the `add` method (Fig. 2), assume that $\sigma = (\{\{this, el\}, \{v\}\}, \{this/nnull, el/nnull, v/nnull\})$ right before evaluating `el` in the third line (we skip type information for simplicity). The expression `el` binds to res the location of el, i.e., forces el and res to share. Since $nl(el) \neq null$ the new sharing is $sh' = (\{\{res\}\} \uplus sh_{el}) \cup sh_{-el} = (\{\{res\}\} \uplus \{\{this, el\}\}) \cup \{\{v\}\} = \{\{res, this, el\}, \{v\}\}$.

$$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!]\sigma = ((sh'|_{-v})|^v_{res}, nl'|^v_{res}, \tau''|_{-res})$$
$$\tau'' = \tau'[v \mapsto (\tau'(v) \cap \tau'(res))]$$
$$(sh', nl', \tau') = \mathcal{SE}^I_\pi[\![expr]\!]\sigma$$

$$\mathcal{SC}^I_\pi[\![v\texttt{.f=}expr]\!]\sigma = (sh'', nl'', \tau')|_{-res}$$
$$sh'' = \begin{cases} \bot & \text{if } nl'(v) = null \\ sh' & \text{if } nl'(res) = null \\ sh^y \cup sh'_{-\{v,res\}} & \text{otherwise} \end{cases}$$
$$nl'' = nl'[v \mapsto nnull]$$
$$sh^y = (\bigcup\{\mathcal{P}(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh'_v\} \cup \bigcup\{\mathcal{P}(s|_{-res} \cup \{v\}) \uplus \{\{res\}\} \mid s \in sh'_{res}\})^*$$
$$(sh', nl', \tau') = \mathcal{SE}^I_\pi[\![expr]\!]\sigma$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!]\sigma = \begin{cases} \sigma'_1 & \text{if } nl(v) = null \\ \sigma'_2 & \text{if } nl(v) = nnull \\ \sigma_1 \sqcup \sigma_2 & \text{if } nl(v) = unk \end{cases}$$
$$\sigma'_i = \mathcal{SC}^I_\pi[\![com_i]\!]\sigma$$
$$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](sh|_{-v}, nl[v \mapsto null], \tau[v \mapsto \downarrow\pi(v)])$$
$$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](sh, nl[v \mapsto nnull], \tau)$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](sh, nl, \tau) = \begin{cases} \sigma'_1 & \text{if } nl(v) = nl(w) = null \\ \sigma'_2 & \text{if } sh|_{\{v,w\}} = \emptyset \\ \sigma'_1 \sqcup \sigma'_2 & \text{otherwise} \end{cases}$$
$$\sigma'_i = \mathcal{SC}^I_\pi[\![com_i]\!](sh, nl, \tau)$$

$$\mathcal{SC}^I_\pi[\![com_1\texttt{;}com_2]\!]\sigma = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!]\sigma)$$

Fig. 4. Abstract semantics for the commands

In the case of pair-sharing, the transfer function [26] for the same initial state $sh = \{\{this, el\}, \{v, v\}\}$ returns $sh'_p = \{\{res, el\}, \{res, this\}, \{this, el\}, \{v, v\}\}$, which translated to set sharing results in $sh'' = \{\{res, el\}, \{res, this\}, \{res, this, el\}, \{this, el\}, \{v\}\}$, a less precise representation (in terms of $\leq_{\mathcal{DS}_\pi}$) than sh'.

Example 3. Our multivariant analysis keeps two different call contexts for the `append` method in the `Vector` class (Fig. 2). Their different sharing information shows how sharing can improve nullity results. The first context corresponds to external calls (invocation from other classes), because of the `public` visibility of the method: $\sigma_1 = (\{\{this\}, \{this, v\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. The second corresponds to an internal (within the class) call, for which the analysis infers that *this* and v do not share: $\sigma_2 = (\{\{this\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. Inside `append`, we avoid creating a circular list by checking that $this \neq v$. Only then is the last element of *this* linked to the first one of v. We use `com` to represent the series of commands `Element e = first; if (e==null)...else..` and `bdy` for the whole body of the method. Independently of whether the input state is σ_1 or σ_2 our analysis infers that $\mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_1 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_2 = (\{\{this, v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\}) = \sigma_3$. However, the more precise sharing information in σ_2 results in a more precise analysis

Algorithm 1. Extend operation

input : state before the call σ, result of analyzing the call σ_λ and actual parameters A
output: resulting state σ_f

if $\sigma_\lambda = \bot$ **then**
  $\sigma_f = \bot$
else
  let $\sigma = (sh, nl, \tau)$, and $\sigma_\lambda = (sh_\lambda, nl_\lambda, \tau_\lambda)$, and $AR = A \cup \{res\}$

$$
\begin{aligned}
star &= (sh_A \cup \{\{res\}\})^* \\
sh_{ext} &= \{s \mid s \in star, s|_{AR} \in sh_\lambda\} \\
sh_f &= sh_{ext} \cup sh_{-A} \\
nl_f &= nl[res \mapsto nl_\lambda(res)] \\
\tau_f &= \tau[res \mapsto \tau_\lambda(res)] \\
\sigma_f &= (sh_f, nl_f, \tau_f)
\end{aligned}
$$

end

of `bdy`, because of the guard `(this!=v)`. In the case of the external calls, $\mathcal{SC}^I_\pi[\![\texttt{bdy}]\!]\sigma_1 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_1 \sqcup \mathcal{SC}^I_\pi[\![\texttt{skip}]\!]\sigma_1 = \sigma_1 \sqcup \sigma_3 = \sigma_1$. When the entry state is σ_2, the semantics at the same program point is $\mathcal{SC}^I_\pi[\![\texttt{bdy}]\!]\sigma_2 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_2 = \sigma_3 < \sigma_1$. So while the internal call requires $v \neq null$ to terminate, we cannot infer the final nullity of that parameter in a public invocation, which might finish even if v is null.

3.1 Method Calls

The semantics of the expression $\texttt{call}(v, m(v_1, \ldots, v_n))$ in state $\sigma = (sh, nl, \tau)$ is calculated by implementing the top-down methodology described in [21]. We will assume that the formal parameters follow the naming convention F in all the implementations of the method; let $A = \{v, v_1, \ldots, v_n\}$ and $F = dom(input(k.m))$ be ordered lists. We first calculate the projection $\sigma_p = \sigma|_A$ and an entry state $\sigma_y = \sigma_p|^F_A$. The abstract execution of the call takes place only in the set of classes $K = \tau(v)$, resulting in an exit state $\sigma_x = \bigsqcup\{\mathcal{SC}^I_\pi[\![k'.m]\!]\sigma_y | k' = lookup(k, m), k \in K\}$, where *lookup* returns the body of k's implementation of m, which can be defined in k or inherited from one of its ancestors. The abstract execution of the method in a subset $K \subseteq \downarrow\pi(v)$ increases analysis precision and is the ultimate purpose of tracking run-time types in our abstraction. We now remove the local variables $\sigma_b = \sigma_x|_{F \cup \{out\}}$ and rename back to the scope of the caller: $\sigma_\lambda = \sigma_b|^{A \cup \{res\}}_{F \cup \{out\}}$; the final state σ_f is calculated as $\sigma_f = extend(\sigma, \sigma_\lambda, A)$. The $extend : D_\pi \times D_\pi \times \mathcal{P}(dom(\pi)) \mapsto D_\pi$ function is described in Algorithm 1.

In Java references to objects are passed by value in a method call. Therefore, they cannot be modified. However, the call might introduce new sharing between actual parameters through assignments to their fields, given that the formal parameters they correspond to have not been reassigned. We keep the original information by copying all the formal parameters at the beginning of each call,

as suggested in [23]. Those copies cannot be modified during the execution of the call, so a meaningful correspondence can be established between A and F.

We can do better by realizing that analysis might refine the information about the actual parameters within a method and propagating the new values discovered back to σ_f. For example, in a method `foo(Vector v){if v!=null skip else throw_null}`, it is clear that we can only finish normally if $nl_x(v) = nnull$, but in the actual semantics we do not change the nullity value for the corresponding argument in the call, which can only be more imprecise. Note that the example is different from `foo(Vector v){v = new Vector}`, which also finishes with $nl_x(v) = nnull$. The distinction over whether new attributes are preserved or not relies on keeping track of those variables which have been assigned inside the method, and then applying the propagation only for the unset variables.

Example 4. Assume an extra snippet of code in the `Vector` class of the form `if (v2!=null) v1.append(v2) else com`, which is analyzed in state $\sigma = (\{\{v_1\}, \{v_2\}\}, \{v_1/nnull,\ v_2/nnull\}, \{v_1/\{vector\}, v_2/\{vector\}\})$. Since we have nullity information, it is possible to identify the block `com` as dead code. In contrast, sharing-only analyses can only tell if a variable is definitely null, but never if it is definitely non-null. The call is analyzed as follows. Let $A = \{v_1, v_2\}$ and $F = \{this, v\}$, then $\sigma_p = \sigma|_A = \sigma$ and the entry state σ_y is $\sigma|_A^F = (\{\{this\}, \{v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\})$. The only class where `append` can be executed is `Vector` and results (see Example 3) in an exit state for the formal parameters and the return variable $\sigma_b = (\{\{this, v\}\}, \{this/nnull, v/nnull, out/null\}, \{this/\{vector\}, v/\{vector\}, out/\{void\}\})$, which is further renamed to the scope of the caller obtaining $\sigma_\lambda = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$. Since the method returns a `void` type we can treat res as a primitive (null) variable so $\sigma_f = extend(\sigma, \sigma_\lambda, \{v_1, v_2\}) = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$.

Example 5. The *extend* operation used during interprocedural analysis is a point where there can be significant loss of precision and where set sharing shows its strengths. For simplicity, we will describe the example only for the sharing component; nullity and type information updates are trivial. Assume a scenario where a call to `append(v1,v2)` in sharing state $sh = \{\{v_0, v_1\}, \{v_1\}, \{v_2\}\}$ results in $sh_\lambda = \{\{v_1, v_2\}\}$. Let A and AR be the sets $\{v_1, v_2\}$ and $\{v_1, v_2, res\}$ respectively. The *extend* operation proceeds as follows: first we calculate *star* as $(sh_A \cup \{\{res\}\})^* = (sh \cup \{\{res\}\})^* = (\{\{v_0, v_1\}, \{v_1\}, \{v_2\}, \{res\}\})^* = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_1\}, \{v_1, v_2\}, \{v_1, v_2, res\}, \{v_1, res\}, \{v_2\}, \{v_2, res\}, \{res\}\}$, from which we delete those elements whose projection over AR is not included in sh_λ, obtaining $sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$. The resulting sharing component is the union of that sh_{ext} with $sh_{-A} = \emptyset$, so $sh_{f1} = sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$.

When the same sh and sh_λ are represented in their pair sharing versions $sh^p = \{\{v_0, v_1\}, \{v_o, v_0\}, \{v_1, v_1\}, \{v_2, v_2\}\}$ and $sh^p_\lambda = \{\{v_1, v_2\}, \{v_1, v_1\}, \{v_2, v_2\}\}$, the *extend* operation in [26] introduces spurious sharings in sh_f because of the lower precision of the pair-sharing representation. In this case, $sh^p_{f2} = (sh \cup$

$sh^p_\lambda)^*_A = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_0, v_0\}, \{v_1, v_1\}, \{v_2, v_1\}\}$. This information, expressed in terms of set sharing, results in $sh_{f2} = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_1, v_2\}, \{v_1, v_2\}, \{v_0\}, \{v_1\}, \{v_2\}\}$, which is much less precise that sh_{f1}.

4 Experimental Results

In our analyzer the abstract semantics presented in the previous section is evaluated by a highly optimized fixpoint algorithm, based on that of [21]. The algorithm traverses the program dependency graph, dynamically computing the strongly-connected components and keeping detailed dependencies on which parts of the graph need to be recomputed when some abstract value changes during the analysis of iterative code (loops and recursions). This reduces the number of steps and iterations required to reach the fixpoint, which is specially important since the algorithm implements *multivariance*, i.e., it keeps different abstract values at each program point for every calling context, and it computes (a superset of) all the calling contexts that occur in the program. The dependencies kept also allow relating these values along execution paths (this is particularly useful for example during error diagnosis or for program specialization).

We now provide some precision and cost results obtained from the implementation in the framework described in [17] of our set-sharing, nullity, and class ($SSNlTau$) analysis. In order to be able to provide a comparison with the closest previous work, we also implemented the pair sharing (PS) analysis proposed in [26]. We have extended the operations described in [26], enabling them to handle some additional cases required by our benchmark programs such as primitive variables, visibility of methods, etc. Also, to allow direct comparison, we implemented a version of our $SSNlTau$ analysis, which is referred to simply as SS, that tracks set sharing using only declared type information and does not utilize the (non-)nullity component. In order to study the influence of tracking run-time types we have implemented a version of our analysis with set sharing and (non-)nullity, but again using only the static types, which we will refer to as $SSNl$. In these versions without dynamic type inference only declared types can affect τ and thus the dynamic typing information that can be propagated from initializations, assignments, or correspondence between arguments and formal parameters on method calls is not used. Note however that the version that includes tracking of dynamic typing can of course only improve analysis results in the presence of polymorphism in the program: the results should be identical (except perhaps for the analysis time) in the rest of the cases. The polymorphic programs are marked with an asterisk in the tables.

The benchmarks used have been adapted from previous literature on either abstract interpretation for Java or points-to analysis [26,24,23,29]. We added two different versions of the `Vector` example of Fig. 2. Our experimental results are summarized in Tables 5, 6, and 7.

The first column ($\#tp$) in Tables 5 and 6 shows the total number of program points (commands or expressions) for each program. Column $\#rp$ then provides, for each analysis, the total number of *reachable* program points, i.e., the

		PS				SS				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$
dyndisp (*)	71	68	3	114	30	68	3	114	29	-2
clone	41	38	3	42	52	38	3	50	81	55
dfs	102	98	4	103	68	98	4	108	68	0
passau (*)	167	164	3	296	97	164	3	304	120	23
qsort	185	142	43	182	125	142	43	204	165	32
integerqsort	191	148	43	159	110	148	43	197	122	10
pollet01 (*)	154	126	28	276	196	126	28	423	256	30
zipvector (*)	272	269	3	513	388	269	3	712	1029	164
cleanness (*)	314	277	37	360	233	277	37	385	504	116
overall	1497	1330	167	2045	1299	1330	167	2497	2374	82.75

Fig. 5. Analysis times, number of program points, and number of abstract states

		SSNl					SSNlTau				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$
dyndisp (*)	71	61	10	103	53	77	61	10	77	20	-33
clone	41	31	10	34	100	92	31	10	34	90	74
dfs	102	91	11	91	129	89	91	11	91	181	166
passau (*)	167	157	10	288	117	18	157	10	270	114	17
qsort	185	142	43	196	283	125	142	43	196	275	119
integerqsort	191	148	43	202	228	107	148	43	202	356	224
pollet01 (*)	154	119	35	364	388	98	98	56	296	264	35
zipvector (*)	272	269	3	791	530	36	245	27	676	921	136
cleanness (*)	314	276	38	383	276	38	266	48	385	413	77
overall	1497	1294	203	2452	2104	61.97	1239	258	2227	2634	102.77

Fig. 6. Analysis times, number of program points, and number of abstract states

number of program points that the analysis explores, while $\#up$ represents the $(\#tp - \#rp)$ points that are not analyzed because the analysis determines that they are unreachable. It can be observed that tracking (non-)nullity (Nl) reduces the number of reachable program points (and increases conversely the number of unreachable points) because certain parts of the code can be discarded as dead code (and not analyzed) when variables are known to be non-null. Tracking dynamic types (Tau) also reduces the number of reachable points, but, as expected, only for (some of) the programs that are polymorphic. This is due to the fact that the class analysis allows considering fewer implementations of methods, but obviously only in the presence of polymorphism.

Since our framework is multivariant and thus tracks many different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number of abstract states inferred is typically larger than the number of reachable program points. Column $\#\sigma$ provides the total number of these abstract states inferred by the analysis. The level of multivariance is the ratio $\#\sigma/\#rp$. It can be observed that the simple

	PS		SS	
	$\#sh$	$\%sh$	$\#sh$	$\%sh$
dyndisp (*)	640	60.37	435	73.07
clone	174	53.10	151	60.16
dfs	1573	96.46	1109	97.51
passau (*)	5828	94.56	3492	96.74
qsort	1481	67.41	1082	76.34
integerqsort	2413	66.47	1874	75.65
pollet01 (*)	793	89.81	1043	91.81
zipvector (*)	6161	68.71	5064	80.28
cleanness (*)	1300	63.63	1189	70.61
overall	20363	73.39	15439	80.24

Fig. 7. Sharing precision results

set sharing analysis (SS) creates more abstract states for the same number of reachable points. In general, such a larger number for $\#\sigma$ tends to indicate more precise results (as we will see later). On the other hand, the fact that addition of Nl and Tau reduces the number of reachable program points interacts with precision to obtain the final $\#\sigma$ value, so that while there may be an increase in the number of abstract states because of increased precision, on the other hand there may be a decrease because more program points are detected as dead code by the analysis. Thus, the $\#\sigma$ values for $SSNl$ and $SSNlTau$ in some cases actually decrease with respect to those of PS and SS.

The t column in Tables 5 and 6 provides the running times for the different analyses, in milliseconds, on a Pentium M 1.73Ghz, 1Gb of RAM, running Fedora Core 4.0, and averaging several runs after eliminating the best and worst values. The $\%\Delta t$ columns show the percentage variation in the analysis time with respect to the reference pair-sharing (PS) analysis, calculated as $\Delta_{dom}\%t = 100*(t_{dom} - t_{PS})/t_{PS}$. The more complex analyses tend to take longer times, while in any case remaining reasonable. However, sometimes more complex analyses actually take less time, again because the increased precision and the ensuing dead code detection reduces the amount of program that must be analyzed.

Table 7 shows precision results in terms of sharing, concentrating on the SP and SS domains, which allow direct comparison. A more usage-oriented way of measuring precision would be to study the effect of the increased precision in an application that is known to be sensitive to sharing information, such as, for example, program parallelization [4]. On the other hand this also complicates matters in the sense that then many other factors come into play (such as, for example, the level of intrinsic parallelism in the benchmarks and the parallelization algorithms) so that it is then also harder to observe the precision of the analysis itself. Such a client-level comparison is beyond the scope of this paper, and we concentrate here instead on measuring sharing precision directly.

Following [6], and in order to be able to compare precision directly in terms of sharing, column $\#sh$ provides the sum over all abstract states in all reachable program points of the cardinality of the sharing *sets* calculated by the analysis.

For the case of pair sharing, we converted the pairs into their equivalent set representation (as in [6]) for comparison. Since the results are always correct, a smaller number of sharing sets indicates more precision (recall that $\top$ is the power set). This is of course assuming σ is constant, which as we have seen is not the case for all of our analyses. On the other hand, if we compare PS and SS, we see that SS has consistently more abstract states than PS and consistently lower numbers of sharing sets, and the trend is thus clear that it indeed brings in more precision. The only apparent exception is *pollet01* but we can see that the number of sharing sets is similar for a significantly larger number of abstract states.

An arguably better metric for measuring the relative precision of sharing is the ratio $\%_{Max} = 100 * (1 - \#sh/(2^{\#vo} - 1))$ which gives $\#sh$ as a percentage of its maximum possible value, where $\#vo$ is the total number of object variables in all the states. The results are given in column $\%sh$. In this metric 0% means all abstract states are $\top$ (i.e., contain no useful information) and 100% means all variables in all abstract states are detected not to share. Thus, larger values in this column indicate more precision, since analysis has been able to infer smaller sharing sets. This relative measure shows an average improvement of 7% for SS over PS.

5 Conclusions

We have proposed an analysis based on abstract interpretation for deriving precise sharing information for a Java-like language. Our analysis is multivariant, which allows separating different contexts, and combines Set Sharing, Nullity, and Classes: the domain captures which instances definitely do not share or are definitively null, and uses the classes to refine the static information when inheritance is present. We have implemented the analysis, as well as previously proposed analyses based on Pair Sharing, and obtained encouraging results: for all the examples the set sharing domains (even without combining with Nullity or Classes) offer more precision than the pair sharing counterparts while the increase in analysis times appears reasonable. In fact the additional precision (also when combined with nullity and classes) brings in some cases analysis time reductions. This seems to support that our contributions bring more precision at reasonable cost.

