Project N^{o}: **FP6-015905**

Project Acronym: **MOBIUS**

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

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D4.2

Report on certificate format and certificate generation

Due date of deliverable: 2007-09-01 (T0+24)

Actual submission date: 2007-10-05

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

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

Revision 8093 — Final

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)	

Contributions

Site	**Contributed to Chapter**
INRIA	2, 3
IoC	3
UPM	3
WU	2
LMU	1, 4

Executive Summary:
Certificates

This document describes the results achieved in Task 4.2 (Certificates) of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

The deliverable presents three notions of certificates, which represent different points in the design space determined by factors such as ease of certificate generation and verification, certificate size, and level of security and trust.

The first notion, foundational certificates, rests upon the formalised infrastructure that has been implemented in WP3 in the theorem prover Coq. Here, certificates refer to the formal basis of the MOBIUS project, namely the Bicolano operational model of the JVM and the MOBIUS base logic. Efficiency of consumer-side certificate checking is increased by the use of reflection.

Complementing foundational certificates, we also present two further notions of certificates. Compared to the former, these formats provide more flexibility and efficiency, at the price of being more specific for a previously agreed-upon program analysis. The first notion consists of fixed points in the setting of abstraction-carrying code, while the second notion reduces the required size of certificates by proving a mechanism to structure proofs in sequent calculus through the use of lemmas.

In addition, we describe a generic mechanism for breaking down certificates into small portions in such a way that at any point in time only one such portion needs to be held in memory during verification thus reducing the memory requirements at the consumer's side.

Finally, we have explored the generation of certicates using the Abstraction Carrying Code (ACC) model. In particular, we have developed an automatic cost analyzer which, according to the ACC model, can be used to generate resource-related certicates and which allows different consumers to impose different safety policies depending on the particular resources of interest (e.g., the number of executed bytecode instructions, the memory consumption, etc.). This allows having expressive and practical resource-related safety policies.

Contents

Chapter 1

Introduction

This deliverable presents three notions of certificates developed within MOBIUS to serve as the basis of the proof-carrying code infrastructure being developed.

Let us briefly recall the role of certificates within the project. We want to allow code from untrusted, possibly even malicious providers to be run securely with respect to particular policies related to resource usage and information flow as detailed in D1.2.

Rather than somehow analysing the received program on the recipients' side, which may be very difficult and often impossible, we require that the code provider produce a certificate that allows the code recipient to become convinced about the security of the shipped program "beyond any reasonable doubt" with moderate computational effort on his side.

There are several competing quality criteria for such certificates which no single notion can attain simultaneously in the best possible way. Hence, at least at present, several notions of certificates must be made available and selected according to the weighting of the criteria in the particular application at hand.

The quality criteria are the following: degree of security, size of trusted computing base, ease of generation, ease of checking, size of certificates, flexibility and adaptability.

Chapter 2 presents a format for foundational certificates in the form of Coq proof objects that establish security properties formulated in terms of the Bicolano formalisation of the operational semantics of the JVM (D3.1)

Such foundational certificates can be automatically checked by the Coq proof system; their security is contingent on the correctness of the Bicolano formalisation, the correctness of the Coq kernel, and the correct formulation of security policies in Coq. No application-specific or policy-specific infrastructure needs to be be trusted. It is also described how such foundational certificates can be efficiently obtained from type inference and efficiently checked using reflection, a computational mechanism built into the Coq prover. These foundational certificates achieve the highest possible level of security and are in immune against mistakes in the implementation of the infrastructure and to a large extent against malicious code producers. Efficiency remains nevertheless reasonable due to the use of reflection which is described in detail in Section 2.1. However, foundational certificates require that a type system or program analysis be completely formalised in Coq inclusive of the entire proof of soundness. While this can be and has been done successfully for nontrivial and useful examples, it is a considerable burden that hampers the flexibility of this method.