Acknowledgments

The authors would like to thank Samir Genaim for many useful comments to previous drafts of this document. Manuel Hermenegildo and Mario Méndez-Lojo are supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. This work was also funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of

Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the *PROMESAS* project.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Alves-Foss, J. (ed.): Formal Syntax and Semantics of Java. LNCS, vol. 1523. Springer, Heidelberg (1999)
3. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proc. of OOPSLA 1996, SIGPLAN Notices, October 1996, vol. 31(10), pp. 324–341 (1996)
4. Bueno, F., García de la Banda, M., Hermenegildo, M.: Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. ACM Transactions on Programming Languages and Systems 21(2), 189–238 (1999)
5. Burke, M.G., et al.: Carini, Jong-Deok Choi, and Michael Hind. In: Pingali, K.K., et al. (eds.) LCPC 1994. LNCS, vol. 892, pp. 234–250. Springer, Heidelberg (1995)
6. Codish, M., et al.: Improving Abstract Interpretations by Combining Domains. ACM Transactions on Programming Languages and Systems 17(1), 28–44 (1995)
7. Cortesi, A., et al.: Complementation in abstract interpretation. ACM Trans. Program. Lang. Syst. 19(1), 7–47 (1997)
8. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL 1977, pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Sixth ACM Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (1979)
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
11. Gosling, J., et al.: Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional, Reading (2005)
12. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI 2006 (2006)
13. Hind, M., et al.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21(4), 848–894 (1999)
14. Jacobs, D., Langen, A.: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In: 1989 North American Conference on Logic Programming, MIT Press, Cambridge (1989)
15. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In: McKinley, K.S. (ed.) Best of PLDI, pp. 473–489. ACM Press, New York (1992)
16. Méndez-Lojo, M., Hermenegildo, M.: Precise Set Sharing for Java-style Programs (and proofs). Technical Report CLIP2/2007.1, Technical University of Madrid (UPM), School of Computer Science, UPM (November 2007)
17. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007) (August 2007)

18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In: ISSTA, pp. 1–11 (2002)
19. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: 1989 North American Conference on Logic Programming, October 1989, pp. 166–189. MIT Press, Cambridge (1989)
20. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: 1991 International Conference on Logic Programming, June 1991, pp. 49–63. MIT Press, Cambridge (1991)
21. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In: 9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007 (July 2007)
22. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: OOPSLA, pp. 146–161 (1991)
23. Pollet, I.: Towards a generic framework for the abstract interpretation of Java. PhD thesis, Catholic University of Louvain, Dept. of Computer Science (2004)
24. Pollet, I., Le Charlier, B., Cortesi, A.: Distinctness and sharing domains for static analysis of java programs. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
25. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999 (1999)
26. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: SAS, pp. 320–335 (2005)
27. Søndergaard, H.: An application of abstract interpretation of logic programs: occur check reduction. In: Duijvestijn, A.J.W., Lockemann, P.C. (eds.) Trends in Information Processing Systems. LNCS, vol. 123, pp. 327–338. Springer, Heidelberg (1981)
28. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI, pp. 387–400 (2006)
29. Streckenbach, M., Snelting, G.: Points-to for java: A general framework and an empirical comparison. Technical report, University Passau (November 2000)
30. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144. ACM Press, New York (2004)
31. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: PLDI, pp. 1–12 (1995)

Towards Execution Time Estimation in Abstract Machine-Based Languages *

E. Mera [1]
[1] Complutense University of Madrid
edison@fdi.ucm.es

P. Lopez-Garcia[2,3]
[2] IMDEA-Software
[3] CSIC
pedro.lopez.garcia@imdea.org

M. Carro[4], M. Hermenegildo[2,4,5]
[4] Technical U. of Madrid
[5] U. of New Mexico
{mcarro, herme}@fi.upm.es

Abstract

Abstract machines provide a certain separation between platform-dependent and platform-independent concerns in compilation. Many of the differences between architectures are encapsulated in the specific abstract machine implementation and the bytecode is left largely architecture independent. Taking advantage of this fact, we present a framework for estimating upper and lower bounds on the execution times of logic programs running on a bytecode-based abstract machine. Our approach includes a one-time, program-independent profiling stage which calculates constants or functions bounding the execution time of each abstract machine instruction. Then, a compile-time cost estimation phase, using the instruction timing information, infers expressions giving platform-dependent upper and lower bounds on actual execution time as functions of input data sizes for each program. Working at the abstract machine level makes it possible to take into account low-level issues in new architectures and platforms by just reexecuting the calibration stage instead of having to tailor the analysis for each architecture and platform. Applications of such predicted execution times include debugging/verification of time properties, certification of time properties in mobile code, granularity control in parallel/distributed computing, and resource-oriented specialization.

Categories and Subject Descriptors D.4.8 [*Performance*]: Modeling and prediction;
F.3.2 [*Semantics of Programming Languages*]: Program analysis;
D.1.6 [*Programming Techniques*]: Logic programming

General Terms Languages, performance

Keywords Execution Time Estimation, Cost Analysis, Profiling, Resource Awareness, Cost Models, Logic Programming.

* The authors have been partially supported by EU projects 215483 *S-Cube*, IST-15905 *MOBIUS*, Spanish projects ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, ITEA/PROFIT FIT-350400-2006-44 *GGCC*, MEC TIN2005-09207-C03-01 *MERIT/COMVERS*, Comunidad de Madrid project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also partially funded by the Prince of Asturias Chair in Information Science and Technology.

PPDP'08, July 15–17, 2008, Valencia, Spain.

1. Introduction

Cost analysis has been studied for several declarative languages (7; 16; 11; 13). In logic programming previous work has focused on inferring upper (12; 11) or lower (13; 8) bounds on the cost of programs, where such bounds are *functions on the size (or values) of input data*. This approach captures well the fact that program execution cost in general depends on input data sizes. On the other hand the results of these analyses are given in terms of *execution steps*. While this measure has the advantage of being platform independent, it is not straightforward to translate such steps into execution time.

Estimation of *worst case execution times* (WCET) has received significant attention in the context of high-level imperative programming languages (24). In (18; 6) a portable WCET analysis for Java is proposed. However, the WCET approach only provides absolute upper bounds on execution time (i.e., bounds that do not depend on program input arguments) and often requires annotating loops manually.

Our objective is to infer automatically more precise bounds on execution times that are in general functions that depend on input data sizes. In (19) a static analysis was proposed in order to infer such platform-dependent time bounds in logic programs. This approach is based on a high-level analysis of certain syntactic characteristics of the program clause text (sizes of terms in heads, sizes of terms in bodies, number of arguments, etc.). Although promising experimental results were obtained, the predicted execution times were not very precise. In this paper we propose a new analysis which, in order to improve the accuracy of the time predictions, on one hand takes into account lower level factors and on the other makes the model richer by directly taking into account the inherently variable cost of certain low-level operations.

Regarding the choice of this lower level, rather than trying for example to model directly the characteristics of the physical processor, as in WCET, and given that most popular logic programming implementations are based on variations of the Warren abstract machine (WAM) (23; 1), we chose to model cost at the level of abstract machine instructions. Abstract machines have been used as a basic implementation technique in several programming paradigms (functional, logic, imperative, and object-oriented) (14) with the advantage that they provide an intermediate layer that separates to a certain extent the many low-level details of real (hardware) machines from the higher-level language, while at the same time making compilation easier. This property can be used to facilitate the design of our framework.

Within this setting, we present a new framework for the static estimation of execution times of programs. The basic ideas in our approach follow:

1. Measure the execution time of each of the instructions in a lower-level L_B (bytecode) language (or approximate it with a function if it depends on the value of an argument) in some specific abstract machine implementation when executed on a given processor / O.S.
2. Make the information regarding instruction execution time available to the timing analyzer. This is, in our proposal, done by means of *cost assertions* (written in a suitable assertion language) which are stored in a module accessible to the compiler/analyzer.
3. Given a concrete program P written in the source language L_H, compile it into L_B and record the relationship between P and its compiled counterpart.
4. Automatically analyze program P, taking into account the instruction execution time (determined in item 1 above) to infer a cost function C_P. This function is an expression which returns (bounds on) the actual execution time of P for different input data sizes for the given platform.

Points (1) and (2) are performed in a one-time profiling phase, independent from program P, while the rest are performed once for each P in the static (compile-time) cost analysis phase. We would like to point out that, in general, the basic ideas underlying our work can be applied to any language L_H as long as (i) cost estimation can be derived for programs written in L_H, (ii) the translation of L_H to some other (usually lower-level) language L_B is accessible, and (iii) the execution time of the instructions in L_B can be timed accurately enough. We will, however, focus herein on logic languages, so that we assume L_H to be a Prolog-like language and L_B some variant of the WAM bytecode.

The proposed framework has been implemented as part of the CiaoPP (17) system in such a way that any abstract machine properly instrumented can be analyzed. To the best of our knowledge, this is the first attempt at providing a timing analysis producing upper- and lower-bound time *functions* based on the cost of lower-level machine instructions.

2. Mappings Between Program Segments and Bytecodes

Let $OpSet = \{b_1, b_2, \ldots, b_n\}$ be the set of instructions of the abstract machine under consideration. We assume that each instruction is defined by a numeric identifier and its arity, i.e., $b_i \equiv f_i/n_i$, where f_i is the identifier and n_i the arity. Each program is compiled into a sequence of expressions of the form $f(a_1, a_2, \ldots, a_n)$ where f is the instruction name and the a_i's are its arguments. For conciseness, we will use I_i to refer to such expressions. The sequences of expressions into which a program is compiled are generally encoded using bytecodes. In the following we will often refer to sequences of abstract machine instructions or sequences of bytecodes simply as "bytecodes."

Let C be a clause H :- $L_1, \ldots, L_m$. Let $E(C)$ be a function that returns the sequence of bytecodes resulting from the compilation of clause C:

$$E(C) =< I_1, I_2, \ldots, I_p >$$

Let $E(C, H)$ be a function that maps the clause head H to the sequence of bytecodes in $E(C)$ starting from the beginning up to the first `call/execute` instruction or to the end of the sequence $E(C)$ if there are no more `call/execute` instructions (i.e., to the end of the bytecode sequence resulting from the compilation of clause C). Let $E(C, L_i)$ be the function that maps literal L_i of clause C to the sequence of bytecodes in $E(C)$ which start at the `call` bytecode instruction corresponding to this literal and up to the next `call/execute` instruction or to the end of the sequence $E(C)$ if there are no more `call/execute` instructions. If $\uplus$ represents the concatenation of sequences of bytecodes, then:

$$E(C) = E(C, H) \biguplus (\biguplus_{i=1}^{m} E(C, L_i))$$

Note that functions $E(C, H)$ and $E(C, L_i)$ do not necessarily return the bytecodes that one would normally associate to the clause head H and literal L_i respectively. Instead, the definition of those functions associates the instructions corresponding to argument preparation for a given call with the (success of the) *previous* call (or head). This is to cater for the fact that, in the context of backtracking, the WAM argument preparation occurs only one time per call to a literal, even if such call is retried more times before failing definitively. As a result, the cost of argument preparation for a given `call/execute` instruction needs to be associated with the previous literal to that `call/execute`, in order not to count it every time the call is retried.

Table 1 shows how predicate append/3 is compiled into bytecodes, and identifies the result of calling the $E(C, H)$ and $E(C, L_i)$ functions for each clause head and body literal. H^1 represents the head of the first clause (C_1), and H^2 and L_1^2 the head of the second (recursive) clause (C_2) and the first literal in such clause body (the only body literal).

$E(C_1, H^1)$	`append([], X, X).`	
	append/3/1:	try_me_else append/3/2 allocate get_constant([],A0) get_variable(V0,A1) get_value(V0,A2) deallocate proceed
$E(C_2, H^2)$	`append([X|Xs], Y, [X|Zs]) :-`	
	append/3/2:	trust_me allocate get_variable(V0,A0) set_variable(V1) set_variable(V2) set_variable(V3) get_list(V1,V3) set_variable(V4) unify_variable(V2,V4) unify_variable(V0,V3) set_variable(V5,A1) get_variable(V6,A2) set_variable(V7) set_variable(V8) get_list(V1,V8) set_variable(V9) unify_variable(V7,V9) unify_variable(V6,V8) put_value(V2,A0) put_value(V5,A1) put_value(V7,A2) deallocate
$E(C_2, L_1^2)$	`append(Xs, Y, Zs).`	
		execute append/3

Table 1. Sequences of bytecodes assigned to clause heads and body literals of the clauses C_1 and C_2 of predicate append by the functions $E(C, H)$ and $E(C, L)$.

3. Modeling the Execution Time of Instructions

We define a function $t(I)$ (the *timing model*), which takes a bytecode instruction I and returns another function which estimates the execution time for it depending on the input data sizes of the bytecode. This is similar to the approach described in (5), where, however, $t(I)$ was a constant.

In many cases we can assume that the time to execute a bytecode is constant. However there are some instructions for which this does not hold because their definitions involve loops. In many of these cases the timing model consists of an initial constant time t_0 plus another additional constant time t_{iter} to cater for the cost of each iteration, and a simple linear model can be used: $t_0 + n \times t_{iter}$. Consider for example the `unify_void` n instruction, which pushes n new unbound cells on the heap (1), and whose execution time is a linear function on n. In some other cases instructions have different execution times depending on the (fixed) values a given argument can take from some finite set. In such cases, execution time is an arbitrary function on the argument. Specific constants are assigned for each possible argument value by means of profiling (Section 5).

Since the cost of a given instruction is different when it succeeds and when it fails, we will have two costs for each instruction that can fail: one for the success case and another for the failure case. Finally, and besides lower-level factors such as cache behavior, there are some additional variable factors (such as, e.g., the length of dereferencing chains) which may affect execution times. These factors are in principle not impossible to cater for via a combination of static and dynamic analysis, but, given the additional complication involved, we will ignore them herein and explore what kind of precision of timing prediction can be achieved with this first level of approximation.

Another factor that we are not taking into account at this moment is garbage collection (GC). GC makes programs run slower, which, at profiling time, increases the (estimated) cost of every instruction. Therefore, turning it off at profile time (which gives a smaller estimation of instruction cost) is safe when finding out lower bounds: if the program whose execution time is to be predicted is run with GC turned on, then it would run slower w.r.t. an execution with GC turned off (as it was when profiling), and the estimated bounds will still be lower bounds, albeit more conservative. An inverse reasoning applies to upper bounds, and the technique herein presented is equally valid. However, for the sake of simplicity, we have taken all the measurements (both for profiling and executions to be predicted) with GC disconnected.

4. Static Cost Analysis

We now present the compile-time component of our combined framework: the static cost analysis. This analysis has been implemented and integrated in `CiaoPP` (17).

4.1 Overview of the Approach

Since the work done by a call to a recursive procedure often depends on the "size" of its input, knowing this size is a prerequisite to statically estimate such work. Our basic approach is as follows: given a call p, an expression $\Phi_p(n)$ is *statically* computed that (i) is relatively simple to evaluate, and (ii) it approximates $\mathtt{Time}_p(n)$, where $\mathtt{Time}_p(n)$ denotes the cost (in time units) of computing p for an input of size n on a given platform. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. It is then evaluated at run-time, when the size of the input is known, yielding (upper or lower) bounds on the execution time required by the computation of the call on a given platform. In the following we will refer to the compile-time computed expressions $\Phi_p(n)$ as *cost functions*.

Certain program information (such as, for example, input/output modes and size metrics for predicate arguments) is first automatically inferred by other analyzers which are part of `CiaoPP` and then provided as input to the size and cost analysis. The techniques involved in inferring this information are beyond the scope of this paper —see, e.g., (17) and its references for some examples. Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the predicate being analyzed, relative to the sizes of the input arguments to this predicate, using the inferred metrics. The size of an output argument in a predicate call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. To this end, and using the input-output argument information, data dependency graphs (namely the *argument dependency graph* and the *literal dependency graph*) are used to set up difference equations whose solution yields size relationships between input and output arguments of predicate calls. The *argument dependency graph* is a directed acyclic graph used to represent the data dependency between argument positions in a clause body (and between them and those in the clause head). The *literal dependency graph* is constructed from the *argument dependency graph* (grouping nodes) and represents the data dependencies between literals.

The information regarding argument sizes is then used to set up another set of difference equations whose solution provides bound functions on predicate calls (execution time). Both the size and cost difference equations must be solved by a difference equation solver. Although the operation of such solvers is beyond the scope of the paper, our implementation does provide a table-based solver which covers a reasonable set of difference equations such as first-order and higher-order linear difference equations in one variable with constant and polynomial coefficients,[1] divide and conquer difference equations, etc. In addition, the system allows the use of external solvers (such as, e.g., Purrs (4), Mathematica, Matlab, etc.) and is currently being extended to interface with other interesting solvers that have been recently developed (2). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of difference equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

4.2 Estimating the Execution Time of Clauses and Predicates

Our cost analysis approach is based on that developed in (12; 11) (for estimation of upper bounds on resolution steps) and further extended in (13) (for lower bounds). More recently, in (19) the analysis was extended to work with *vectors* of cost components, with each component considering a known aspect that affects the total cost of the program. In these approaches the cost of a clause can be bounded by the cost of head unification together with the cost of each of its body literals. For simplicity, the discussion that follows is focused on the estimation of upper bounds. We refer the reader to (13) for details on lower-bounds cost analysis.

Consider a predicate defined by r clauses $\mathtt{C}_1, \ldots, \mathtt{C}_r$. We take into account that a given clause $\mathtt{C}_k$ will be tried only if clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ fail to yield a solution. Consider clause $\mathtt{C}_k$ defined as $\mathtt{H}^k$:- $\mathtt{L}^k_1, \ldots, \mathtt{L}^k_m$. Because of backtracking, the number of times a literal will be executed depends on the number of solutions of the previous literals. Assume that $\overline{n}$ is a vector such that each element corresponds to the size of an input argument to clause $\mathtt{C}_k$ and that each $\overline{n}_i, i = 1 \ldots m$, is a vector such that each element corresponds to the size of an input argument to literal $\mathtt{L}^k_i$. Assume also that $\tau(\mathtt{H}^k, \overline{\mathtt{n}})$ is the execution time needed to resolve the head $\mathtt{H}^k$ of

[1] Note that it is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable.

the clause $\mathtt{C}_k$ with the literal being solved, $\mathtt{Sols}_{\mathtt{L}_j^k}$ is the number of solutions literal $\mathtt{L}_j^k$ can generate, and $\beta(\mathtt{L}_i^k, \overline{n}_i)$ the time needed to prepare the call to literal $\mathtt{L}_i^k$ in the body of the clause $\mathtt{C}_k$. Because of space constraints, we refer the reader to (11; 13) for details about the algorithms used to estimate the number of solutions that a literal can generate, and the sizes of input arguments. Then, an upper bound $\mathtt{Cost}_{\mathtt{C}^k}(\overline{n})$ on the cost of clause $\mathtt{C}^k$ (assuming all solutions are required) can be expressed as:

$$\mathtt{Cost}_{\mathtt{C}^k}(\overline{n}) \leq \quad \tau(\mathtt{H}^k, \overline{n}) + \sum_{i=1}^{m} (\prod_{j \prec i} \mathtt{Sols}_{\mathtt{L}_j^k}(\overline{n}_j))(\beta(\mathtt{L}_i^k, \overline{n}_i) + \mathtt{Cost}_{\mathtt{L}_i^k}(\overline{n}_i))$$

Here we use $j \prec i$ to denote that $\mathtt{L}_j^k$ precedes $\mathtt{L}_i^k$ in the literal dependency graph for the clause $\mathtt{C}_k$ (described in Section 4.1). We have that:

$$\tau(\mathtt{H}^k, \overline{n}) = \delta_k(\overline{n}) + \sum_{I \in E(\mathtt{C}^k, \mathtt{H}^k)} t(I)(\overline{n})$$

where $\delta_k(\overline{n})$ denotes the execution time necessary to determine that clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ will not yield a solution and that $\mathtt{C}_k$ must be tried: the function δ_k obviously takes into account the type and cost of the indexing scheme being used in the underlying implementation. Also:

$$\beta(\mathtt{L}_i^k, \overline{n}_i) = \sum_{I \in E(\mathtt{C}, \mathtt{L}_i^k)} t(I)(\overline{n}_i), i = 1, \cdots, m$$

with $E(\mathtt{C}, \mathtt{L}_i^k)$ and $t(I)$ defined as in Sections 2 and 3 respectively.

A difference equation is set up for each recursive clause, whose solution (using as boundary conditions the execution times of non-recursive clauses) is a function that yields the execution time of a clause. The execution time of a predicate is then computed from the execution time of its defining clauses. Since the number of solutions which will be required from a predicate is generally not known in advance, a conservative upper bound on the execution time of a predicate can be obtained by assuming that all solutions are needed, and, thus, all clauses are executed and the execution time of the predicate will be the sum of the execution times of its defining clauses. When the clauses of a predicate are mutually exclusive, a more precise estimation of the execution time of such a deterministic predicate can be obtained as the maximum of the execution times of the clauses it is composed of.

Note that our approach allows defining via assertions the execution time of external predicates, which can then be used for modular composition. This includes also predicates for which the code is not available or which are even written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the execution time of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. The description of the assertion language used is out of the scope of this paper, and we refer the reader to (21) for details.

5. Estimating Instruction Execution Times via Profiling

In this section we will see how data regarding the expected execution time of each instruction in the abstract machine (Section 3) can be accurately measured in a realistic environment.

5.1 Instruction Profiling

Profiling aims at calculating a function $t(I)$ for each bytecode instruction I. An approach is to instrument the WAM implementation so that time measures are taken and recorded at appropriate points in the execution (18). In practice, a number of issues have to be taken into account in order to obtain accurate enough measurements. These include the selection of the places where the instrumentation code will be inserted, how to minimize the effects of such instrumentation on the execution (not only execution time but also, e.g., cache behavior), and how to work around the complex instruction scheduling performed by modern processors, which may lead to large variance in the results, especially since we aim at measuring very small fragments of code.

```
while (op != END) { /* WAM emulation loop */
  ...
   record_profile_info (op); /* op is the current bytecode */

  switch(op) {
    ...
  }
  ...
  op = get_next_op();
}
```

Figure 1. A simple WAM emulation loop instrumented.

A first approximation is to add profiling-related calls in designated parts of the bytecode interpreter main loop. Figure 1 shows a piece of code illustrating this. The `record_profile_info(op)` operation records the start time for the bytecode `op`. The end time is processed when the next opcode is fetched. The data for each bytecode is maintained in memory during execution (and in raw form in order to impact execution as little as possible) and later saved to an external file.

A benchmark-based analysis is also proposed in (18), which describes how the instrumented code can be reused effectively on various platforms without modifying it, and how the execution time of a specific set of bytecodes can be measured.

However, the methods mentioned above have drawbacks. For example, the first one (instrumenting the main loop) depends on the existence of very precise, non-intrusive, low-overhead timing operations which, unfortunately, are not always available in all platforms. Portable O.S. calls, besides having a typically high associated overhead, are in general not accurate enough for our purposes. Even if a very fast timing operation is available (which is not the case in platforms such as mobile and embedded devices), its introduction may affect the behavior of the machine being analyzed if the abstract machine loop is very optimized. For example, if the new code changes register and variable allocation, program behavior will be affected in unforeseen ways.

We will, however, use an instrumented loop like that of Figure 1 to count the number of bytecodes executed in a calibration step.

5.2 Measuring Time Accurately

In order to do portable time measurements in platforms where high resolution timing is difficult or impossible to achieve, workarounds have to be used. The approach that we have followed is based on using synthetic benchmarks which on purpose repeatedly execute the instructions under estimation for a large enough time, and later divide the total execution time by the number of times the instructions were executed. A complication in this process is that it is in general not possible to run a single instruction repeatedly within the abstract machine, since the resulting sequence would not be legal and may "break" the abstract machine, run out of memory, etc. In general, more complex sequences of instructions must be constructed and repeated instead.

Therefore, the approach we have followed involves designing a set of *legal* programs which cover all the bytecode instructions,

Programs	Instructions	Trace
c1_5 :- c1_5_0.	00 : execute 01	00 : execute 01
c1_5_0 :- c1_5_1.	01 : execute 02	01 : execute 02
c1_5_1 :- c1_5_2.	02 : execute 03	02 : execute 03
c1_5_2 :- c1_5_3.	03 : execute 04	03 : execute 04
c1_5_3 :- c1_5_4.	04 : execute 05	04 : execute 05
c1_5_4 :- c1_5_5.	05 : execute 06	05 : execute 06
c1_5_5.	06 : proceed	06 : proceed
c1_0 :- c1_0_0.	01 : execute 02	01 : execute 02
c1_0_0.	02 : proceed	02 : proceed

Table 2. Programs used in order to get the execution time of the `execute` instruction.

relate the execution time of these programs with the individual instruction execution times with a system of equations, and solving such a system.

5.3 Getting Instruction Execution Time

We now discuss how to set up calibration programs in order to get the cost of bytecodes. In this section, and in order to simplify the discussion, we deal with those bytecodes whose execution time is bound by a constant. In the following section we extend our technique to manage instructions whose execution time is unbound.

Let $C_i, i = 0, 1, \ldots, n$ be a set of synthetic calibration programs, each of them returning the execution time of a block of code. Each C_i, which we will refer to as calibrator, is generated in such a way that it repeats such block a given number of times, say r. Let us assume, for example, that we want to calibrate the WAM instruction "`execute`" when it does not fail and that we want to repeat its execution 5 times (i.e., $r = 5$). Table 2 shows a set of programs which can be used to calibrate this WAM instruction. Columns **Instructions** and **Trace** show the WAM code as generated by the compiler and the sequence of instructions executed when running the program starting from the first clause respectively. In general, in our approach, rather than a concrete program, calibrators are program generation templates which take r as an input and return, e.g., the programs in Table 2 for that value of r. The actual calibration program includes an entry point which calls the programs in Table 2 and returns the value of the execution time of the `execute` instruction, subtracting the time spent in the entry calls (e.g., `c1_5` for Table 2). In this case the calibration time is easy to compute as the difference between the execution time of `c1_5` and `c1_0` divided by r. The result of the calibration should ideally be invariant with respect to r; in practice this is however not true due, among other factors, to timing imprecision. Thus, r needs to be determined for each case: it has to be a large enough value to ensure stability of the time measured by the calibrator for the particular platform and the method used to measure time, but not excessively large, as this would make calibration impractical.

In some cases we cannot isolate the behavior of only one bytecode. This is specially the case in the calibrators of instructions which alter the program flow, such as `call`, `proceed`, `trust_me`, `try_me_else`, `retry_me_else`, `allocate`, `deallocate`. It is also the case when measuring the cost of failure for any of the instructions which can fail (generally the `get_` and `unify_` instructions). All these instructions need to be always executed together with other bytecodes in order to make the calibration program legal. As a result, and due to interactions between the costs of the different instructions, the equations are not as easy to configure in all cases as the simple case for the `execute` instruction above.

As a simple example, the calibrator that returns the cost of `call` and the `proceed` instructions uses the programs in Table 3 (where we have turned off the optimization of register / variable allocation in the compiler for simplicity). In order to be able to separate the

Programs	Instructions	Trace
c2_5 :-	00 : allocate	00 : allocate
c_5,	01 : call 09	01 : call 09
c_5,	02 : call 09	09 : proceed
c_5,	03 : call 09	02 : call 09
c_5,	04 : call 09	09 : proceed
c_5,	05 : call 09	03 : call 09
c_5,	06 : call 09	09 : proceed
c_5.	07 : deallocate	04 : call 09
	08 : execute 09	09 : proceed
		05 : call 09
c_5.	09 : proceed	09 : proceed
		06 : call 09
		09 : proceed
		07 : deallocate
		08 : execute 09
		09 : proceed
c2_0 :-	00 : allocate	00 : allocate
c_0,	01 : call 04	01 : call 04
c_0.	02 : deallocate	04 : proceed
	03 : execute 04	02 : deallocate
		03 : execute 04
c_0.	04 : proceed	04 : proceed

Table 3. Programs used to get the execution time of the `call` and `proceed` instructions.

cost of `call` and `proceed` an idea might be to find a calibrator that isolates the cost of `proceed` by itself and subtract from the value given by the calibrator for `call` and `proceed` and obtain the cost of `call`. However, that is in general not possible since in all legal calibrators `proceed` and `call` must always appear combined with other bytecodes. In general we need to set up a system of equations in which the known values are the costs given by our calibrators and the unknown values are the costs of the individual bytecodes. Such equations can be configured automatically, by executing the calibration programs in a special version of the WAM with the bytecode dispatch loop instrumented as in Figure 1 so that the profiler keeps an account of the executed bytecodes.

Let c_i, $0 \leq i \leq n$, be the time calibrator C_i has returned, and let β_j, $0 \leq j \leq m$, $m \geq n$, be the cost of a bytecode B_j, distinguishing between the case of a fail or a success in the execution of such bytecode. In other words, $B_j \in I \times \{fail, success\}$, where I the set of all possible bytecodes and *fail* and *success* represent the failure or success of the execution of a bytecode. We can then set up the following system of equations:

$$\begin{array}{rcl} c_1 & = & a_{11}\beta_1 + a_{12}\beta_2 + \cdots + a_{1m}\beta_m \\ c_2 & = & a_{21}\beta_1 + a_{22}\beta_2 + \cdots + a_{2m}\beta_m \\ & \cdots & \\ c_n & = & a_{n1}\beta_1 + a_{n2}\beta_2 + \cdots + a_{nm}\beta_m \end{array} \quad (1)$$

which we can rewrite such using matrix notation:

$$C = AB \quad (2)$$

where $B = (\beta_i)$ is the vector of execution times for the bytecodes. In order to obtain B we ideally need to configure as many calibrators as bytecodes. Finding a solution to this system of equations requires, in principle, independence among the equations (i.e., there is no other linear independent equation but those in (1)), and to have as many equations as variables. However, that is not always possible due to dependencies between the number of times a bytecode is executed. For example, in the WAM under analysis, the following invariant holds:

PROPOSITION 1. *For any program, the number of times* `retry_me_else` *is called plus the number of times* `trust_me` *is called is equal to the number of failures.*

This holds since a failure always causes backtracking to the next choice point, which always implies executing either a `retry_me_else` or a `trust_me` instruction. As the coefficients a_{ij} in the equation above are precisely the number of times every bytecode is executed, it turns out that, for a given execution, some coefficients are dependent on some other coefficients, therefore breaking the initial independence assumption: the system of equations is underdetermined and it does not have a unique solution.

For this reason, since the coefficients a_{ij} where obtained by summarizing legal programs (i.e., the calibrators), and they will be affected by the linear dependency mentioned above, the undetermined system (2) will not have a unique solution. However, note that when several bytecodes in a block must be executed together (because of constraints in the WAM compilation and execution scheme) knowing the execution time of each of them in isolation is not needed: knowing the total execution time of the whole block is enough. This intuitive idea can be formalized and generalized with the following result:

PROPOSITION 2. *Given a set of n calibration programs C_i, that define n linear independent equations with β_i variables (corresponding to the m bytecodes, with both success and failure cases included), if we have that for all programs there exist $m-n$ linear independent relationships between the number of bytecodes that are always fulfilled, then the estimated execution time is invariant with respect to the choice of any arbitrary element of the solution set of such linear system.*

Proof : Let B be an arbitrary solution of $C = AB$. Let X be a vector which represents the number of times each bytecode has been executed for a given program. The estimated execution time is $E = X^T B$, i.e., the sum of the time for each bytecode multiplied by the number of times it has been executed.

By linear algebra, and considering that each calibrator defines a linear independent equation, we have that the range of A is n, and the kernel (or nullspace) of A is given by the set of all λ such that $A\lambda = 0$, a vector space of dimension $m - n$ (0 represents the null vector of dimension n). In other words, we have that:

$$C = AB = AB + 0 = AB + A\lambda = A(B + \lambda) \tag{3}$$

Then, $B+\lambda$ is a solution of (2), and it is also a representative of the solution set of such equation system. What we should prove now is that $X^T(B + \lambda) = X^T B$, that is, canceling common terms and transposing the equations:

$$\lambda^T X = 0 \tag{4}$$

On the other hand, we have a set of $m-n = k$ linear dependencies between the number of bytecodes executed of the form:

$$\begin{array}{rcl} 0 & = & v_{11}x_1 + v_{12}x_2 + \cdots + v_{1m}x_m \\ 0 & = & v_{21}x_1 + v_{22}x_2 + \cdots + v_{2m}x_m \\ & \cdots & \\ 0 & = & v_{k1}x_1 + v_{k2}x_2 + \cdots + v_{km}x_m \end{array}$$

Or, rewriting them using matrices:

$$0 = VX \tag{5}$$

The result of multiplying an arbitrary vector d by V is a vector $\mu^T = (dV)$ and for the equation above, it follows that $\mu^T X = 0$.

But note that the rows of A were obtained executing a program that meets the linear dependencies too, that is, $\mu^T A^T = 0$. Transposing, we have:

$$A\mu = 0 \tag{6}$$

For this reason, we can see that as λ, μ is a member of the kernel of A, and considering the uniqueness of the kernel of a matrix, and that μ is an element of a space of dimension $m - n$, we can choose μ such that $\lambda = \mu$, that is, we can express λ as the product $(dV)^T$, as result of basic theorems of linear algebra. Therefore, we have that:

$$\lambda^T X = \mu^T X = (dV)X = d(VX) = d(0) = 0 \tag{7}$$

□

Then, the method we follow to select a representative solution B is simply to complete the equation systems with $m - n$ arbitrary equations in order to make them become determined. Such equations should be selected in such a way that the β_i values be positive, for example, by setting the cost to 0 as the time of the bytecodes that are faster, avoiding negative solutions.

5.4 Dealing with unbound instructions

We now consider the case of bytecode instructions whose execution time depends on the specific values that certain parameters can take at run time. In such cases the accuracy of our analysis can be increased by taking advantage of static term-size analysis and the addition of cost-related assertions for such instructions. Such assertions make it possible to introduce ad-hoc functions giving the size of the input parameters of the bytecode.

In fact, our system is able to deal with several metrics (e.g., value, length, size, depth, ...) as shown in (12; 11; 13), but for brevity, in the following paragraphs we will describe an example unifying lists.

Let us take, the instruction `unify_variable(V, W)` and let us assume that we want to calculate an upper bound for its execution time upon success and for the case where the two arguments to unify are lists of numbers. We assume that an upper bound to the execution time is proportional to the number of iterations necessary to scan the lists. The timing model for such instruction is thus $K_1 + K_2 * length(V)$, because if the instruction succeeds, the length of both V and W should be equal. The value of constants K_1 and K_2 is calculated by setting up two benchmarks which unify lists of different length l_1 and l_2. If the cost of `unify_variable` for these two list lengths is, respectively, B_1 and B_2, then we set up the following system of linear equations:

$$\begin{array}{rcl} B_1 & = & K_1 + K_2 \times l_1 \\ B_2 & = & K_1 + K_2 \times l_2 \end{array} \tag{8}$$

Note that B_1 and B_2 can be added to the system of equations (2) to get its values in one step, and later, we solve K_1 and K_2 in the system of linear equations (6).

6. Experimental results

In order to evaluate the techniques presented so far we need to choose a concrete bytecode language and an implementation of its abstract machine to execute and profile with. As mentioned before, the de-facto target abstract machine for most Prolog compilers is the WAM (23; 1) or one of its derivatives. In order to evaluate the feasibility of the approach we have chosen a relatively simple WAM design, which is quite close to the original WAM definition. It is based on (9), but has been ported from Java to C/C++ to achieve similar performance of other Prolog systems. The use of a relatively simple abstract machine allows evaluating the technique while avoiding the many practical complications present in modern implementations, such as having complex instructions resulting from merging other, simpler ones, or specializations of instruction and argument combinations. This of course does not preclude the application of our technique to the more complex cases.

In our concrete abstract machine, we have considered 42 equations for 43 bytecodes, differentiating the success and failure cases.

As we have seen in Proposition **??**, there exists a linear relationship between the number of bytecodes that a program will call which can be stated as:

$$\begin{aligned} 0 = \quad & x_{30} + x_{38} - x_{13} - x_{15} - x_{17} - x_{22} - x_{41} \\ & - x_{43} - x_{49} - x_{50} - x_{51} - x_{52} - x_{53} \end{aligned}$$

where the x_i represent the number of times the bytecode tagged as β_i has been executed for any program being analyzed (see Tables 4 and 6).

By Proposition 1, we are free to select any arbitrary solution of the linear system. The proposed solution has been found by setting arbitrarily the cost of fail to zero. Then, our set of linear equations, discarding those whose calibrators are composed only with one bytecode, is as follows:

$$\begin{array}{ll|ll} 0 & = \beta_{13} & c_{01} & = \beta_{01} + \beta_{07} \\ c_{20} & = \beta_{20} + \beta_{33} + \beta_{43} & c_{09} & = \beta_{09} + \beta_{24} \\ c_{11} & = \beta_{01} + \beta_{11} + 2\beta_{28} + \beta_{30} & c_{15} & = \beta_{15} + \beta_{38} \\ c_{46} & = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{50} & c_{17} & = \beta_{17} + \beta_{30} \\ c_{42} & = \beta_{01} + 2\beta_{27} + \beta_{30} + \beta_{52} & c_{07} & = \beta_{07} + \beta_{24} \\ c_{22} & = \beta_{01} + \beta_{22} + \beta_{23} + \beta_{30} & c_{29} & = \beta_{01} + \beta_{17} + \beta_{30} \\ c_{34} & = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{35} & c_{37} & = \beta_{17} + \beta_{38} \\ c_{36} & = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{37} & c_{38} & = \beta_{07} + \beta_{24} + \beta_{39} \\ c_{40} & = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{41} & c_{19} & = \beta_{19} + \beta_{33} \\ c_{43} & = \beta_{01} + \beta_{27} + \beta_{28} & c_{13} & = \beta_{01} + \beta_{13} + \beta_{30} \\ & \quad + \beta_{30} + \beta_{49} & & \\ c_{49} & = \beta_{01} + 2\beta_{19} + 2\beta_{27} & c_{51} & = \beta_{01} + 2\beta_{20} \\ & \quad + \beta_{30} + 2\beta_{31} + 2\beta_{33} + \beta_{51} & & \quad + \beta_{30} + 2\beta_{31} + \beta_{53} \end{array} \tag{9}$$

Solving this linear system we get:

$$\begin{array}{rcl} \beta_{01} & = & c_{29} - c_{17} \\ \beta_{07} & = & -c_{29} + c_{17} + c_{01} \\ \beta_{09} & = & -c_{29} + c_{17} + c_{09} - c_{07} + c_{01} \\ \beta_{11} & = & -2c_{27} - c_{13} + c_{11} \\ \beta_{13} & = & 0 \\ \beta_{15} & = & -c_{37} + c_{29} + c_{15} - c_{13} \\ \beta_{17} & = & c_{29} - c_{13} \\ \beta_{19} & = & c_{19} - c_{32} \\ \beta_{20} & = & -c_{44} - c_{32} + c_{20} \\ \beta_{22} & = & -c_{23} + c_{22} - c_{13} \\ \beta_{24} & = & c_{29} - c_{17} + c_{07} - c_{01} \\ \beta_{30} & = & -c_{29} + c_{17} + c_{13} \\ \beta_{35} & = & c_{34} - c_{23} - c_{13} \\ \beta_{37} & = & c_{36} - 2c_{27} - c_{13} \\ \beta_{38} & = & c_{37} - c_{29} + c_{13} \\ \beta_{39} & = & c_{38} - c_{07} \\ \beta_{41} & = & c_{40} - c_{23} - c_{13} \\ \beta_{49} & = & c_{43} - c_{27} - c_{26} - c_{13} \\ \beta_{50} & = & c_{46} - 2c_{27} - c_{13} \\ \beta_{51} & = & c_{49} - 2c_{30} - 2c_{26} - 2c_{19} - c_{13} \\ \beta_{52} & = & c_{42} - 2c_{26} - c_{13} \\ \beta_{53} & = & c_{51} + 2c_{44} + 2c_{32} - 2c_{30} - 2c_{20} - c_{13} \end{array} \tag{10}$$

The leftmost column of Tables 4 and 6 summarizes the calibration data for the instructions of our WAM implementation. For brevity, we actually only show those being used in the examples tested, although we have calibrated all of them. In the second column there is a tag that is the variable name in the linear equations system. In the examples we deal with a subset of Prolog which only has operations on integers, atoms, lists, and terms. Likewise, we obviate issues like modules or syntactic sugar which can be dealt with at the Prolog level. A few additional built-in predicates are required to have a minimal functionality including `write/1`, `consult/1`, etc. They are profiled separately and their timing is given to the system through assertions. This is also a valid solution in order to be able to analyze larger programs.