A related notion of certificate—proof scripts using the MOBIUS base logic—has already been described in Deliverables 3.1 and 2.3 and is therefore not contained in this document. Here the type system or program analysis employed for certificate generation is implemented outside Coq but in such a way that a Coq proof script is generated automatically. This achieves the same degree of security as the reflective method; is in some cases easier to implement, but on the other hand, it is less robust against changes of the type system at hand. For the code recipient both methods look the same: he receives a proof script in Coq that establishes the desired security property with respect to Bicolano.

Two alternative notions of certificates which achieve a higher degree of efficiency are described in Chapter 3. These notions of certificates do not employ the Coq / Bicolano infrastructure and thus achieve higher efficiency both in terms of generation and of checking but at the price of an application-specific trusted computing base. At the present stage these notions of certificates are generic in nature and have not been fully adapted to Java Bytecode.

Reduced Certificates in Abstraction-Carrying Code (Section 3.1) extend the successful approach of Java's Bytecode Verification to advanced type systems and program analyses: a certificate is essentially given by type annotations, resp. fixpoints for loops. A full type derivation can then be reconstructed at little expense at the recipients side. This requires that the code producer and recipient have agreed upon a type system or program analysis beforehand. When applicable this approach has the advantage of being easier to implement than the foundational one since it is not necessary to formalise the analysis or type system within Coq. On the other hand, the type system or analyses itself must be trusted by the consumer as well as its implementation. Only the results of the fixpoint engine are verified subsequently at the consumer's side.

Tables of Lemmas described in Section 3.3 are based on the idea that a formal proof in sequent calculus can be reconstructed from the lemmas employed therein plus some administrative information. In this way, the size of a certificate based on proofs in sequent calculus or indeed any other deductive proof system can be considerably reduced.

A further notion of certificate based on logical systems is described in D3.2 (*proof-transforming compiler*). There certificates take the form of proofs of verification conditions generated from a successful run of an extended static checker such as ESC Java. These certificates can be, in principle, integrated with Bicolano and the MOBIUS base logic: they will then achieve the same degree of security as foundational certificates.

Section 3.2 presents a generic way to reduce the memory footprint of certificate checking independent of the particular format used, by replacing one-off checking of monolithic certificates with a dialogue between code producer and code consumer: during such a dialogue, only a small portion of the certificate must be held in the consumer's memory. It thus complements both foundational and flexible certificates and has the potential to facilitate the implementation of on-device checking.

The techniques to generate flexible certificates in the context of *Abstraction-Carrying Code* (ACC) are described in Section 3.4, where the static analyzer COSTA is used to generate flexible resource-related (i.e., *cost and termination) certificates* for Java bytecode. In adittion, we present some techniques used to improve the efficiency and the accuracy of COSTA with the aim of having more practical and more expressive safety policies and certificates.

Chapter 2

Foundational (Coq) Certificates

The verification infrastructure on the consumer side is the central element in the MOBIUS security architecture, and it is therefore of utmost importance to achieve the highest guarantees that its design and implementation are correct. However, providing a correct implementation of the MOBIUS verification infrastructure is a significant challenge, because it relies on advanced type systems, program logics, and combinations of both. Thus, subtle errors may arise both at the conceptual level (e.g. by formulating unsound proof rules), or at an implementation level (e.g. by omitting some check). In order to prevent such flaws that could be exploited by malicious code, the consortium decided early in the project to pursue a foundational approach in which the verification infrastructure deployed for certificate checking is formally verified using a proof assistant. This foundational approach, which has been pioneered by Appel [20], offers several advantages:

1. *small TCB*: Foundational Proof Carrying Code (FPCC) considerably reduces the trusted computing base (TCB), since the verification condition generator is formally proved sound w.r.t. the operational semantics. In addition, FPCC provides the expected level of guarantee for the correctness of the certificate checking infrastructure;

2. *uniform support for verification methods*: since all justifications are ultimately given in terms of the operational semantics, FPCC naturally supports the use of different verification methods, such as type systems and program logics.

However, current instances of FPCC do not scale easily to large programs, in particular because they favor deductive reasoning, and do not exploit the computational power of the underlying proof assistant. Yet recent research demonstrates that a tight integration between deduction and computation contributes significantly to the scalability of proof checking. In particular, computational reflection, whose goal is to replace deduction by computation, has proved an effective means to carry computation-intensive proofs for facts that cannot be established by purely deductive means; in addition, reflective proofs yield small certificates, because computations are not recorded in proof terms. In order to deliver the benefits of FPCC while avoiding problems of scalability and size of certificates, the MOBIUS verification architecture makes a heavy use of computational reflection: the verification condition generator (VCgen), which forms the focus of this chapter, and the information flow lightweight checker (LWC) of [24], are implemented in a reflective style, and provide prime examples of *reflective proof carrying code.*

In order to maximize flexibility and scalability, the verification relies crucially on hybrid methods that combine static analysis and verification condition generation. Indeed, the VCgen generates proof obligations for *every* possible path in the control flow graph of the program: in the case of Java source or bytecode programs, the number of possible execution paths may turn exceedingly large in proportion to the size of code, since many instructions may throw an instruction, and thus the control flow graph contains a large number of edges that are related to the exceptional behavior of programs. However, most exceptional paths are unfeasible, i.e. no concrete program execution can follow such path: for example, a recent study on null pointers in Java has shown that simple analyses can prove that nearly two thirds of references are meant to be non-null. Thus, eliminating unfeasible paths prior by using safety type systems yields a very significant pay-off and greatly reduces the number of proof obligations. In the VCgen, the possibility of performing type-based analyses and logic-based analyses is provisioned by the notion of hybrid specification and hybrid certificate: the former combines typing information and logical assertions, and the latter provides certificates for the type-based analysis and for proof obligations. Section 2.4 provides a detailed account of the combination of type-based analyses and logical verification. Section 2.5 shows that, for a hybrid verification method based on numerical static analysis and verification condition generation, compilation preserves proof obligations and therefore it is possible to transfer evidence from source to compiled programs. To overcome the loss of precision incurred by performing static analyses on compiled (rather than source) code, we extend the bytecode analysis with a symbolic execution of stack expressions.

2.1 A primer on reflection

The objective of this chapter is to provide a gentle introduction to reflection through an overview of the underlying principles and an illustration of its applications. For pedagogical purposes, we focus on a basic application, namely primality testing, rather than applications to programming language semantics. This section is based on [51].

2.1.1 Principles

In the Coq system, the rewriting steps are explicit in a proof: each step builds a predicate having the size of the current goal when the rewriting was performed, hence the size of the proof term heavily depends on the number of these rewriting steps. The reflection technique introduced by [18] takes benefit of the reduction system of the proof assistant to reduce the size of the proof term computed and consequently to speed up its checking. It relies on the following remark:

- Let $P : A \to \mathit{Prop}$ be a predicate over a set A.
- Suppose that we are able to write in the system a semi decision procedure f, such that f is computable and if f returns *true* on the entry x, then $P(x)$ is valid, that is to say:
  `f_correct:  forall x, f(x)=true -> P(x).`

If we want to prove $P(y)$ for a particular y, and if we know that $f(y)$ *reduces* to *true*, then we can simply apply the lemma `f_correct` to y and to a proof that *true* = *true*. Thanks to the conversion rule which allows to change implicitly the type of a term by an equivalent (modulo β-reduction):

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad T \equiv U}{\Gamma \vdash t : U}$$

This latter proof, which is `(refl_equal true)`, is also implicitly a proof that $f(y) = \mathit{true}$ because $f(y)$ reduces to *true*, so $\mathit{true} = \mathit{true}$ is convertible with $f(y) = \mathit{true}$. Finally the proof of $P(y)$ we have built is :

```
f_correct y (refl_equal true)
```

The size of such a proof now only depends on the size of the particular argument y and does not depend on the number of implicit β-reduction steps: explicit rewriting steps have been replaced by implicit β-reductions. The size of the proof term of the correctness lemma for f may be large, it is only done once and for all. It will be shared by all the instantiations and will no more be type-checked.