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
allocate	β_{01}	29	366	1055
arith_add	β_{02}	29	489	1438
arith_div	β_{03}	29	580	1541
arith_mod	β_{04}	29	641	1553
arith_mul	β_{05}	28	519	1468
arith_sub	β_{06}	28	519	1438
call	β_{07}	11	183	261
cut	β_{08}	13	183	581
deallocate	β_{09}	7	305	142
execute	β_{12}	15	152	574
get_constant_atom	β_{14}	38	518	1211
get_constant_int	β_{16}	28	396	1157
get_level	β_{18}	28	213	1054
get_list	β_{19}	20	275	763
get_struct	β_{20}	52	642	1766
get_value	β_{21}	43	488	1457
get_variable	β_{23}	43	549	1658
proceed	β_{24}	17	61	699
put_a_constant_atom	β_{25}	20	122	594
put_a_constant_int	β_{26}	20	122	506
put_constant_atom	β_{27}	37	274	1085
put_constant_int	β_{28}	37	274	997
put_value	β_{29}	21	183	910
retry_me_else	β_{30}	33	336	999
set_constant_atom	β_{31}	26	213	861
set_constant_int	β_{32}	25	183	767
set_variable	β_{33}	29	213	850
trust_me	β_{38}	29	336	973
try_me_else	β_{39}	30	457	1132
unequal	β_{40}	21	244	1021
unify_variable(nvar,var)	β_{42}	35	396	1309
unify_variable(var,nvar)	β_{43}	35	397	1309
unify_variable(int,int)	β_{44}	32	275	1179
unify_variable(atm,atm)	β_{46}	44	427	1413
unify_variable(str(1),str(1))	β_{47}	77	885	2560
unify_variable(list(1),list(1))	β_{45}	96	1068	3291
unify_variable(list(100),list(100))	β_{48}	4062	42511	217975

Table 4. Timing model for the WAM instructions. Cost of bytecodes when they succeed.

The experiments were made on the following representative platforms:

- Ultra**Sparc**-T1, 8 cores x 1GHz (4 threads per core), 8GB of RAM, SunOS 5.10.
- **Intel** Core Duo 1.66GHz, 2GB of RAM, Ubuntu Linux 7.04.
- Nokia **N810**. 400MHz processor, 128MB of RAM, Internet Tablet OS, Maemo Linux based OS2008 51.3

In order to reduce noise in the data because of spurious results, we have repeated each experiment 20 times and present the lowest results. In the calibration step 1000 repetitions were made (i.e., $r = 1000$). When possible, the tests were performed with the machines in single-user mode, stopping unnecessary processes. System tasks such as garbage collection, which, as mentioned before, is not considered in our model at the moment, were turned off.

Platform	Timing Model (ns)
Intel	$44 + 40.62 * length(X)$
N810	$427 + 425.11 * length(X)$
Sparc	$1413 + 2179.75 * length(X)$

Table 5. Timing model given by a linear function, for `unify_variable(X,Y)` when the arguments are lists of integers, and the instruction does not fail.

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
fail	β_{13}	0	0	0
get_constant_atom	β_{15}	32	457	1256
get_constant_int	β_{17}	26	366	1169
get_value	β_{22}	25	244	1106
unequal	β_{41}	11	61	651
unify_variable	β_{43}	121	1065	3867
unify_variable(const1,const2) const1 $\neq$ const2	β_{49}	41	154	697
unify_variable(int,int)	β_{50}	122	1035	3830
unify_variable(list(1),list(1))	β_{51}	338	3227	12229
unify_variable(atm,atm)	β_{52}	127	1126	4282
unify_variable(str(1),str(1))	β_{53}	223	2381	9239

Table 6. Timing model for the WAM instructions. Cost of bytecodes when they fail.

Tables 4 and 6 show the timing model for this WAM and the architectures studied. In the benchmarks used the `is/2` instruction is compiled into basic operations over pairs of numbers. The table shows the corresponding instructions named `arith_*`. We also have separated the cost of the instructions `put_constant`, `get_constant` when they are called for an atom or an integer. Note however, that their cost is very similar in most cases, but this will still help to reduce errors in the estimation. For the `unify_variable` instruction we have also included calibrations for several cases depending on the type and size of the input arguments in order to increase precision. In other cases, as mentioned in 5.4, the execution time of this instruction is not bounded by any a-priori known constant. Since, as also shown in Section 5.4, in our implementation it is possible to use functions instead of constants as timing model for a given instruction, in this table we include in the calibrations two data points for the case when the arguments are lists of integers, and for lists of size (length) 1 and 100 (β_{45} and β_{48} in Table 4). The value for an empty list is the same as for unifying any two equal atoms, i.e., β_{46} in Table 4. Table 5 shows the resulting timing model for `unify_variable` using these values to fit our linear model for this instruction.

Using the timing model shown in Tables 4, 5, and 6, we have performed some experiments with a series of programs on the three platforms (Intel, N810, and Sparc) in order to assess the accuracy of our technique for estimating execution times. The results of these experiments are shown in Tables 8 (Intel), 9 (N810), and 10 (Sparc).

Column **Pr. No.** lists the program identifiers, whose association with the programs and the input data sizes used is shown in Table 7. Column **Cost App.** indicates the type of approximation of the automatically inferred cost functions which estimate execution times (as a function on input data size): upper bound (**U**), lower bound (**L**), or exact (**E**). Such cost functions are shown in column **Cost Function** for the three different platforms considered

No.	Program	Data size
1	append(+A,+,-)	x=length(A)=150
2	evalpol(+A,+X,-)	x=length(A)=100
3	fib(+N,-)	x=N=16
4	hanoi(+N,+,+,+,-)	x=N=8
5	nreverse(+L,-)	x=length(L)=83
6	palindro(+A,-)	x=length(A)=9
7	powset(+A,-)	x=length(A)=11
8	list_diff(+L,+D,-)	x=length(L)=65 y=length(D)=65
9	list_inters(+L,+D,-)	x=length(L)=65 y=length(D)=65
10	substitute(+A,+B,-)	x=term_size(A)=67 y=term_size(B)=80
11	derive(+E,+,-)	x=term_size(E)=75

Table 7. List of program examples used in the experimental assessment.

in our experiments. The variables x and y represent the sizes of the input arguments to the programs which are relevant for the inference of the cost functions. The type of approximation directly depends on the one used by the static analysis described in Section 4 for estimating the number of executed instructions (as a function on input data size). The value **E** means that the lower and upper bound cost functions are the same, and thus, since the analysis is safe, this means that the exact cost function was inferred. Using the cost functions shown in column **Cost Function**, and in order to assess their accuracy, we have also estimated execution times for particular input data for each program and compared them with the observed execution times. These execution times are shown in columns **Est.** and **Obs.** respectively. Column **D.** shows the relative harmonic difference between the estimated and the observed time [2]. The source of inaccuracies in the execution time estimations of our technique come mainly from two sources: the timing model (which gives the execution time estimation of bytecodes, as shown in Tables 4 and 6)) and the static analysis (described in Section 4, which estimates the number of times that the bytecodes are executed, depending on the input data size). Since we are interested in identifying the source(s) of inaccuracies, we have also introduced the column **Prf.** It shows the result of estimating execution times using the timing model and assuming that the static analysis was perfect and obtained a function which provides the *exact* number of times that the bytecodes are executed. This obviously represents the case in which all loss of accuracy must be assigned to the timing model. The "perfect" cost model is obtained from an actual execution by instrumenting the profiler so that it records the number of times each instruction is executed for the application and the particular input data. Column **Pr.D.** shows the relative harmonic difference between **Prf.** and the observed execution time **Obs.**

The upper part of Tables 8, 9, and 10, up to the double line corresponds to examples where an exact cost function for the number of executed bytecodes was automatically inferred by the static analysis (note that, as expected, the values **Est.** and **Prf.** are the same). We can see that with an exact static analysis, the estimated execution times **Est.** are quite precise, which in turn supports the accuracy of our timing model.

It is particularly interesting to compare these results with those which were obtained using a variety of higher-level models in (19). Table 11 provides the standard deviation of the four high-level models of (19) as well as that of the abstract machine-based model presented in this paper, for the Intel platform and our set of bench-

[2] $rel\_harmonic\_diff(x, y) = (x - y)(1/x + 1/y)/2$.

Pr. No.	Cost. App.	Intel (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$0.73x + 0.21$	110	110	113	-2.4	-2.4
2	E	$0.69x + 0.19$	69	69	71	-2.3	-2.3
3	E	$0.69 \cdot 1.6^x + 0.21(-0.62)^x - 0.72$	1525	1525	1576	-3.3	-3.3
4	E	$-0.0042 \cdot 2^x + 0.73x \cdot 2^x - 0.86$	1501	1501	1589	-5.7	-5.7
5	E	$0.37x^2 + 0.49x + 0.12$	2569	2569	2638	-2.7	-2.7
6	E	$0.36 \cdot 2^x + 0.37x \cdot 2^x - 0.24$	1875	1875	2027	-7.8	-7.8
7	E	$0.91 \cdot 2^x + 0.87x - 0.6$	1868	1868	1931	-3.3	-3.3
8	L	$0.66x + 0.2$	43	68	81	-67.2	-17.8
	U	$0.78xy + 1.7x + 0.4$	3414	3569	3640	-6.4	-2.0
9	L	$0.83x + 0.2$	54	79	91	-54.6	-14.8
	U	$0.78xy + 1.7x + 0.4$	3414	3694	4011	-16.2	-8.2
10	L	$2x$	135	142	124	8.6	13.7
	U	$1.4xy + 1.4y + 6.1x + 4.1$	7922	2937	2858	120.6	2.7
11	L	$2.9x$	216	138	111	72.3	22.5
	U	$3x + 3$	226	216	162	34.0	29.5

Table 8. Observed and estimated execution time with cost functions, Intel platform (microseconds).

Pr. No.	Cost. App.	N810 (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$7.8x + 2.7$	1169	1169	1037	12.0	12.0
2	E	$7.8x + 2.7$	786	786	641	20.6	20.6
3	E	$8.3 \cdot 1.6^x + 2.5(-0.62)^x - 8.4$	18333	18333	14496	23.7	23.7
4	E	$0.74 \cdot 2^x + 7.8x \cdot 2^x - 10$	16095	16095	16144	-0.3	-0.3
5	E	$3.9x^2 + 5.7x + 1.6$	27247	27247	28381	-4.1	-4.1
6	E	$4.4 \cdot 2^x + 3.9x \cdot 2^x - 2.9$	20167	20167	20416	-1.2	-1.2
7	E	$9.5 \cdot 2^x + 10x - 6$	19517	19517	19653	-0.7	-0.7
8	L	$7.3x + 2.8$	474	744	640	-30.4	15.1
	U	$8.2xy + 19x + 5.5$	35849	37162	29266	20.4	24.1
9	L	$8.7x + 2.8$	569	839	732	-25.4	13.7
	U	$8.2xy + 19x + 5.5$	35849	38076	29907	18.2	24.4
10	L	$21x$	1399	1475	1068	27.3	32.9
	U	$15xy + 15y + 64x + 43$	85893	30375	25543	153.3	17.4
11	L	$29x$	2190	1423	854	108.7	53.3
	U	$30x + 30$	2306	2193	1342	56.8	51.1

Table 9. Observed and estimated execution time with cost functions, Nokia N810 platform (microseconds).

Model		Deviation
High Level	1	51.17 %
	2	31.06 %
	3	21.48 %
	4	58.45 %
Abs. Machine		4.72 %

Table 11. Comparison between the higher level models and the abstract machine-based model, on the Intel platform.

marks. It can be observed that the results obtained with the abstract machine-based model are more than five times better on the same platform than those obtained using the higher-level models.

With the abstract machine-based model, and for this type of programs we believe that the remaining error comes simply from the accumulated loss of accuracy of the bytecode instruction profiling and expect that making the *timing* model more precise will increase precision even further.

The lower part of Tables 8, 9, and 10 shows programs for which there is no unique value for $\texttt{Time}_p(n)$, where $\texttt{Time}_p(n)$ (as described in Section 4.1) denotes the cost (in time units) of computing a call to program p for an input of size n on a given platform. The reason is that for such programs, the number of instructions executed does not only depend on the input data sizes, but also depends on other characteristics of the input data (e.g., their actual values). Thus, for a given data size, there are actual lower and upper bounds for the cost of the program calls. For this reason, the two observed execution times shown in column **Obs.** for each program have been obtained by running the program with the input data, of the size specified in Table 7, that yield the actual lower and upper bounds to the execution times for such size. In this case, the static analysis infers approximations to such actual lower and upper bound cost functions (**L** and **U** respectively). These predictions are understandably much less accurate in these cases than those in the first part of the table, but still reasonable. In any case, lower bounds and upper bounds tend to be reasonably smaller or bigger than the observed execution times respectively. In general, for the programs in the lower part of the tables with big (absolute) values for **D.**, the (absolute) value for **Pr.D.** is reasonably small. This means that, in those cases, most of the inaccuracy in the estimation of execution times (**Est.**) comes from the static analysis, which does not approximate actual lower and upper bound cost functions accurately enough, and that the timing model used for predicting

Pr. No.	Cost. App.	Sparc (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$26x + 7.4$	3906	3906	4670	-18.0	-18.0
2	E	$25x + 7.1$	2543	2543	2985	-16.1	-16.1
3	E	$26 \cdot 1.6^x + 7.8(-0.62)^x - 27$	56828	56828	59120	-4.0	-4.0
4	E	$1.2 \cdot 2^x + 26x \cdot 2^x - 33$	53504	53504	63156	-16.7	-16.7
5	E	$13x^2 + 17x + 4.3$	90973	90973	109849	-19.0	-19.0
6	E	$13 \cdot 2^x + 13x \cdot 2^x - 8.5$	66400	66400	78980	-17.4	-17.4
7	E	$32 \cdot 2^x + 32x - 22$	66224	66224	78151	-16.6	-16.6
8	L	$24x + 7.1$	1574	2458	2991	-68.7	-19.7
	U	$27xy + 62x + 14$	118269	123733	129951	-9.4	-4.9
9	L	$30x + 7.1$	1940	2824	3394	-58.9	-18.5
	U	$27xy + 62x + 14$	118269	127378	133703	-12.3	-4.8
10	L	$68x$	4545	4821	4634	-1.9	4.0
	U	$48xy + 48y + 207x + 140$	277175	101779	111829	103.8	-9.4
11	L	$95x$	7104	4628	4038	59.6	13.7
	U	$98x + 98$	7454	7147	6081	20.5	16.2

Table 10. Observed and estimated execution time with cost functions, Sparc platform (microseconds).

the execution time of bytecodes is reasonably precise. Thus, we believe that using a better static analysis for inferring cost functions which take into account other characteristics of the input data, besides their sizes, would significantly improve the predictions. In any case, there is always a reasonable slack in the precision of the estimations due to the timing measurements and the timing model.

7. Conclusions and Future Work

We have developed a framework for estimating upper and lower bounds on the execution times of logic programs running on a bytecode-based abstract machine. We have shown that working at the abstract machine level allows taking into account low-level issues without having to tailor the analysis for each architecture and platform, and allows obtaining more accurate estimates than with previous approaches, including comparatively accurate upper and lower bound estimations of execution time.

Although the framework has been presented in the context of logic programs, we believe the technique can easily be applied to other languages. This adaptation of the approach, while certainly not trivial, to some extent would actually imply some simplification, since backtracking does not need to be taken into account. For example, analyses have been recently developed for Java bytecode (3) which infer the number of execution steps using similar techniques to those used in logic programming (12; 11; 13). Such analyses could be adapted, following the techniques presented herein, to take into account the bytecode timing information and would then be able to estimate actual execution time for Java programs. Appropriate cost models for Java bytecode are already being developed in (22).

We believe that the more accurate execution time estimates that can be obtained with our technique can be very useful in several contexts including parallelism, compilation, real-time applications, pervasive systems, etc. More concretely, increased timing precision can improve the effectiveness of resource/granularity control in parallel/distributed computing. This belief is based on previous experimental results, where it appeared that, even if improved precision in timing estimates is not essential, it does yield increased speedups. Also, the inferred cost functions can be used to develop automatic program optimization techniques. For example, they can be used for performing self-tuning specialization which compares statically the estimated execution time of different specialized versions (10).

Given that our experimental results are encouraging with respect to actually being able to find more accurate upper and lower bounds to program execution times, the approach may eventually also be used for verification (or falsification) of timing constraints, as in, for example, real-time systems, which was not possible in an accurate way with previous approaches. In fact, our approach (which can be adapted to take also into account destructive assignment, as in (20)) can potentially be used to solve a common problem in current WCET static analysis, where only constant WCET bounds (i.e., non dependent on input data sizes) are inferred. These bounds are not always appropriate since the WCET of a given program often depends on several input parameters, and using an absolute bound, covering all possible situations (i.e., all possible values or sizes of input), produces only a very gross over approximation (15). Substituting the estimated costs of the bytecodes by the actual worst-case costs of the instructions and using our approach, the WCET is expressed as a cost function parameterized by the size or values of input arguments, providing tighter WCET approximations. On the other hand, WCET work has produced more accurate (but, unfortunately, non-freely available) timing models which take into account many low-level parameters (such as cache behavior, pipeline state, etc.) which we have abstracted away in our work. It is clear that a combination of both techniques might be very useful in practice.

References

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Proc. of Static Analysis Symposium (SAS)*, LNCS. Springer-Verlag, July 2008. To appear.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[4] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.

[5] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[6] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Washington, DC, USA*, Apr. 2002.

[7] R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[8] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[9] S. Buettcher. Warren's Abstract Machine - A Java Implementation. `http://www.stefan.buettcher.org/cs/wam/index.html`.

[10] S.-J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[11] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[12] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[13] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[14] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[15] A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from Industrial WCET Analysis Case Studies. In R. Wilhelm, editor, *Proc. Fifth International Workshop on Worst-Case Execution Time (WCET) Analysis*, Palma de Mallorca, July 2005.

[16] G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *PEPM*. ACM Press, 2002.

[17] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[18] E. Y.-S. Hu, A. J. Wellings, and G. Bernat. Deriving java virtual machine timing models for portable worst-case execution time analysis. In *On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops*, volume 2889 of *LNCS*, pages 411–424. Springer, October 2003.

[19] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[20] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. Technical report, University of New Mexico, Department of Computer Science, UNM, January 2008. Submitted for publication.

[21] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[22] G. Román-Díez and G. Puebla. Java bytecode timing cost models. Technical Report CLIP12/2007.0, Technical University of Madrid, School of Computer Science, UPM, December 2007.

[23] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[24] R. Wilhelm. Timing analysis and timing predictability. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323. Springer, 2004.

Customizable Resource Usage Analysis for Java Bytecode

Jorge Navas,[1] Mario Méndez-Lojo,[1] Manuel V. Hermenegildo[1,2]

[1] Dept. of Computer Science, University of New Mexico (USA)
[2] Dept. of Computer Science, Tech. U. of Madrid (Spain) and IMDEA-Software

Abstract. Automatic cost analysis of programs has been traditionally studied in terms of a number of concrete, predefined *resources* such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, bytes sent or received by an application, number of files left open, number of SMSs sent or received, number of accesses to a database, money spent, energy consumption, etc. We present a fully automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of *application programmer-definable* resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis proposed is independent of the particular resource. We also present some experimental results from a prototype implementation of the approach covering an ample set of interesting resources.

1 Introduction

The usefulness of analyses which can infer information about the costs of computations is widely recognized since such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [21, 28, 29, 16, 8, 17, 32] for functional languages, [30, 7, 15, 34] for imperative languages, and [13, 12, 14, 26] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [23]), resource-oriented specialization (e.g., [10, 27]), or, more recently, certification of the resources used by mobile code (e.g., [11, 4, 9, 3, 18]). Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent rather than (or, rather, in addition to) the predefined, more traditional costs such as steps, time, or memory. Regarding the object of certification, in the case of mobile code the certification and checking process is often performed at the bytecode level [22], since, in addition to other reasons of syntactic convenience, bytecode is what is most often available at the receiving (checker) end.

We propose a fully automated framework which infers upper bounds on the usage that a Java bytecode program makes of *application programmer-definable* resources. Examples of such programmer-definable resources are bytes sent or received by an application over a socket, number of files left open, number of SMSs sent or received, number of accesses to a database, number of licenses consumed, monetary units spent, energy consumed, disk space used, and of course, execution steps (or bytecode instructions), time, or memory. In our context, resources are defined by programmers by means of *annotations*. The annotations defining each resource must provide for some user-selected elements corresponding to the bytecode program being analyzed (classes, methods, variables, etc.), a value that describes the cost of that element for that particular resource. These values can be constants or, more generally, functions of the input data sizes. The objective of our analysis is then to statically derive from these elementary costs an upper bound on the amount of those resources that the program as a whole (as well as individual blocks) will consume or provide.

Our approach builds on the work of [13, 12] for logic programs, where cost functions are inferred by solving recurrence equations derived from the syntactic structure of the program. Also, most previous work deals only with concrete, traditional resources (e.g., execution steps, time, or memory). The analysis of [26] is parametric but it is designed for Prolog and works at the source code level, and thus cannot be applied to Java bytecode due to particularities like virtual method invocation, unstructured control flow, assignment, the fact that statements are low-level bytecode instructions, the absence of backtracking (which has a significant impact on the method used in [26]), etc. More importantly, the presentation of [26] is descriptive in contrast to the concrete algorithm provided herein. In [1], a cost analysis is described that does deal with Java bytecode and is capable of deriving cost relations which are functions of input data sizes. However, while the approach proposed can conceptually be adapted to infer different resources, for each analysis developed the measured resource is fixed and changes in the implementation are needed to develop analyses for other resources. In contrast, our approach allows the application programmer to define the resources through annotations in the Java source, and without changing the analyzer in any way. In addition, the presentation in [1] is again descriptive, while herein we provide a concrete, memo table-based analysis algorithm, as well as implementation results.

2 Overview of the Approach

We start by illustrating the overall approach through a working example. The Java program in Fig. 1 emulates the process of sending text messages within a cell phone. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The phone (class `CellPhone)` receives a list of packets (`SmsPacket`), each one containing a single SMS, encodes them (`Encoder`), and sends them through a stream (`Stream`). There are two types of encoding: `TrimEncoder`, which eliminates any leading and trailing white spaces, and `UnicodeEncoder`, which converts any special character into its Unicode($\backslash uxxxx$) equivalent. The length of the SMS which the cell phone ultimately sends through the stream depends on the size of the encoded message.

A *resource* is a fundamental component in our approach. A resource is a user-defined notion which associates a basic cost function with some user-selected elements (class, method, statement) in the program. This is expressed by adding Java annotations to the code. The

```
import java.net.URLEncoder;
public class CellPhone {
 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
  if (smsPk != null) {
    String newSms = enc.format(smsPk.sms);
    stm.send(newSms);
    smsPk.next=sendSms(smsPk.next,enc,stm);
    smsPk.sms = newSms;
  }
  return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=size(s)")
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  @Cost({"cents","2*size(data)"})
  native void send(String data);
}
```

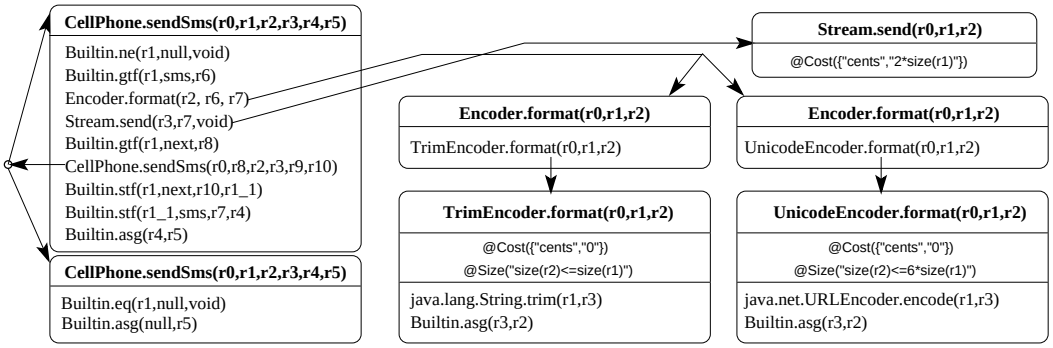


Fig. 1. Motivating example: Java source code and Control Flow Graph

objective of the analysis is to approximate the usage that the program makes of the resource. In the example, the resource is the cost in cents of a dollar for sending the list of text messages, since we will assume for simplicity that the carrier charges are proportional (2 cents/character) to the number of characters sent. This domain knowledge is reflected by the user in the method that is ultimately responsible for the communication (`Stream.send`), by adding the annotation `@Cost({"cents","2*size(data)"})`. Similarly, the formatting of an SMS done in any implementation of `Encoder.format` is free, as indicated by the `@Cost({"cents","0")})` annotation. The analysis understands these resource usage expressions and uses them to infer a safe upper bound on the total usage of the program.

Step 1: Constructing the Control Flow Graph. In the first step, the analysis translates the Java bytecode into an intermediate representation building a Control Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to our code example is also shown in Fig. 1.

The original `sendSms` method has been compiled into two block methods that share the same signature: class where declared, name (`CellPhone.sendSms`), and number and type of the formal parameters. The bottom-most box represents the base case, in which we re-

turn null, here represented as an assignment of `null` to the return variable r_5; the sibling corresponds to the recursive case. The virtual invocation of `format` has been transformed into a static call to a block method named `Encoder.format`. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in `TrimEncoder.format` and `UnicodeEncoder.format`. Note that the resource-related annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships. The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. For now, we can assume that size of (the contents of) a variable is the maximum number of pointers we need to traverse, starting at the variable, until `null` is found. The following equations are inferred by the analysis for the two `CellPhone.sendSms` block methods :

$$\begin{aligned}\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) &\leq 0\\ \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) &\leq 7 \times s_{r_1} - 6 + \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})\end{aligned}$$

The size of the returned value r_5 is independent of the sizes of the input parameters *this*, *enc*, and *stm* (s_{r_0}, s_{r_2} and s_{r_3} respectively) but not of the size s_{r_1} of the list of text messages *smsPk* (r_1 in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. In the example in Fig. 1, the user indicates that the formatting in `UnicodeEncoder` results in strings that are at most six times longer than the ones received as input `@Size("size(ret)<=6*size(s)")`, while the trimming in `TrimEncoder` returns strings that are equal or shorter than the input (`@Size("size(ret)<=size(s)")`). The equation system shown above is approximated by a recurrence solver included in our analysis in order to obtain the closed form solution $\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 3.5 \times s^2_{r_1} - 2.5 \times s_{r_1}$.

Step 3: Resource Usage Analysis. In the this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps to infer a resource usage equation for each block method in the CFG (possibly simplifying such equations) and obtain closed form solutions (in general, approximated –upper bounds). Therefore, the objective of the resource analysis is to statically derive safe upper bounds on the amount of resources that each of the block methods in the CFG consumes or provides. The result given by our analysis for the monetary cost of sending the messages (`CellPhone.sendSms`) is

$$\begin{aligned}Cost_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) &\leq 0\\ Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) &\leq 12 \times s_{r_1} - 12 + Cost_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})\end{aligned}$$

i.e., the cost is proportional to the size of the message list (`smsPk` in the source, r_1 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 6 \times s^2_{r_1} - 6 \times s_{r_1}$.

3 Intermediate program representation

Analysis of a Java bytecode program normally requires its translation into an intermediate representation that is easier to manipulate. In particular, our decompilation (assisted by the Soot [31] tool) involves elimination of stack variables, conversion to three-address statements, static single assignment (SSA) transformation, and generation of a Control Flow Graph

(CFG) that is ultimately the subject of analysis. The decompilation process is an evolution of the work presented in [25], which has been successfully used as the basis for other (non resource-related) analyses [24]. Our ultimate objective is to support the full Java language but the current transformation has some limitations: it does not yet support reflection, threads, or runtime exceptions. The following grammar describes the intermediate representation; some of the elements in the tuples are named so we can refer to them as *node*.name.

$$
\begin{array}{lll}
CFG & ::= & Block^{+} \\
Block & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{fpars}{:}Id^{+},\mathsf{annot}{:}expr^{*},\mathsf{body}{:}Stmt^{*}) \\
Sig & ::= & (\mathsf{class}{:}Type,\mathsf{name}{:}Id,\mathsf{pars}{:}Type^{+}) \\
Stmt & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{apars}{:}(Id|Ct)^{+}) \\
Var & ::= & (\mathsf{name}{:}Id,\mathsf{type}{:}Type)
\end{array}
$$

The Control Flow Graph is composed of *block methods*. A block method is similar to a Java method, with some particularities: a) if the program flow reaches it, every statement in it will be executed, i.e, it contains no branching; b) its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types; c) it always includes as formal parameters the returned value *ret* and, unless it is static, the instance self-reference *this*; d) for every formal parameter (*input* formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the block method that contains its final version in the SSA transformation (*output* formal parameter); e) every statement in a block method is an invocation, including builtins (assignment `asg`, field dereference `gtf`, etc.), which are understood as block methods of the class `Builtin`.

As mentioned before, there is no branching within a block method. Instead, each conditional `if` *cond* $stmt_1$ `else` $stmt_2$ in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of decompiling *cond*, while we attach $\overline{cond}$ to its sibling. A similar approach is used in virtual invocations, for which we introduce as many block methods in the graph as possible receivers of the call were in the original program. A set of block methods with the same signature *sig* can be retrieved by the function $getBlocks(CFG, sig)$.

User specifications are written using the annotation system introduced in Java 1.5 which, unlike JML specifications, has the very useful characteristic of being preserved in the bytecode. Annotations are carried over to our CFG representation, as can be seen in Fig. 1.

Example 1 We now focus our attention on the two block methods in Fig. 1, which are the result of (de)compiling the `CellPhone.sendSms` method. Input formal parameters r_0, r_1, r_2, r_3 correspond to $this, smsPk, enc$, and stm, respectively. In the case of r_1, the contents of its fields *next* and *sms* are altered by invoking the `stf` and accessed by invoking the `gtf` (abbreviation for `setfield` and `getfield`, respectively) builtin block methods. The output formal parameter r_4 contains the final state of r_1 after those modifications. The value returned by the block methods is contained in r_5. Space reasons prevent us from showing any type information in the CFG in Fig 1. In the case of `Encoder.format`, for example, we say that there are two blocks with the same signature because they are both defined in class `Encoder`, have the same name (`format`) and list of types of formal parameters {`Encoder,String,String`}.

```
resourceAnalysis(CFG, res) {
 CFG←classAnalysis(CFG)
 mt←initialize(CFG)
 dg←dataDependencyAnalysis(CFG, mt)
 for (SCC:SCCs)
   //in reverse topological order
   mt←genSizeEqs(SCC, mt, CFG, dg)
   mt←genResourceUsageEqs(SCC, res, mt, CFG)
 return mt
}
```

```
normalize(Eqs) {
 for (size relation p ≤ e1:Eqs)
   do
     if (expression s appears in e1
         and s ≤ e2 ∈Eqs)
       replace ocurrences of s in e1 with e2
   while there is change
 return Eqs
}
```

Fig. 2. Generic resource analysis algorithm and normalization.

4 The resource usage analysis framework

We now describe our framework for inferring upper bounds on the usage that the Java bytecode program makes of a set of resources defined by the application programmer, as described before. The algorithm in Fig 2 takes as input a Control Flow Graph in the format described in the previous section, including the user annotations that assign elementary costs to certain graph elements for a particular resource. The user also indicates the set of resources to be tracked by the analysis. Without loss of generality we assume for conciseness in our presentation a single resource.

A preliminary step in our approach is a class hierarchy analysis [5, 24], aimed at simplifying the CFG and therefore improving overall precision. Then, another analysis is performed over the CFG to extract data dependencies, as described below. The next step is the decomposition of the CFG into its strongly-connected components. After these steps, two different analyses are run separately on each strongly connected component: a) the size analysis, which estimates parameter size relationships for each statement and output formal parameters as a function of the input formal parameter sizes (Sec. 4.1); and b) the actual resource analysis, which computes the resource usage of each block method in terms also of the input data sizes (Sec. 4.2). Each phase is dependent on the previous one.

The *data dependency analysis* is a dataflow analysis that yields *position dependency graph*s for the block methods within a strongly connected component. Each graph $G = (V, E)$ represents data dependencies between positions corresponding to statements in the same block method, including its formal parameters. Vertexes in V denote positions, and edges $(s_1, s_2) \in E$ denote that s_2 is dependent on s_1 (s_1 is a *predecessor* of s_2). We will assume a `predec` function that takes a position dependency graph, a statement, and a parameter position and returns its nearest predecessor in the graph. Fig. 3 shows the position dependency graph of the `TrimEncoder.format` block method.

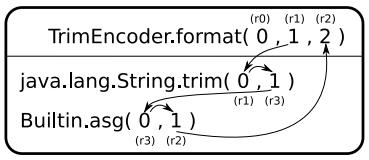


Fig. 3.

4.1 Size analysis

We now show our algorithm for estimating parameter size relations based on the data dependency analysis, inspired by the original ideas of [13, 12]. Also, we provide a concrete algorithm for performing the analysis, rather than the more descriptive presentations of the

related work discussed previously. Our goal is to represent input and output size relationships for each statement as a function defined in terms of the formal parameter sizes. Unless otherwise stated, whenever we refer to a parameter we mean its position.

The size of an input is defined in terms of measures. By *measure* we mean a function that, given a data structure, returns a number. Our method is parametric on measures, which can be defined by the user and attached via annotations to parameters or classes. For concreteness, we have defined herein two measures, int for integer variables, and the *longest path-length* [1] ref for reference variables. The longest path-length of a variable is the cardinality of the longest chain of pointers than can be followed from it. More complex measures can be defined to handle other data types such as cyclic structures, arrays, etc. The set of measures will be denoted by $\mathcal{M}$.

The size analysis algorithm is given in pseudo-code in Fig. 4; its main steps are:

1. Assign an upper bound to the size of every parameter position of all statements, including formal parameters, for all the block methods with the same signature (genSigSize).
2. For a given signature, take the set of size inequations returned by (1) and rename each size relation in terms of the sizes of input formal parameters (normalize).
3. Repeat the first step for every signature in the same strongly-connected component (genSizeEqs).
4. Simplify size relationships by resolving mutually recursive functions, and find closed form solutions for the output formal parameters (genSizeEqs).

Intermediate results are cashed in a memo table mt, which for every parameter position stores measures, sizes, and resource usage expressions defined in the $\mathcal{L}$ language:

$$
\begin{array}{lll}
\langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid (\sum \mid \prod) \langle expr \rangle \\
& \mid & \langle expr \rangle^{\langle expr \rangle} \mid log_{num} \langle expr \rangle \mid -\langle expr \rangle \\
& \mid & \langle expr \rangle ! \mid \infty \mid \text{num} \\
& \mid & \text{size}([\langle measure \rangle,]\text{arg(r num)}) \\
\langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\
\langle measure \rangle & ::= & \textbf{int} \mid \textbf{ref} \mid \ldots
\end{array}
$$

The size of the parameter at position i in statement $stmt$, under measure m, is referred to as $\texttt{size}(m, stmt, i)$. We consider a parameter position to be *input* if it is bound to some data when the statement is invoked. Otherwise, it is considered an *output parameter position*. In the case of input parameter and output formal parameter positions, an upper bound on that size is returned by getSize (Fig. 4). The upper bound can be a concrete value when there is a constant in the referred position, i.e., when the val function returns a non-infinite value:

Definition 1. *The concrete size value for a parameter position under a particular measure is returned by* $\texttt{val} : \mathcal{M} \times \mathcal{S}tmt \times \mathbb{N} \rightarrow \mathcal{L}$*, which evaluates the* syntactic *content of the actual parameter in that position:*

$$
\texttt{val}(m, stmt, i) = \begin{cases} n \text{ if } stmt.\mathsf{apars}_i \text{ is an integer } n \text{ and } m{=}\textbf{int} \\ 0 \text{ if } stmt.\mathsf{apars}_i \text{ is } \texttt{null} \text{ and } m{=}\textbf{ref} \\ \infty \text{ otherwise} \end{cases}
$$

If the content of that input parameter position is a variable, the algorithm searches the data dependency graph for its immediate predecessor. Since the intermediate representation is in SSA form, the only possible scenarios are that either there is a unique predecessor

```
genSizeEqs(SCC,mt,CFG,dg) {
 Eqs← ∅^|SCC|
 for (sig: SCC)
   Eqs[sig]←genSigSize(sig,mt,SCC,CFG,dg)
 Sols←recEqsSolver(simplifyEqs(Eqs))
 for (sig:SCC)
   insert(mt,size,sig,Sols[sig])
 return mt
}

genSigSize(sig,mt,SCC,CFG,dg) {
 Eqs← ∅
 BMS←getBlocks(CFG,sig)
 for (bm:BMs)
   Eqs←Eqs ∪ genBlockSize(bm,mt,SCC,dg)
 return normalize(Eqs)
}

genBlockSize(bm,mt,SCC,dg) {
 Eqs← ∅
 for (stmt:bm.body)
   I←stmt input parameter positions
   Eqs←Eqs ∪ genInSize(stmt,I,mt,dg)
   Eqs←Eqs ∪ genOutSize(stmt,mt,SCC)
 K← bm output formal parameter positions
 Eqs←Eqs ∪ genInSize(bm,K,mt,dg)
 return Eqs
}

genInSize(elem,Pos,mt,dg) {
 Eqs← ∅
 for (pos:Pos)
   m←lookup(mt,measure,elem.sig,pos)
   s←getSize(m,elem.id,pos,dg)
   Eqs←Eqs ∪ {size(m,elem.id,pos)≤s}
 return Eqs
}

genOutSize(stmt,mt,SCC) {
 {i_1,...,i_l} ← stmt input positions
 sig←stmt.sig
 {m_{i_1},...,m_{i_l}} ←{lookup(mt,measure,sig,i_1),...,
                         lookup(mt,measure,sig,i_l)}
 {s_{i_1},...,s_{i_l}} ← {size(m_{i_1},stmt.id,i_1),...,
                          size(m_{i_l},stmt.id,i_l)
 Eqs← ∅
 O← stmt output parameter positions
 for (o:O)
   m_o ←lookup(mt,measure,sig,o)
   if (sig∉SCC)
     Size_user ← A^o_sig(s_{i_1},...,s_{i_l})
     Size_alg' ←max(lookup(mt,size,sig,o))
     Size_alg ←Size_alg'(s_{i_1},...,s_{i_l})
     Size_o ←min(Size_user,Size_alg)
   else
     Size_o ← Size^o_sig(m_o,s_{i_1},...,s_{i_l})
   Eqs←Eqs ∪ {size(m_o,stmt.id,o)≤ Size_o}
 return Eqs
}

getSize(m,id,pos,dg) {
 result←val(m,id,i)
 if (result≠∞)
   return result
 else
   if (∃(elem,pos_p)∈ predec(dg,id,pos))
     m_p ←lookup(mt,measure,elem.sig,pos_p)
     if (m=m_p)
       return size(m_p,elem.id,pos_p)
 return ∞
}
```

Fig. 4. The size analysis algorithm

whose size is assigned to that input parameter position, or there is none, causing the input parameter size to be unbounded (∞).

Consider now an output parameter position within a block method, case covered in genOutSize (Fig. 4). If the output parameter position corresponds to a non-recursive invoke statement, either a size relationship function has already been computed recursively (since the analysis traverses each strongly-connected component in reverse topological order), or it is provided by the user through size annotations. In the first case, the size function of the output parameter position can be retrieved from the memo table by using the lookup operation, taking the maximum in case of several size relationship functions, and then passing the input parameter size relationships to this function to evaluate it. In the second scenario, the size function of the output parameter position is provided by the user through size annotations, denoted by the $\mathcal{A}$ function in the algorithm. In both cases, it will able to return an explicit size relation function.

Example 2 We have already shown in the `CellPhone` example how a class can be annotated. The `Builtin` class includes the assignment method `asg`, annotated as follows:

```
public class Builtin {
  @Size{"size(ret)<=size(o)"}
  public static native Object asg(Object o);
  // ...rest of annotated builtins
}
```

which results in equation $\mathcal{A}^1_{\mathsf{asg}}(\mathtt{ref}, \mathtt{size}(\mathtt{ref}, asg, 0)) \leq \mathtt{size}(\mathtt{ref}, asg, 0)$.

If the output parameter position corresponds to a recursive invoke statement, the size relationships between the output and input parameters are built as a symbolic size function. Since the input parameter size relations have already been computed, we can establish each output parameter position size as a function described in terms of the input parameter sizes.

At this point, the algorithm has defined size relations for all parameter positions within a block method. However, those relations are either constants or given in terms of the immediate predecessor in the dependency graph. The algorithm rewrites the equation system such that we obtain an equivalent system in which only formal parameter positions are involved. This process, called *normalization*, is shown in Fig. 2

After normalization, the analysis repeats the same process for all block methods in the same strongly-connected component (SCC). Once every component has been processed, the analysis further simplifies the equations in order to resolve mutually recursive calls among block methods within the same SCC in the simplifyEqs procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver (recEqsSolver, called from genSizeEqs) in order to obtain approximated upper-bound closed forms. The interesting subject of how the equations are solved is beyond the scope of this paper (see, e.g., [33]). Our implementation does provide a dedicated implementation (an evolution of the solver of the Caslog system [12]) which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients, divide and conquer recurrence equations, etc. In addition, the system has interfaces to external solvers such as Purrs [6] or Mathematica.