2.1.2 Efficiency

The efficiency of reflection of course strongly depends on the efficiency of the decision procedure itself, but also on the efficiency of the system to perform convertibility.

Convertibility is generally implemented in a purely interpretative way[1], because it is a hard task to perform strong β-reduction (reduction occurs also under binders) in a compiled setting. In [49], Grégoire and Leroy show how to decide β-equivalence on terms, by compiling proof-terms to the bytecode of an abstract machine derived with minimal modifications from the ZAM machine used in the Objective Caml bytecode interpreter. Their approach yields substantial speed-ups in reflective proofs, as documented e.g. in [50, 51], and is formally justified by results of Barras and Grégoire [22], who show that it is sound for the compiler to erase type annotations.

One important research issue to improve the performance of convertibility checking is the treatment of proof terms during equivalence checking. The types being compared for equivalence can contain arbitrary proof terms, which are often large and costly to normalize. However, the general principle of proof irrelevance suggests that it might not be necessary to normalize them, since a proof of a given proposition is (morally) just as good as any other proof of this proposition. This suggests erasing all proof terms before testing the equivalence of two types. More work is needed to prove that the corresponding relaxed conversion rule is logically consistent, and to implement the improved convertibility checker. We anticipate that checking convertibility modulo proof irrelevance would yield significant performance improvement.

2.1.3 Applications

Primality checking was studied as a prime example of proof carrying result in Deliverable D4.1. In this section, we illustrate the benefits of reflection by comparing deductive and reflective certificates for primality checking using Pocklington's certificates.

Pocklington's criterion

Pocklington's criterion provides a set of sufficient conditions that must be verified by some partial prime decomposition of $n-1$ to ensure that the number n is prime. *Given a natural number $n > 1$, a witness a, and some pairs $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$, it is sufficient for n to be prime that the following conditions hold:*

$$\begin{array}{rcll}
p_1 \ldots p_k & \text{are} & \text{prime numbers} & (2.0)\\
(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) & | & (n-1) & (2.1)\\
a^{n-1} & = & 1 (\equiv n) & (2.2)\\
\forall i \in \{1,\ldots,k\}\ \gcd(a^{\frac{n-1}{p_i}} - 1, n) & = & 1 & (2.3)\\
p_1^{\alpha_1} \ldots p_k^{\alpha_k} & > & \sqrt{n}. & (2.4)
\end{array}$$

Thus, checking primality of a natural number n with certificate $p_1, \ldots, p_k$ and a boils down to performing numerical computations to verify conditions 1-4, and verifying condition 0 recursively–note, however, that Pocklington's criterion cannot prove that 2 is prime.

Certificates

In order to be self-contained, a certificate must contain a,$p_1, \alpha_1, \ldots, p_k, \alpha_k$, as well as certificates for the p_i's and the factors occurring in these new certificates. This leads to a recursive notion of Pocklington certificate whose verification consists entirely of computations. Thus, a certificate for a single number n is given by the tuple $c = \{n, a, [c_1^{\alpha_1}; \ldots; c_k^{\alpha_k}]\}$ where $c_1, \ldots, c_k$ are certificates for the prime numbers $p_1, \ldots, p_k$. This means certificates can be understood as trees whose branches are themselves certificates corresponding to the prime divisors. Such structures are easily handled in Coq as an inductive type. A certificate is either:

- such tuples: $c = \{n, a, [c_1^{\alpha_1}; \ldots; c_k^{\alpha_k}]\}$[2]

[1] By interpretative, we mean algorithms that perform the conversion test by explicitly manipulating proof terms represented as trees.

[2] In the following, to shorten certificate we write c_i instead of c_i^1.

- a pair (n, ψ) composed by a number n and a proof ψ that this number is prime.

The second case is added in order to allow primality proofs which do not rely on Pocklington's theorem. This is useful for 2 (which cannot be proved prime using Pocklington's theorem) but also for using other methods which may be more efficient than Pocklington for some numbers.