Example 3 We now illustrate the definitions and algorithm with an example of how the size relations are inferred for the two `CellPhone.sendSms` block methods (Fig. 1), using the ref measure for reference variables. We will refer to the k-th occurrence of a statement $stmt$ in a block method as $stmt_k$, and denote `CellPhone.sendSms`, `Encoder.format`, and `Stream.send` by `sendSms`, `format`, and `send` respectively. Finally, we will refer to the size of the input formal parameter position i, corresponding to variable r_i, as s_{r_i}.

The main steps in the process are listed in Fig. 5. The first block of rows contains the most relevant size parameter relationship equations for the recursive block method, while the second block of rows corresponds to the base case. These size parameter relationship equations are constructed by the analysis by first following the algorithm in Fig. 4, and then normalizing them (expressing them in terms of the input formal parameter sizes s_{r_i}). Also, in the first block of rows we observe that the algorithm has returned $6 \times \mathtt{size}(\mathtt{ref}, format, 1)$ as upper bound for the size of the formatted string, $\mathsf{max}(\mathsf{lookup}(mt, \mathtt{size}, format, 2))$. The result is the maximum of the two upper bounds given by the user for the two implementations for `Encoder.format` since `TrimEncoder.format` eliminates any leading and trailing white

Size parameter relationship equations (normalized)

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, ne, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, ne, 1) & \leq \texttt{val}(\texttt{ref}, ne, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, gtf_1, 0) & \leq \texttt{size}(\texttt{ref}, ne, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_1, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_1, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 1) & \leq \texttt{size}(\texttt{ref}, gtf_1, 2) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 2) & \leq \texttt{max}(\texttt{lookup}(mt, \texttt{size}, format, 2))(\texttt{size}(\texttt{ref}, format, 2)) \\
 & \leq \texttt{max}(s_{r1}, 6 \times s_{r1})(s_{r_1} - 1) \\
 & \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, send, 1) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, gtf_2, 0) & \leq \texttt{size}(\texttt{ref}, gtf_1, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_2, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_2, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, sendSms, 1) & \leq \texttt{size}(\texttt{ref}, gtf_2, 2) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, sendSms, 5) & \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, \_, \texttt{size}(\texttt{ref}, sendSms, 1), \_, \_) \\
 & \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 0) & \leq \texttt{size}(\texttt{ref}, gtf_2, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, stf_1, 2) & \leq \texttt{size}(\texttt{ref}, sendSms, 5) \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_1, 0), \_, \texttt{size}(\texttt{ref}, stf_1, 2)) \\
 & \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 0) & \leq \texttt{size}(\texttt{ref}, stf_1, 3) \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 2) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, stf_2, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_2, 0), \_, \texttt{size}(\texttt{ref}, stf_2, 2)) \\
 & \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{size}(\texttt{ref}, stf_2, 3) \\
 & \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \\
 & \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})
\end{array}
$$

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, eq, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, eq, 1) & \leq \texttt{val}(\texttt{ref}, eq, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{val}(\texttt{ref}, asg, 0) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq 0
\end{array}
$$

Output parameter size functions for builtins (provided through annotations)

$$\mathcal{A}^2_{\texttt{gtf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf, 0), \_) \leq \texttt{size}(\texttt{ref}, gtf, 0) - 1$$
$$\mathcal{A}^1_{\texttt{asg}}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq \texttt{size}(\texttt{ref}, asg, 0)$$
$$\mathcal{A}^3_{\texttt{stf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf, 0), \_, \texttt{size}(\texttt{ref}, stf, 2)) \leq \texttt{size}(\texttt{ref}, stf, 0) + \texttt{size}(\texttt{ref}, stf, 2)$$

Simplified size equations and closed form solution

$$\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}$$

$$\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 3.5 \times s^2_{r1} - 2.5 \times s_{r1}$$

Fig. 5. Size equations example

spaces (thus the output is at most as bigger as the input), whereas `UnicodeEncoder.format` converts any special character into its Unicode equivalent (thus the output is at most six times the size of the input), a safe upper bound for the output parameter position size is given by the second annotation.

```
genResourceUsageEqs(SCC, res, mt, CFG) {
  Eqs ← ∅^|SCC|
  for (sig : SCC)
    Eqs[sig] ← genSigRU(sig, res, mt, SCC, CFG)
  Sols ← recEqsSolver(simplifyEqs(Eqs))
  for (sig : SCC)
    insert(mt, cost, max(Sols[sig]))
  return mt
}

genSigRU(sig, res, mt, SCC, CFG) {
  Eqs ← ∅
  BMs ← getBlocks(CFG, sig)
  for (bm : BMs)
    body ← bm.body
    Cost_body ← 0
    for (stmt : body)
      Cost_stmt ← genStmtRU(stmt, res, mt, SCC)
      Cost_body ← Cost_body + Cost_stmt
    Cost_bm ← genBlockRU(bm, res, mt)
    Eqs ← Eqs ∪ {Cost_bm ≤ Cost_body}
}

genStmtRU(stmt, res, mt, SCC) {
  {i_1, ..., i_k} ← stmt input parameter positions
  {s_{i_1}, ..., s_{i_k}} ←
      {max(lookup(mt, size, stmt.sig, i_1)), ...,
       max(lookup(mt, size, stmt.sig, i_k))}
  if (stmt.sig ∉ SCC)
    Cost_user ← A_{stmt.sig}(res, s_{i_1}, ..., s_{i_k})
    Cost_alg' ← lookup(mt, cost, res, stmt.sig)
    Cost_alg ← Cost_alg'(s_{i_1}, ..., s_{i_k})
    return min(Cost_alg, Cost_user)
  else return Cost(stmt.sig, res, s_{i_1}, ..., s_{i_k})
}

genBlockRU(bm, res, mt) {
  {i_1, ..., i_l} ← bm input formal parameter positions
  {s_{i_1}, ..., s_{i_l}} ←
      {lookup(mt, size, bm.id, i_1), ...,
       lookup(mt, size, bm.id, i_l)
  return Cost(bm.id, res, s_{i_1}, ..., s_{i_l})
}
```

Fig. 6. The resource usage analysis algorithm

In the particular case of builtins and methods for which we do not have the code, size relationships are not computed but rather taken from the user `@Size` annotations. These functions are illustrated in the third block of rows. Finally, in the fourth block of rows we show the recurrence equations built for the output parameter sizes in the block method and in the final row the closed form solution obtained.

4.2 Resource usage analysis

The core of our framework is the resource usage analysis, whose pseudo code is shown in Fig 6. It takes a strongly-connected component of the CFG, including the set of annotations which provide the application programmer-provided resources and cost functions, and calculates an resource usage function which is an upper bound on the usage made by the program of those resources. The algorithm manipulates the same memo table described in Sec. 4.1 in order to avoid recomputations and access the size relationships already inferred.

The algorithm is structured in a very similar way to the size analysis (which also allows us to draw from it to keep the explanation within space limits): for each element of the strongly-connected component the algorithm will construct an equation for each block method that shares the same signature representing the resource usage of that block. To do this, the algorithm will visit each invoke statement. There are three possible scenarios, covered by the genStmtRU function. If the signatures of caller and callee(s) belong to the same strongly-connected component, we are analyzing a recursive invoke statement. Then, we add to the body resource usage a symbolic resource usage function, in an analogous fashion to the case of output parameters in recursive invocations during the size analysis.

The other scenarios occur when the invoke statement is non-recursive. Either a resource usage function $Cost_{alg}$ for the callee has been previously computed, or there is a user anno-

tation $Cost_{usr}$ that matches the given signature, or both. In the latter case, the minimum between these two functions is chosen (i.e., the most precise safe upper bound assigned by the analysis to the resource usage of the non-recursive invoke statement).

Example 4 The call (sixth statement) in the upper-most `CellPhone.sendSms` block method matches the signature of the block method itself and thus it is recursive. The first four parameter positions are of input type. The upper-bound expression returned by genStmtRU is $Cost_{sendSms}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$. Note that the input size relationships were already normalized during the size analysis. Now consider the invocation of `Stream.send`. The resource usage expression for the statement is defined by the function $\mathcal{A}_{send}(\$, \_, 6\times(s_{r1}-1))$ since the input parameter at position one is at most six times the size of the second input formal parameter, as calculated by the size analysis in Fig. 5. Note also that there is a resource annotation `@Cost({"cents","2*size(r1)"})` attached to the block method describing the behavior of $\mathcal{A}_{send}$ and yielding the expression $Cost_{user} = 12\times(s_{r1}-1)$. On the other hand, the absence of any callee code to analyze –the original method is native– results in $Cost_{alg} = \infty$. Then, the upper bound obtained by the analysis for the statement is $\min(Cost_{alg}, Cost_{user}) = Cost_{user}$.

At this point, the analysis has built a resource usage function (denoted by $Cost_{body}$) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form $Cost_{block} \leq Cost_{body}$ where $Cost_{block}$ is again a symbolic resource usage function built by replacing each input formal parameter position with its size relations in that block method. These resource usage equations are simplified by calling simplifyEqs and, finally, they are solved calling recEqsSolver, both already defined in Sec. 4.1. This process yields an (in general, approximate, but always safe) closed form upper bound on the resource usage of the block methods in each strongly-connected component. Note that given a signature the analysis constructs a closed form solution for every block method that shares that signature. These solutions approximate the resource usage consumed in or provided by each block method. In order to compute the total resource usage of the signature the analysis returns the maximum of these solutions yielding a safe global upper bound.

Example 5 The resource usage equations generated by our algorithm for the two `sendSms` block methods and the "$" resource (monetary cost of sending the SMSs) are listed in Fig. 7. The computation is partially based on the size relations in Fig. 5. The resource usage of each block method is calculated by building an equation such that the left part is a symbolic function constructed by replacing each parameter position with its size (i.e., $Cost_{sendSms}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3})$ and $Cost_{sendSms}(\$, s_{r0}, 0, s_{r2}, s_{r3})$), and the rest of the equation consists of adding the resource usage of the invoke statements in the block method. These are calculated by computing the minimum between the resource usage function inferred by the analysis and the function provided by the user. The equations corresponding to the recursive and non-recursive block methods are in the first and second row, respectively. They can be simplified (third row) and expressed in closed form (fourth row), obtaining a final upper bound for the charge incurred by sending the list of text messages of $6 \times s_{r1}^2 - 6 \times s_{r1}$.

5 Experimental results

We have completed an implementation of our framework, and tested it for a representative set of benchmarks and resources. Our experimental results are summarized in Table 1. Column

Resource usage equations

$$Cost_{sendSms}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, ne)}^{\infty}, \overbrace{\mathcal{A}_{ne}(\$, s_{r1}, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, gtf)}^{\infty}, \overbrace{\mathcal{A}_{gtf}(\$, s_{r1}, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, format)(\_, s_{r1}-1)}^{0}, \overbrace{\mathcal{A}_{format}(\$, \_, s_{r1}-1)}^{\infty})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, send)}^{\infty}, \overbrace{\mathcal{A}_{send}(\$, \_, 6 \times (s_{r1}-1))}^{@\mathtt{Cost}(\text{"cents"},\text{"2*size(r1)"})=12\times(s_{r1}-1)})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, gtf)}^{\infty}, \overbrace{\mathcal{A}_{gtf}(\$, s_{r1}, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0}) + Cost_{sendSms}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, stf)}^{\infty}, \overbrace{\mathcal{A}_{stf}(\$, s_{r1}, \_, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, stf)}^{\infty}, \overbrace{\mathcal{A}_{stf}(\$, s_{r1}, \_, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0})$$
$$+\min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, asg)}^{\infty}, \overbrace{\mathcal{A}_{asg}(\$, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0})$$
$$\leq 12 \times (s_{r1}-1) + Cost_{sendSms}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$$

$$Cost_{sendSms}(\$, s_{r0}, 0, s_{r2}, s_{r3}) \leq \min(\overbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, eq)}^{\infty}, \overbrace{\mathcal{A}_{eq}(\$, 0, \_)}^{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0}) + \min(\underbrace{\mathsf{lookup}(mt, \mathtt{cost}, \$, asg)}_{\infty}, \underbrace{\mathcal{A}_{asg}(\$, 0))}_{@\mathtt{Cost}(\text{"cents"},\text{"0"})=0}) \leq 0$$

Simplified resource usage equations and closed form solution

$$Cost_{sendSms}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 12 * s_{r1} - 12 + Cost_{sendSms}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}$$

$$Cost_{sendSms}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 6 \times s_{r1}^2 - 6 \times s_{r1}$$

Fig. 7. Resource equations example

Program provides the name of the main class to be analyzed. Column **Resource(s)** shows the resource(s) defined and tracked. Column t_s shows the time (in milliseconds) required by the size analysis to construct the size relations (including the data dependency analysis and class hierarchy analysis) and obtain the closed form. Column t_r lists the time taken to build the resource usage expressions for all method blocks and obtain their closed form solutions. t provides the total times for the whole analysis process. Finally, column **Resource Usage Func.** provides the upper bound functions inferred for the resource usage. For space reasons, we only show the most important (asymptotic) component of these functions, but the analysis yields concrete functions with constants.

Regarding the benchmarks we have covered a reasonable set of data-structures used in object-oriented programming and also standard Java libraries used in real applications. We have also covered an ample set of application-dependent resources which we believe can be relevant in those applications. In particular, not only have we represented high-level resources such as cost of SMS, bytes received (including a coarse measure of bandwidth, as

Program	Resource(s)	t_s	t_r	t	Resource Usage Func.	
BST	Heap usage	250	22	367	$O(2^n)$	$n \equiv$ tree depth
CellPhone	SMS monetary cost	271	17	386	$O(n^2)$	$n \equiv$ packets length
Client	Bytes received and	391	38	527	$O(n)$	$n \equiv$ stream length
	bandwidth required				$O(1)$	—
Dhrystone	Energy consumption	602	47	759	$O(n)$	$n \equiv$ int value
Divbytwo	Stack usage	142	13	219	$O(log_2(n))$	$n \equiv$ int value
Files	Files left open and	508	53	649	$O(n)$	$n \equiv$ number of files
	Data stored				$O(n \times m)$	$m \equiv$ stream length
Join	DB accesses	334	19	460	$O(n \times m)$	$n, m \equiv$ records in tables
Screen	Screen width	388	38	536	$O(n)$	$n \equiv$ stream length

Table 1. Times of different phases of the resource analysis and resource usage functions.

a ratio of data per program step), and files left open, but also other low-level (i.e., bytecode level) resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program *Files* (a fragment characteristic of operating system kernel code) we kept track of the number of file descriptors left open. The data inferred for this resource can be clearly useful, e.g., for debugging: the resource usage function inferred in this case ($O(n)$) denotes that the programmer did not close $O(n)$ file descriptors previously opened. In program *Join* (a database transaction which carries out accesses to different tables) we decided to measure the number of accesses to such external tables. This information can be used, e.g., for resource-oriented specialization in order to perform optimized checkpoints in transactional systems. The rest of the benchmarks include other definitions of resources which are also typically useful for verifying application-specific properties: *BST* (a generic binary search tree, used in [2] where a heap space analysis for Java bytecode is presented), *CellPhone* (extended version of program in Figure 1), *Client* (a socket-based client application), *Dhrystone* (a modified version of a program from [20] where a general framework is defined for estimating the energy consumption of embedded JVM applications; the complete table with the energy consumption costs that we used can be found there), *DivByTwo* (a simple arithmetic operation), and *Screen* (a MIDP application for a cellphone, where the analysis is used to make sure that message lines do not exceed the phone screen width). The benchmarks also cover a good range of complexity functions ($O(1), O(log(n), O(n), O(n^2) \ldots, O(2^n), \ldots$) and different types of structural recursion such as simple, indirect, and mutual. The code for these benchmarks and a demonstrator are available at `http://www.cs.unm.edu/~jorge/RUA`.

6 Conclusions

We have presented a fully-automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. Our analysis derives a vector of functions, one for each defined resource. Each of these functions returns, for each given set of input data sizes, an upper bound on the usage that the whole program (and each individual method) make of the corresponding resource. Important novel aspects of our approach are the fact that it allows the application programmer to define the resources to be tracked by writing simple resource descriptions via source-level annotations, as well as the fact that we have provided a concrete analysis algorithm and report on an

implementation. The current results show that the proposed analysis can obtain non-trivial bounds on a wide range of interesting resources in reasonable time. Another important aspect of our work, because of its impact on the scalability, precision, and automation of the analysis, is that our approach allows using the annotations also for a number of other purposes such as stating the resource usage of external methods, which is instrumental in allowing modular composition and thus scalability. In addition, our annotations allow stating the resource usage of any method for which the automatic analysis infers a value that is not accurate enough to prevent inaccuracies in the automatic inference from propagating. Annotations are also used by the size and resource usage analysis to express their output. Finally, the annotation language can also be used to state specifications related to resource usage, which can then be proved or disproved based on the results of analysis following, e.g., the scheme of [19] thus finding bugs or verifying (the resource usage of) the program.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
2. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.
3. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.
4. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.
5. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.
6. R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.
7. I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing*, Apr. 2002.
8. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.
9. Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.
10. S.J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.
11. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
12. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.
13. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.
14. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.
15. J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proc. of DDECS*. IEEE Computer Society, 2006.

16. G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.
17. B. Grobauer. Cost recurrences for DML programs. In *Int'l. Conf. on Functional Programming*, pages 253–264, 2001.
18. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.
19. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
20. Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.
21. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *TOPLAS*, 10(2), 1988.
22. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
23. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.
24. M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.
25. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.
26. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, LNCS, 2007.
27. G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of PPDP'06*, pages 261–271. ACM Press, 2006.
28. M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.
29. D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.
30. Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour*, 2004.
31. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.
32. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.
33. H. S. Wilf. *Algorithms and Complexity*. A.K. Peters Ltd, 2002.
34. R. Wilhelm. Timing analysis and timing predictability. In *Proc. FMCO*, LNCS. Springer-Verlag, 2004.

Bytecode 2009 Preliminar Version

User-Definable Resource Usage Bounds Analysis for Java Bytecode

Jorge Navas[1,4]

[1] *School of Computing*
National University of Singapore
Republic of Singapore

Mario Méndez-Lojo[2,4]

[2] *Department of Computer Science*
University Texas at Austin
Austin, TX (USA)

Manuel V. Hermenegildo[3,4,5]

[3] *IMDEA-Software, Madrid (Spain),*
Departments of Computer Science
[4] *University of New Mexico, Albuquerque, NM (USA)*
and [5] *Technical University of Madrid, Madrid (Spain).*

Abstract

Automatic cost analysis of programs has been traditionally concentrated on a reduced number of resources such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, bytes sent or received by an application, number of files left open, number of SMSs sent or received, number of accesses to a database, money spent, energy consumption, etc. We present a fully automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of *application programmer-definable* resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis proposed is independent of the particular resource. We also present some experimental results from a prototype implementation of the approach covering a significant set of interesting resources.

1 Introduction

The usefulness of analyses which can infer information about the costs of computations is widely recognized since such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [27,34,36,20,9,21,40] for functional languages, [38,8,19,42] for imperative languages, and [17,16,18,32] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [29]), resource-oriented specialization (e.g., [13,33]), or, more recently, certification of the resources used by mobile code (e.g., [14,6,12,5,22]). Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent rather than

[1] Email: navas@comp.nus.edu.sg
[2] Email: marioml@ices.utexas.edu
[3] Email: herme@fi.upm.es

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

(or, rather, in addition to) the predefined, more traditional costs such as steps, time, or memory. Regarding the object of certification, in the case of mobile code the certification and checking process is often performed at the bytecode level [28], since, in addition to other reasons of syntactic convenience, bytecode is what is most often available at the receiving (checker) end.

We propose a fully automated framework which infers upper bounds on the usage that a Java bytecode program makes of *application programmer-definable* resources. Examples of such programmer-definable resources are bytes sent or received by an application over a socket, number of files left open, number of SMSs sent or received, number of accesses to a database, number of licenses consumed, monetary units spent, energy consumed, disk space used, and of course, execution steps (or bytecode instructions), time, or memory. A key issue in approach is that resources are defined by programmers and by means of *annotations*. The annotations defining each resource must provide for some relevant user-selected elements corresponding to the bytecode program being analyzed (classes, methods, variables, etc.), a value that describes the cost of that element for that particular resource. These values can be constants or, more generally, functions of the input data sizes. The objective of our analysis is then to statically derive from these elementary costs an upper bound on the amount of those resources that the program as a whole (as well as individual blocks) will consume or provide.