Using this representation, a possible certificate for 127 is:

$$\begin{aligned}\{127, 3, [\{&7, 2, [\{3, 2, [(2, \mathsf{prime2})]\}; (2, \mathsf{prime2})]\};\\ &\{3, 2, [(2, \mathsf{prime2})]\};\\ &(2, \mathsf{prime2})]\}\end{aligned}$$

where prime2 is a proof that 2 is prime. One can remark that this kind of representation duplicates some certificates (here 3 and 2). So, the verification routine will verify many times these certificates. It order to share certificates, trees are flattened to lists. In this case this yields:

$$[\{127, 3, [7; 3; 2]\}; \{7, 2, [3; 2]\}; \{3, 2, [2]\}; (2, \mathsf{prime2})].$$

Note that doing so, the certificates for 2 and 3 now appear only once.

The above intuition translates straightforwardly into the following Coq definitions. First, one introduces the notion of partial factorization which is a list of prime numbers and their exponent (`dec_prime`). Second, one defines the notion of pre-certificate which is either a pair of a prime number n and its primality proof (`Proof_certif`), or a tuple of a prime number n, a witness a and a list of numbers representing a partial factorization of the predecessor of n (`Pock_certif`). This case is not self contained, since it does not contain the primality proofs of the numbers in the partial factorization. This is why it is called pre-certificate. Finally, a complete certificate is a list of pre-certificates. The head of the list is generally a triple

$$(\texttt{Pock\_certif}\ n\ a\ d)$$

and the tail contains pre-certificates for numbers appearing in the factorization list d of the first one (and recursively).

```
Definition dec_prime := list (positive*positive).

Inductive pre_certif : Set :=
 | Pock_certif : forall n a : positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif.

Definition certificate := list pre_certif.
```

Certificate checking is reported in [51].

Benchmarks

Certificates are checked in Coq with processor Intel Pentium 4 (3.60 GHz) and a RAM of 1Gb, using the compilation scheme described in [49].

Figure 2.1 gives the time to build the certificates for the 100000 first primes, the size of the certificate and the time for the Coq system to check them. On average, generating certificates for a small prime number takes about 3.10^{-5} seconds, their sizes are 215 bytes average, and it takes about 0.0144 seconds to verify.

Figure 2.2 makes a comparison between the deductive approach (the one developed by Oostdijk and Caprotti) and our reflexive one, using curious primes. The 44 digits prime number $(2^{148} + 1)/17$ is the biggest one proved in Coq before our work.

As expected, the reflexive method considerably reduces the size of the proof, as showed by the size column of the table. For the prime number $(2^{148}+1)/17$, the size is reduced by a factor 1500. The reduction of the proof size is natural since explicit deduction is replaced by implicit computation.

from - to	build	size	verify
2 - 5000	0.15s	989K	35.85s
5001 - 10000	0.17s	1012K	42.59s
10001 - 20000	0.38s	2.1M	134.14s
20001 - 30000	0.38s	2.1M	138.30s
30001 - 40000	0.38s	2.1M	145.81s
40001 - 50000	0.38s	2.2M	153.65s
50001 - 60000	0.41s	2.2M	153.57s
60001 - 70000	0.43s	2.2M	158.13s
70001 - 80000	0.39s	2.2M	160.07s
80001 - 90000	0.40s	2.2M	162.58s
90001 - 100000	0.44s	2.2M	162.03s

Figure 2.1: Time to verify the first 100000 prime numbers

		size		time		
prime	digits	deduc.	refl.	deduc.	refl.	refl. + VM
1234567891	10	94K	0.453K	3.98s	1.50s	0.50s
74747474747474747	17	145K	0.502K	9.87s	7.02s	0.56s
1111111111111111111	19	223K	0.664K	17.41s	16.67s	0.66s
$(2^{148}+1)/17$	44	1.2M	0.798K	350.63s	338.12s	2.77s
P_{200}	200	-	2.014K	-	-	190.98s

Figure 2.2: Comparison with the non recursive method

The three last columns compare the verification time for the different approaches. Without the use of the virtual machine, the reflexive approach is a little bit faster, but the times are comparable. If one verifies the proof with the version of Coq using a virtual machine to perform the conversion test (here to compute the result of the checking function), one can observe a gain of a factor 9 for small examples to more than 120 for the biggest ones. This means that the combination of computational reflexion with the virtual machine allows small proofs that are quickly verified.