Our approach builds on the work of [17,16] for logic programs, where cost functions are inferred by solving recurrence equations derived from the syntactic structure of the program. Most previous work deals only with concrete, traditional resources (e.g., execution steps, time, or memory). The analysis of [32] also allows program-level definition of resources, but it is designed for Prolog and works at the source code level, and thus is not directly applicable to Java bytecode due to particularities like virtual method invocation, unstructured control flow, assignment, the fact that statements are low-level bytecode instructions, the absence of backtracking (which has a significant impact on the method used in [32]), etc. Also, the presentation of [32] is descriptive in contrast to the concrete algorithm provided herein. In [2], a cost analysis is described that does deal with Java bytecode and is capable of deriving cost relations which are functions of input data sizes. The authors also presented in [3] an experimental evaluation of the approach. This approach is generic, in the same sense as, e.g., [16], in that both the conceptual framework and its implementation allow adaptation to different resources. However, this is done typically in the implementation. Our approach is interesting in that it allows the application programmer to define the resources through annotations directly in the Java source, and without changing the analyzer code or tables in any way. Also, without claiming it as any significant contribution of course, we provide for implementation convenience a somewhat more concrete, algorithmic presentation, in contrast to the more descriptive approach of previous work (including [17,16,32,2,3], etc.).

2 User-Defined Resources: Overview of the Approach

A *resource* is a fundamental component in our approach. A resource is a user-defined notion which associates a basic cost function with some user-selected elements (class,

```
import java.net.URLEncoder;
public class CellPhone {
 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
  if (smsPk != null) {
    String newSms = enc.format(smsPk.sms);
    stm.send(newSms);
    smsPk.next=sendSms(smsPk.next,enc,stm);
    smsPk.sms = newSms;
  }
  return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=size(s)")
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  @Cost({"cents","2*size(data)"})
  native void send(String data);
}
```

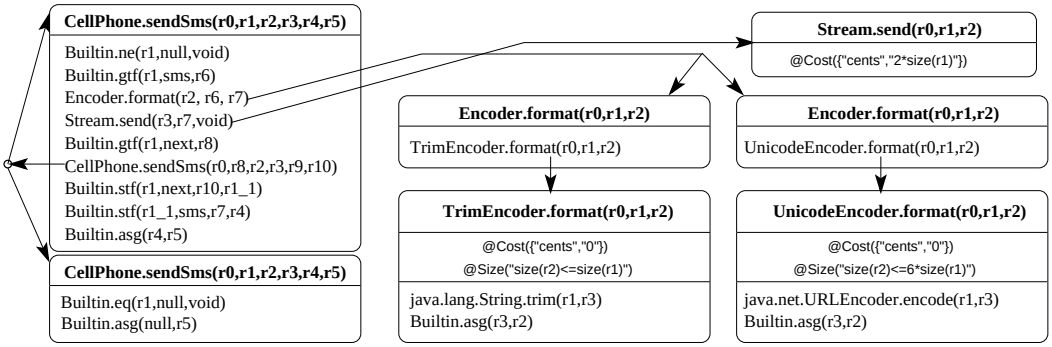


Fig. 1. Motivating example: Java source code and Control Flow Graph

method, statement) in the program. This is expressed by adding *Java annotations* to the code. The objective of the analysis is to approximate the usage that the program makes of the resource.

We start by illustrating the overall approach through a working example. The Java program in Fig. 1 emulates the process of sending text messages within a cell phone. This example is not intended to be realistic, but rather a small piece of code that illustrates a number of aspects of the approach. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The phone (class `CellPhone`) receives a list of packets (`SmsPacket`), each one containing a single SMS, encodes them (`Encoder`), and sends them through a stream (`Stream`). There are two types of encoding: `TrimEncoder`, which eliminates any leading and trailing white spaces, and `UnicodeEncoder`, which converts any special character into its Unicode($\backslash uxxxx$) equivalent. The length of the SMS which the cell phone ultimately sends through the stream depends on the size of the encoded message.

In the example, the resource is the cost in cents of a dollar for sending the list of text messages. We will assume for the sake of discussion that the carrier charges are proportional to the number of characters sent, and at 2 cents/character. This is reflected by the user in the method that is ultimately responsible for the communication (`Stream.send`), by adding the annotation `@Cost({"cents","2*size(data)"})`. Similarly, the formatting of an SMS made in

any implementation of `Encoder.format` is free, as indicated by the `@Cost-({"cents","0"})` annotation (the actual system allows defining overall cost defaults but we express them here explicitly). The analysis then processes these local resource usage expressions and uses them to infer a safe upper bound on the *total* (global) usage of the defined resources made by the program.

As illustrated by the example, these Java annotations allow defining the resources to be tracked (which is done by simply mentioning them in the annotations) and to provide cost functions for the built-in and external (library) blocks that are relevant to the particular resource (i.e., which affect the usage of such resource). They also allow defining data size relations among arguments and defining and declaring size measures. The resource usage expressions are defined using the following language (which we will call $\mathcal{L}$):

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid (\sum \mid \prod) \langle expr \rangle \\
& \mid & \langle expr \rangle^{\langle expr \rangle} \mid log_{num} \langle expr \rangle \mid -\langle expr \rangle \\
& \mid & \langle expr \rangle ! \mid \infty \mid \text{num} \\
& \mid & \text{size}([\langle measure \rangle,]\text{arg(r num)}) \\
\langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\
\langle measure \rangle & ::= & \textbf{int} \mid \textbf{ref} \mid \ldots
\end{array}
$$

We now summarize the fundamental steps of the analysis:

Step 1: Constructing the Control Flow Graph.

In the first step, the analysis translates the Java bytecode into an intermediate representation building a Control Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to our code example is also shown in Fig. 1.

The original `sendSms` method has been compiled into two block methods that share the same signature: class where declared, name (`CellPhone.sendSms`), and number and type of the formal parameters. The bottom-most box represents the base case, in which we return null, here represented as an assignment of `null` to the return variable r_5; the sibling corresponds to the recursive case. The virtual invocation of `format` has been transformed into a static call to a block method named `Encoder.format`. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in `TrimEncoder.format` and `UnicodeEncoder.format`. Note that the resource-related annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships.

The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. We assume that the size of (the contents of) a linked structure pointed to by a variable is the maximum number of pointers we need to traverse, starting at the variable, until `null` is found. The following equations are inferred by the analysis for the two `CellPhone.sendSms` block methods (with s_{r_i} we denote the size of input formal parameter position i, corresponding to variable r_i):

$$\begin{aligned}
\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) &\leq 0 \\
\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) &\leq 7 \times s_{r_1} - 6 + \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})
\end{aligned}$$

The size of the returned value r_5 is independent of the sizes of the input parameters *this*, *enc*, and *stm* (s_{r_0}, s_{r_2} and s_{r_3} respectively) but not of the size s_{r_1} of the list of text messages *smsPk* (r_1 in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. In the example in Fig. 1, the user indicates that the formatting in `UnicodeEncoder` results in strings that are at most six times longer than the ones received as input `@Size("size(ret)<=6*size(s)")`, while the trimming in `TrimEncoder` returns strings that are equal or shorter than the input (`@Size("size(ret)<=size(s)")`). In this case the equations provide implicitly the size measure (i.e., that the size of a string is its length). The equation system shown above is approximated by a recurrence solver included in our analysis in order to obtain the closed form solution $\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 3.5 \times s^2_{r_1} - 2.5 \times s_{r_1}$. This is a reasonable bound given that we have not specified a maximum size for each string.

Step 3: Resource Usage Analysis.

In this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps to infer a resource usage equation for each block method in the CFG (possibly simplifying such equations) and obtain closed form solutions (in general, approximated –upper bounds). Therefore, the objective of the resource analysis is to statically derive safe upper bounds on the amount of resources that each of the block methods in the CFG consumes or provides. The result given by our analysis for the monetary cost of sending the messages (`CellPhone.sendSms`) is

$$\begin{aligned}
Cost_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) &\leq 0 \\
Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) &\leq 12 \times s_{r_1} - 12 + Cost_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})
\end{aligned}$$

i.e., the cost is proportional to the size of the message list (`smsPk` in the source, r_1 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 6 \times s^2_{r_1} - 6 \times s_{r_1}$.

3 Intermediate program representation

Analysis of a Java bytecode program normally requires its translation into an intermediate representation that is easier to manipulate. In particular, our decompilation (assisted by the Soot [39] tool) involves elimination of stack variables, conversion to three-address statements, static single assignment (SSA) transformation, and generation of a Control Flow Graph (CFG) that is ultimately the subject of analysis. Note that in this representation loops are converted into recursive blocks. The decompilation process is an evolution of the work presented in [31], which has been successfully used as the basis for other (non resource-related) analyses [30]. Our ultimate objective is to support the full Java language but the current transforma-

tion has some limitations: it does not yet support reflection, threads, or runtime exceptions. The following grammar describes the intermediate representation; some of the elements in the tuples are named so we can refer to them as $node.\mathsf{name}$.

$$
\begin{array}{lcl}
CFG & ::= & Block^{+} \\
Block & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{fpars}{:}Id^{+},\mathsf{annot}{:}expr^{*},\mathsf{body}{:}Stmt^{*}) \\
Sig & ::= & (\mathsf{class}{:}Type,\mathsf{name}{:}Id,\mathsf{pars}{:}Type^{+}) \\
Stmt & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{apars}{:}(Id|Ct)^{+}) \\
Var & ::= & (\mathsf{name}{:}Id,\mathsf{type}{:}Type)
\end{array}
$$

The Control Flow Graph is composed of *block methods*. A block method is similar to a Java method, with some particularities: a) if the program flow reaches it, every statement in it will be executed, i.e, it contains no branching; b) its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types; c) it always includes as formal parameters the returned value *ret* and, unless it is static, the instance self-reference *this*; d) for every formal parameter (*input* formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the block method that contains its final version in the SSA transformation (*output* formal parameter); e) every statement in a block method is an invocation, including builtins (assignment `asg`, field dereference `gtf`, etc.), which are understood as block methods of the class `Builtin`.

As mentioned before, there is no branching within a block method. Instead, each conditional `if` *cond* $stmt_1$ `else` $stmt_2$ in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of decompiling *cond*, while we attach $\overline{cond}$ to its sibling. A similar approach is used in virtual invocations, for which we introduce as many block methods in the graph as possible receivers of the call were in the original program. A set of block methods with the same signature *sig* can be retrieved by the function $getBlocks(CFG, sig)$.

User specifications are written using the annotation system introduced in Java 1.5 which, unlike JML specifications, has the very useful characteristic of being preserved in the bytecode. Annotations are carried over to our CFG representation, as can be seen in Fig. 1.

Example 1 We now focus our attention on the two block methods in Fig. 1, which are the result of (de)compiling the `CellPhone.sendSms` method. The input formal parameters r_0, r_1, r_2, r_3 correspond to $this, smsPk, enc$, and stm, respectively. In the case of r_1, the contents of its fields *next* and *sms* are altered by invoking the `stf` and accessed by invoking the `gtf` (abbreviation for `setfield` and `getfield`, respectively) builtin block methods. The output formal parameter r_4 contains the final state of r_1 after those modifications. The value returned by the block methods is contained in r_5. Space reasons prevent us from showing any type information in the CFG in Fig 1. In the case of `Encoder.format`, for example, we say that there are two blocks with the same signature because they are both defined in class `Encoder`, have the same name (`format`) and the same list of types of formal parameters {`Encoder,String,String`}.

```
resourceAnalysis(CFG, res) {
 CFG←classAnalysis(CFG)
 Aliases←aliasAnalysis(CFG)
 mt←initialize(CFG)
 dg←dataDependencyAnalysis(CFG, Aliases ,mt)
 for (SCC:SCCs)
   //in reverse topological order
   mt←genSizeEqs(SCC,mt,CFG,dg)
   mt←genResourceUsageEqs(SCC, res ,mt,CFG)
 return mt
}
```

```
normalize(Eqs) {
 for (size relation p ≤ e1:Eqs)
   do
     if (expression s appears in e1
         and s ≤ e2 ∈Eqs)
       replace ocurrences of s in e1 with e2
   while there is change
 return Eqs
}
```

Fig. 2. Generic resource analysis algorithm and normalization.

4 The resource usage analysis framework

We now describe our framework for inferring upper bounds on the usage that the Java bytecode program makes of a set of resources defined by the application programmer, as described before. The algorithm in Fig 2 takes as input a Control Flow Graph in the format described in the previous section, including the user annotations that assign elementary costs to certain graph elements for a particular resource. The user also indicates the set of resources to be tracked by the analysis. Without loss of generality we assume for conciseness in our presentation a single resource.

A preliminary step in our approach is a class hierarchy analysis [15,30], aimed at simplifying the CFG and therefore improving overall precision. More importantly, we also require the existence of an alias analysis [35,26,11], whose results are used by a third phase (described below) in which data dependencies between variables in the CFG are inferred. The next step is the decomposition of the CFG into its strongly-connected components. After these steps, two different analyses are run separately on each strongly connected component: a) the size analysis, which estimates parameter size relationships for each statement and output formal parameters as a function of the input formal parameter sizes (Sec. 4.1); and b) the actual resource analysis, which computes the resource usage of each block method in terms also of the input data sizes (Sec. 4.2). Each phase is dependent on the previous one.

The *data dependency analysis* is a dataflow analysis that yields *position dependency graph*s for the block methods within a strongly connected component. Each graph $G = (V, E)$ represents data dependencies between positions corresponding to statements in the same block method, including its formal parameters. Vertexes in V denote positions, and edges $(s_1, s_2) \in E$ denote that s_2 is dependent on s_1 (s_1 is a *predecessor* of s_2). We will assume a `predec` function that takes a position dependency graph, a statement, and a parameter position and returns its nearest predecessor in the graph. Fig. 3 shows the position dependency graph of the `TrimEncoder.format` block method.

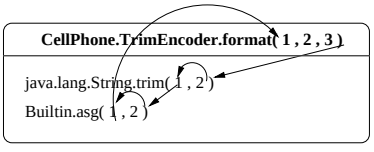


Fig. 3:

4.1 Size analysis

We now show our algorithm for estimating parameter size relations based on the data dependency analysis, inspired by the original ideas of [17,16]. Our goal is

to represent input and output size relationships for each statement as a function defined in terms of the formal parameter sizes. Unless otherwise stated, whenever we refer to a parameter we mean its position.

The size of an input is defined in terms of measures. By *measure* we mean a function that, given a data structure, returns a number. Our method is parametric on measures, which can be defined by the user and attached via annotations to parameters or classes. For concreteness, we have defined herein two measures, int for integer variables, and the *longest path-length* [37,2] ref for reference variables. The longest path-length of a variable is the cardinality of the longest chain of pointers than can be followed from it. More complex measures can be defined to handle other data types such as cyclic structures, arrays, etc. The set of measures will be denoted by $\mathcal{M}$.

The size analysis algorithm is given in pseudo-code in Fig. 4; its main steps are:

(i) Assign an upper bound to the size of every parameter position of all statements, including formal parameters, for all the block methods with the same signature (genSigSize).

(ii) For a given signature, take the set of size inequations returned by (i) and rename each size relation in terms of the sizes of input formal parameters (normalize).

(iii) Repeat the first step for every signature in the same strongly-connected component (genSizeEqs).

(iv) Simplify size relationships by resolving mutually recursive functions, and find closed form solutions for the output formal parameters (genSizeEqs).

Intermediate results are cached in a memo table mt, which for every parameter position stores measures, sizes, and resource usage expressions defined in the $\mathcal{L}$ language.

The size of the parameter at position i in statement $stmt$, under measure m, is referred to as $\texttt{size}(m, stmt, i)$. We consider a parameter position to be *input* if it is bound to some data when the statement is invoked. Otherwise, it is considered an *output parameter position*. In the case of input parameter and output formal parameter positions, an upper bound on that size is returned by getSize (Fig. 4). The upper bound can be a concrete value when there is a constant in the referred position, i.e., when the `val` function returns a non-infinite value:

Definition 4.1 The concrete size value for a parameter position under a particular measure is returned by $\texttt{val} : \mathcal{M} \times \mathcal{S}tmt \times \mathbb{N} \rightarrow \mathcal{L}$, which evaluates the *syntactic* content of the actual parameter in that position:

$$\texttt{val}(m, stmt, i) = \begin{cases} n & \textit{if } stmt.\textsf{apars}_i \textit{ is an integer } n \textit{ and } m{=}\textbf{\textit{int}} \\ 0 & \textit{if } stmt.\textsf{apars}_i \textit{ is } \texttt{null} \textit{ and } m{=}\textbf{\textit{ref}} \\ \infty & \textit{otherwise} \end{cases}$$

If the content of that input parameter position is a variable, the algorithm searches the data dependency graph for its immediate predecessor. Since the intermediate representation is in SSA form, the only possible scenarios are that either

```
genSizeEqs(SCC,mt,CFG,dg) {
 Eqs← ∅^|SCC|
 for (sig: SCC)
   Eqs[sig]←genSigSize(sig,mt,SCC,CFG,dg)
 Sols←recEqsSolver(simplifyEqs(Eqs))
 for (sig:SCC)
   insert(mt,size,sig,Sols[sig])
 return mt
}

genSigSize(sig,mt,SCC,CFG,dg) {
 Eqs← ∅
 BMs←getBlocks(CFG,sig)
 for (bm:BMs)
   Eqs←Eqs ∪ genBlockSize(bm,mt,SCC,dg)
 return normalize(Eqs)
}

genBlockSize(bm,mt,SCC,dg) {
 Eqs← ∅
 for (stmt:bm.body)
   I←stmt input parameter positions
   Eqs←Eqs ∪ genInSize(stmt,I,mt,dg)
   Eqs←Eqs ∪ genOutSize(stmt,mt,SCC)
 K← bm output formal parameter positions
 Eqs←Eqs ∪ genInSize(bm,K,mt,dg)
 return Eqs
}

genInSize(elem,Pos,mt,dg) {
 Eqs← ∅
 for (pos:Pos)
   m←lookup(mt,measure,elem.sig,pos)
   s←getSize(m,elem.id,pos,dg)
   Eqs←Eqs ∪ {size(m,elem.id,pos)≤s}
 return Eqs
}
```

```
genOutSize(stmt,mt,SCC) {
 {i_1,...,i_l} ← stmt input positions
 sig←stmt.sig
 {m_{i_1},...,m_{i_l}} ←{lookup(mt,measure,sig,i_1),...,
                         lookup(mt,measure,sig,i_l)}
 {s_{i_1},...,s_{i_l}} ← {size(m_{i_1},stmt.id,i_1),...,
                          size(m_{i_l},stmt.id,i_l)
 Eqs← ∅
 O← stmt output parameter positions
 for (o:O)
   m_o ←lookup(mt,measure,sig,o)
   if (sig∉SCC)
     Size_user ← A^o_sig(s_{i_1},...,s_{i_l})
     Size_alg' ←max(lookup(mt,size,sig,o))
     Size_alg ←Size_alg'(s_{i_1},...,s_{i_l})
     Size_o ←min(Size_user,Size_alg)
   else
     Size_o ← Size^o_sig(m_o,s_{i_1},...,s_{i_l})
   Eqs←Eqs ∪ {size(m_o,stmt.id,o)≤ Size_o}
 return Eqs
}

getSize(m,id,pos,dg) {
 result←val(m,id,i)
 if (result≠∞)
   return result
 else
   if (∃(elem,pos_p)∈ predec(dg,id,pos))
     m_p ←lookup(mt,measure,elem.sig,pos_p)
     if (m=m_p)
       return size(m_p,elem.id,pos_p)
 return ∞
}
```

Fig. 4. The size analysis algorithm

there is a unique predecessor whose size is assigned to that input parameter position, or there is none, causing the input parameter size to be unbounded (∞).

Consider now an output parameter position within a block method, case covered in genOutSize (Fig. 4). If the output parameter position corresponds to a non-recursive invoke statement, either a size relationship function has already been computed recursively (since the analysis traverses each strongly-connected component in reverse topological order), or it is provided by the user through size annotations. In the first case, the size function of the output parameter position can be retrieved from the memo table by using the lookup operation, taking the maximum in case of several size relationship functions, and then passing the input parameter size relationships to this function to evaluate it. In the second scenario, the size function of the output parameter position is provided by the user through size annotations, denoted by the $\mathcal{A}$ function in the algorithm. In both cases, it will able to return an explicit size relation function.

Example 2 We have already shown in the `CellPhone` example how a class can be annotated. The `Builtin` class includes the assignment method `asg`, annotated as follows:

```
public class Builtin {
  @Size{"size(ret)<=size(o)"}
  public static native Object asg(Object o);
```

```
    // ...rest of annotated builtins
}
```

which results in equation $\mathcal{A}^1_{\mathtt{asg}}(\mathtt{ref}, \mathtt{size}(\mathtt{ref}, asg, 0)) \leq \mathtt{size}(\mathtt{ref}, asg, 0)$.

If the output parameter position corresponds to a recursive invoke statement, the size relationships between the output and input parameters are built as a symbolic size function. Since the input parameter size relations have already been computed, we can establish each output parameter position size as a function described in terms of the input parameter sizes.

At this point, the algorithm has defined size relations for all parameter positions within a block method.

However, those relations are either constants or given in terms of the immediate predecessor in the dependency graph. The algorithm rewrites the equation system such that we obtain an equivalent system in which only formal parameter positions are involved. This process, called *normalization*, is shown in Fig. 2

After normalization, the analysis repeats the same process for all block methods in the same strongly-connected component (SCC). Once every component has been processed, the analysis further simplifies the equations in order to resolve mutually recursive calls among block methods within the same SCC in the simplifyEqs procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver (recEqsSolver, called from genSizeEqs) in order to obtain approximated upper-bound closed forms. The interesting subject of how the equations are solved is beyond the scope of this paper (see, e.g., [41]). Our implementation does provide a simple built-in solver (an evolution of the solver of the Caslog system [16]) which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients, divide and conquer recurrence equations, etc. However, it also includes an interface to the Parma Polyhedra Library [7] (and previously to other tools such as Mathematica, Matlab, etc.). Work is also under way to interface with the quite interesting solver of [1].

Example 3 We now illustrate the definitions and algorithm with an example of how the size relations are inferred for the two `CellPhone.sendSms` block methods (Fig. 1), using the ref measure for reference variables. We will refer to the k-th occurrence of a statement $stmt$ in a block method as $stmt_k$, and denote `CellPhone.sendSms`, `Encoder.format`, and `Stream.send` by `sendSms`, `format`, and `send` respectively. Finally, as mentioned before, we refer to the size of the input formal parameter position i, corresponding to variable r_i, as s_{r_i}.

The main steps in the process are listed in Fig. 5. The first block of rows contains the most relevant size parameter relationship equations for the recursive block method, while the second block of rows corresponds to the base case. These size parameter relationship equations are constructed by the analysis by first following the algorithm in Fig. 4, and then normalizing them (expressing them in terms of the input formal parameter sizes s_{r_i}). Also, in the first block of rows we observe that the algorithm has returned $6 \times \mathtt{size}(\mathtt{ref}, format, 1)$ as upper bound for the size of the formatted string, $\mathsf{max}(\mathsf{lookup}(mt, \mathtt{size}, format, 2))$. The result is

Size parameter relationship equations (normalized)

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, ne, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, ne, 1) & \leq \texttt{val}(\texttt{ref}, ne, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, gtf_1, 0) & \leq \texttt{size}(\texttt{ref}, ne, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_1, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_1, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 1) & \leq \texttt{size}(\texttt{ref}, gtf_1, 2) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 2) & \leq \texttt{max}(\texttt{lookup}(mt, \texttt{size}, format, 2))(\texttt{size}(\texttt{ref}, format, 2)) \\
& \leq \texttt{max}(s_{r1}, 6 \times s_{r1})(s_{r_1} - 1) \\
& \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, send, 1) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, gtf_2, 0) & \leq \texttt{size}(\texttt{ref}, gtf_1, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_2, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_2, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, sendSms, 5) & \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, \_, \texttt{size}(\texttt{ref}, sendSms, 1), \_, \_) \\
& \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 0) & \leq \texttt{size}(\texttt{ref}, gtf_2, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, stf_1, 2) & \leq \texttt{size}(\texttt{ref}, sendSms, 5) \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_1, 0), \_, \texttt{size}(\texttt{ref}, stf_1, 2)) \\
& \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 0) & \leq \texttt{size}(\texttt{ref}, stf_1, 3) \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 2) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, stf_2, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_2, 0), \_, \texttt{size}(\texttt{ref}, stf_2, 2)) \\
& \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{size}(\texttt{ref}, stf_2, 3) \\
& \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \\
& \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})
\end{array}
$$

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, eq, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, eq, 1) & \leq \texttt{val}(\texttt{ref}, eq, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{val}(\texttt{ref}, asg, 0) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq 0
\end{array}
$$

Output parameter size functions for builtins (provided through annotations)

$$
\begin{array}{rl}
\mathcal{A}^2_{\texttt{gtf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf, 0), \_) & \leq \texttt{size}(\texttt{ref}, gtf, 0) - 1 \\
\mathcal{A}^1_{\texttt{asg}}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) & \leq \texttt{size}(\texttt{ref}, asg, 0) \\
\mathcal{A}^3_{\texttt{stf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf, 0), \_, \texttt{size}(\texttt{ref}, stf, 2)) & \leq \texttt{size}(\texttt{ref}, stf, 0) + \texttt{size}(\texttt{ref}, stf, 2)
\end{array}
$$

Simplified size equations and closed form solution

$$
\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}
$$

$$
\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 3.5 \times s^2_{r1} - 2.5 \times s_{r1}
$$

Fig. 5. Size equations example

the maximum of the two upper bounds given by the user for the two implementations for `Encoder.format` since `TrimEncoder.format` eliminates any leading and trailing white spaces (thus the output is at most as bigger as the input), whereas `UnicodeEncoder.format` converts any special character into its Unicode equivalent (thus the output is at most six times the size of the input), a safe upper bound for the output parameter position size is given by the second annotation.

In the particular case of builtins and methods for which we do not have the code, size relationships are not computed but rather taken from the user `@Size` annotations. These functions are illustrated in the third block of rows. Finally, in the fourth block of rows we show the recurrence equations built for the output

```
genResourceUsageEqs(SCC, res, mt, CFG) {
  Eqs ← ∅^|SCC|
  for (sig : SCC)
    Eqs[sig] ← genSigRU(sig, res, mt, SCC, CFG)
  Sols ← recEqsSolver(simplifyEqs(Eqs))
  for (sig : SCC)
    insert(mt, cost, max(Sols[sig]))
  return mt
}

genSigRU(sig, res, mt, SCC, CFG) {
  Eqs ← ∅
  BMs ← getBlocks(CFG, sig)
  for (bm : BMs)
    body ← bm.body
    Cost_body ← 0
    for (stmt : body)
      Cost_stmt ← genStmtRU(stmt, res, mt, SCC)
      Cost_body ← Cost_body + Cost_stmt
    Cost_bm ← genBlockRU(bm, res, mt)
    Eqs ← Eqs ∪ {Cost_bm ≤ Cost_body}
}

genStmtRU(stmt, res, mt, SCC) {
  {i_1, ..., i_k} ← stmt input parameter positions
  {s_i1, ..., s_ik} ←
      {max(lookup(mt, size, stmt.sig, i_1)), ...,
       max(lookup(mt, size, stmt.sig, i_k))}
  if (stmt.sig ∉ SCC)
    Cost_user ← A_stmt.sig(res, s_i1, ..., s_ik)
    Cost_alg' ← lookup(mt, cost, res, stmt.sig)
    Cost_alg ← Cost_alg'(s_i1, ..., s_ik)
    return min(Cost_alg, Cost_user)
  else return Cost(stmt.sig, res, s_i1, ..., s_ik)
}

genBlockRU(bm, res, mt) {
  {i_1, ..., i_l} ← bm input formal parameter positions
  {s_i1, ..., s_il} ←
      {lookup(mt, size, bm.id, i_1), ...,
       lookup(mt, size, bm.id, i_l)
  return Cost(bm.id, res, s_i1, ..., s_il)
}
```

Fig. 6. The resource usage analysis algorithm

parameter sizes in the block method and in the final row the closed form solution obtained.

4.2 Resource usage analysis

The core of our framework is the resource usage analysis, whose pseudo code is shown in Fig 6. It takes a strongly-connected component of the CFG, including the set of annotations which provide the application programmer-provided resources and cost functions, and calculates a resource usage function which is an upper bound on the usage made by the program of those resources. The algorithm manipulates the same memo table described in Sec. 4.1 in order to avoid recomputations and access the size relationships already inferred.

The algorithm is structured in a very similar way to the size analysis (which also allows us to draw from it to keep the explanation within space limits): for each element of the strongly-connected component the algorithm will construct an equation for each block method that shares the same signature representing the resource usage of that block. To do this, the algorithm will visit each invoke statement. There are three possible scenarios, covered by the `genStmtRU` function. If the signatures of caller and callee(s) belong to the same strongly-connected component, we are analyzing a recursive invoke statement. Then, we add to the body resource usage a symbolic resource usage function, in an analogous fashion to the case of output parameters in recursive invocations during the size analysis.

The other scenarios occur when the invoke statement is non-recursive. Either a resource usage function $Cost_{alg}$ for the callee has been previously computed, or there is a user annotation $Cost_{usr}$ that matches the given signature, or both. In the latter case, the minimum between these two functions is chosen (i.e., the most precise safe upper bound assigned by the analysis to the resource usage of the non-recursive invoke statement) or a warning is issued.

Example 4 The call (sixth statement) in the upper-most `CellPhone.sendSms`

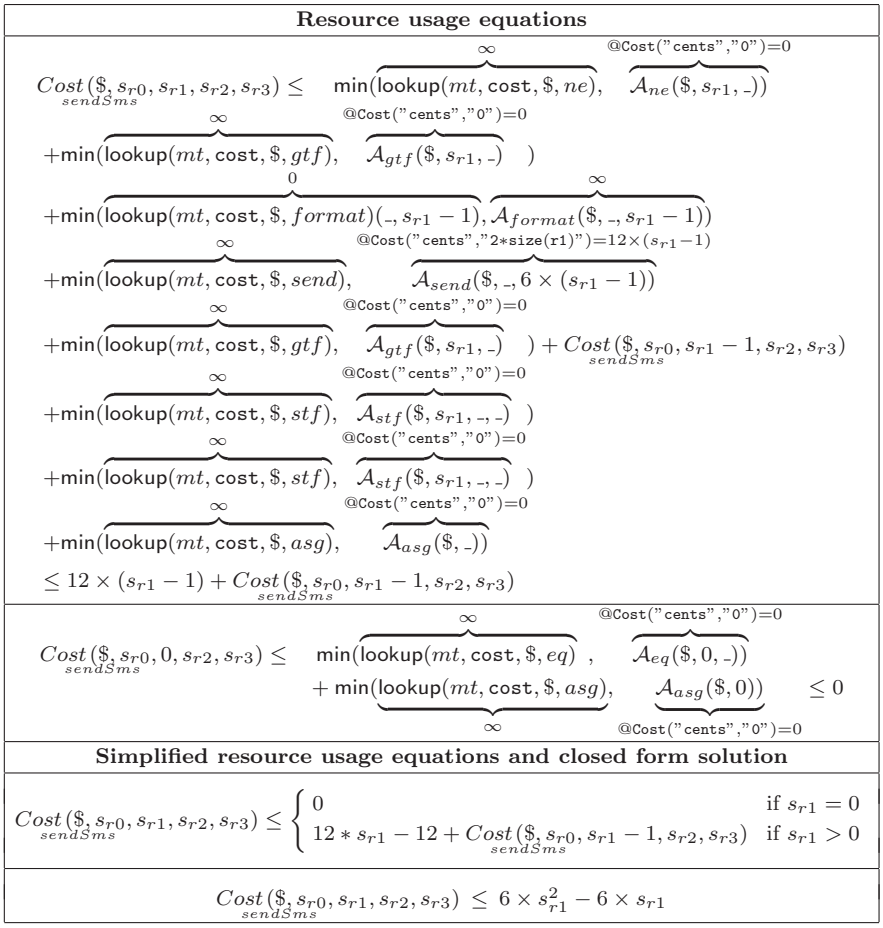


Fig. 7. Resource equations example

block method matches the signature of the block method itself and thus it is recursive. The first four parameter positions are of input type. The upper-bound expression returned by genStmtRU is $Cost_{sendSms}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$. Note that the input size relationships were already normalized during the size analysis. Now consider the invocation of `Stream.send`. The resource usage expression for the statement is defined by the function $\mathcal{A}_{send}(\$, \_, 6 \times (s_{r1}-1))$ since the input parameter at position one is at most six times the size of the second input formal parameter, as calculated by the size analysis in Fig. 5. Note also that there is a resource annotation `@Cost({"cents","2*size(r1)"})` attached to the block method describing the behavior of $\mathcal{A}_{send}$ and yielding the expression $Cost_{user} = 12 \times (s_{r1}-1)$. On the other hand, the absence of any callee code to analyze –the original method is native– results in $Cost_{alg} = \infty$. Then, the upper bound obtained by the analysis for the statement is $\min(Cost_{alg}, Cost_{user}) = Cost_{user}$.

At this point, the analysis has built a resource usage function (denoted by $Cost_{body}$) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form $Cost_{block} \leq Cost_{body}$ where $Cost_{block}$ is again a symbolic resource usage function built by replacing each input formal parameter position with its size relations in that block method. These resource usage equations are simplified by calling simplifyEqs and, finally, they are solved calling recEqsSolver, both already defined in Sec. 4.1. This process yields an (in gen-

eral, approximate, but always safe) closed form upper bound on the resource usage of the block methods in each strongly-connected component. Note that given a signature the analysis constructs a closed form solution for every block method that shares that signature. These solutions approximate the resource usage consumed in or provided by each block method. In order to compute the total resource usage of the signature the analysis returns the maximum of these solutions yielding a safe global upper bound.

Example 5 The resource usage equations generated by our algorithm for the two `sendSms` block methods and the "$" resource (monetary cost of sending the SMSs) are listed in Fig. 7. The computation is partially based on the size relations in Fig. 5. The resource usage of each block method is calculated by building an equation such that the left part is a symbolic function constructed by replacing each parameter position with its size (i.e., $Cost_{sendSms}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3})$ and $Cost_{sendSms}(\$, s_{r0}, 0, s_{r2}, s_{r3})$), and the rest of the equation consists of adding the resource usage of the invoke statements in the block method. These are calculated by computing the minimum between the resource usage function inferred by the analysis and the function provided by the user. The equations corresponding to the recursive and non-recursive block methods are in the first and second row, respectively. They can be simplified (third row) and expressed in closed form (fourth row), obtaining a final upper bound for the charge incurred by sending the list of text messages of $6 \times s_{r1}^2 - 6 \times s_{r1}$.

5 Experimental results

We have completed an implementation of our framework (in Ciao [10], using components from CiaoPP [23], and with help from the Soot tool [39], as mentioned before), and tested it for a representative set of benchmarks and resources. Our experimental results are summarized in Table 1. Column **Program** provides the name of the main class to be analyzed. Column **Resource(s)** shows the resource(s) defined and tracked. Column t_s shows the time (in milliseconds) required by the size analysis to construct the size relations (including the data dependency analysis and class hierarchy analysis) and obtain the closed form. Column t_r lists the time taken to build the resource usage expressions for all method blocks and obtain their closed form solutions. t provides the total times for the whole analysis process. Finally, column **Resource Usage Func.** provides the upper bound functions inferred for the resource usage. For space reasons, we only show the most important (asymptotic) component of these functions, but the analysis yields concrete functions with constants.

Regarding the benchmarks we have covered a reasonable set of data-structures used in object-oriented programming and also standard Java libraries used in real applications. We have also covered an ample set of application-dependent resources which we believe can be relevant in those applications. In particular, not only have we represented high-level resources such as cost of SMS, bytes received (including a coarse measure of bandwidth, as a ratio of data per program step), and files left open, but also other low-level (i.e., bytecode level) resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program *Files* (a fragment characteristic of operating system kernel

Program	Resource(s)	t_s	t_r	t	Resource Usage Func.	
BST	Heap usage	250	22	367	$O(2^n)$	$n \equiv$ tree depth
CellPhone	SMS monetary cost	271	17	386	$O(n^2)$	$n \equiv$ packets length
Client	Bytes received and	391	38	527	$O(n)$	$n \equiv$ stream length
	bandwidth required				$O(1)$	—
Dhrystone	Energy consumption	602	47	759	$O(n)$	$n \equiv$ int value
Divbytwo	Stack usage	142	13	219	$O(log_2(n))$	$n \equiv$ int value
Files	Files left open and	508	53	649	$O(n)$	$n \equiv$ number of files
	Data stored				$O(n \times m)$	$m \equiv$ stream length
Join	DB accesses	334	19	460	$O(n \times m)$	$n, m \equiv$ records in tables
Screen	Screen width	388	38	536	$O(n)$	$n \equiv$ stream length

Table 1
Times of different phases of the resource analysis and resource usage functions.

code) we kept track of the number of file descriptors left open. The data inferred for this resource can be clearly useful, e.g., for debugging: the resource usage function inferred in this case ($O(n)$) denotes that the programmer did not close $O(n)$ file descriptors previously opened. In program *Join* (a database transaction which carries out accesses to different tables) we decided to measure the number of accesses to such external tables. This information can be used, e.g., for resource-oriented specialization in order to perform optimized checkpoints in transactional systems. The rest of the benchmarks include other definitions of resources which are also typically useful for verifying application-specific properties: *BST* (a generic binary search tree, used in [4] where a heap space analysis for Java bytecode is presented), *CellPhone* (extended version of program in Figure 1), *Client* (a socket-based client application), *Dhrystone* (a modified version of a program from [25] where a general framework is defined for estimating the energy consumption of embedded JVM applications; the complete table with the energy consumption costs that we used can be found there), *DivByTwo* (a simple arithmetic operation), and *Screen* (a MIDP application for a cellphone, where the analysis is used to make sure that message lines do not exceed the phone screen width). The benchmarks also cover a good range of complexity functions ($O(1), O(log(n), O(n), O(n^2) \ldots, O(2^n), \ldots$) and different types of structural recursion such as simple, indirect, and mutual.

6 Conclusions

We have presented a fully-automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. Our analysis derives a vector of functions, one for each defined resource. Each of these functions returns, for each given set of input data sizes, an upper bound on the usage that the whole program (and each individual method) make of the corresponding resource. Our approach allows the application programmer to define the resources to be tracked by writing simple resource descriptions via source-level annotations, The current results suggest that the proposed analysis can obtain non-trivial bounds on a wide range of interesting resources in reasonable time. Our approach allows using the annotations also for a number of other purposes such as stating the resource usage of external methods, which is instrumental in allowing modular composition and thus scalability. In addition, our annotations allow stating the resource usage of any method for which the automatic analysis infers a value that is not accurate enough to prevent inaccuracies in the automatic

inference from propagating. Annotations are also used by the size and resource usage analysis to express their output. Finally, the annotation language can also be used to state specifications related to resource usage, which can then be proved or disproved based on the results of analysis following, e.g., the scheme of [24,5,22] thus finding resource bugs or verifying the resource usage of the program.

Acknowledgments: This work was funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM, EU projects FP6 FET IST-15905 *MOBIUS*, IST-215483 *SCUBE*, and 06042-ESPASS, Ministry of Science projects TIN-2008-05624 *DOVES*, TIN2005-09207-C03 *MERIT-COMVERS*, Ministry of Industry project FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*. Thanks also to Pedro López García for comments on drafts of this paper.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*, LNCS 5079, pages 221–237, 2008.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 67–83. Elsevier - North Holland, July 2007.

[4] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[5] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[6] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.

[7] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[8] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing*, Apr. 2002.

[9] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[11] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC*, pages 234–250, 1994.

[12] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.

[13] S.J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[14] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[15] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*, pages 77–101, 1995.

[16] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.

[17] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.

[18] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.

[19] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proc. of DDECS*. IEEE Computer Society, 2006.

[20] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.

[21] B. Grobauer. Cost recurrences for DML programs. In *Int'l. Conf. on Functional Programming*, pages 253–264, 2001.

[22] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.

[23] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.

[24] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[25] Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

[26] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, 1992.

[27] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *TOPLAS*, 10(2), 1988.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[29] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.

[30] M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[31] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

[32] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, LNCS, 2007.

[33] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of PPDP'06*, pages 261–271. ACM Press, 2006.

[34] M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[35] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented*, pages 43–55, 2001.

[36] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[37] F. Spoto, P.M. Hill, and E. Payet. Path-length analysis of object-oriented programs. In *EAAI'06*, ENTCS. Elsevier, 2006.

[38] Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour*, 2004.

[39] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[40] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

[41] H. S. Wilf. *Algorithms and Complexity*. A.K. Peters Ltd, 2002.

[42] R. Wilhelm. Timing analysis and timing predictability. In *Proc. FMCO*, LNCS. Springer-Verlag, 2004.