Note that when verifying a reflexive proof the time consuming task is the reduction of decision procedure. It is precisely the reduction that the virtual machine improves. This explains why one gets such a speed up. Using virtual machine for checking deductive proofs usually does not provide any speed up since these proofs do not use much reduction.

The reflexive approach can be used to prove a new random prime number of 200 digits:

$$\begin{aligned} P_{200} = \quad & 67948478220220424719000081242787129583354660769625 \\ & 17084497493695001130855677194964257537365035439814 \\ & 34650243928089694516285823439004920100845398699127 \\ & 45843498592112547013115888293377700659260273705507 \end{aligned}$$

in 191 seconds and the proof size is 2K.

Figure 2.3 gives the time to verify the certificates of the 7th to the 18th first Mersenne numbers (using a Pocklington certificate)—the 7th first which are part of the first 100000 primes. The biggest one is a 969 digit number.

For all these benchmarks, we have to keep in mind that Coq uses its own arithmetic: numbers are encoding by a inductive type, i.e. a chained list of booleans. No native machine arithmetic is used. This means that the computations which are done to check certificates are more similar to symbolic computation than to numerical computation. For example, when dealing with the 20th Mersenne number, a 4422 digits number in base 2 (1332 in base 10), we manipulate list of 4423 elements. The fact that we are capable to

#	n	digits	years	discoverer	certificate	time
8	31	10	1772	Euler	0.527K	0.51s
9	61	19	1883	Pervushin	0.648K	0.66s
10	89	27	1911	Powers	0.687K	0.94s
11	107	33	1914	Powers	0.681K	1.14s
12	127	39	1876	Lucas	0.775K	2.03s
13	521	157	1952	Robinson	2.131K	178.00s
14	607	183	1952	Robinson	1.818K	112.00s
15	1279	386	1952	Robinson	3.427K	2204.00s
16	2203	664	1952	Robinson	5.274K	11983.00s
17	2281	687	1952	Robinson	5.995K	44357.00s
18	3217	969	1957	Riesel	7.766K	94344.00s
19	4253	1281	1961	Hurwitz	–	–
20	4423	1332	1961	Hurwitz	–	–

Figure 2.3: Time to verify Mersenne numbers

perform such a symbolic computation clearly indicates that the introduction of the virtual machine in Coq is an effective gain in computing power.

2.2 A review of Deliverable D3.1

Deliverable D3.1 defines two essential components of the MOBIUS architecture: the MOBIUS base logic, and the VCgen. These components are related to the Bicolano semantics (also described in Deliverable D3.1), since the correctness of both is proved with respect to the semantics. In this section, we'll concentrate in the VCgen.

The VCgen is a tool used to prove that programs are correct with respect to their specification. Given a specified program, the VCgen computes a set of formulas (proof obligations) whose validity implies the validity of the program with respect to its specification. While the VCgen is proved correct against the Bicolano semantics, in practice, it has two drawbacks that hinder its use in a proof assistant like Coq: the large amount of time it takes to check a proof, and the number of generated proof obligations. In the rest of the section, we'll describe in more detail these drawbacks, and propose solutions for them.

Checking proofs The components for program verification described in Deliverable D3.1, namely, the MOBIUS logic and the VCgen, were applied to an example program in order to exemplify the use of both techniques. In appendix D of Deliverable D3.1, the Coq script that prove that the example program satisfies its specification is given, where the corresponding formula is generated by applying the VCgen on the program. However, checking that this script produce a correct proof for the generated formula takes an enormous amount of time. In fact, it is prohibitively high for mobile devices (which are the target of the MOBIUS project).

To solve this problem, we redefine the VCgen, in this case, as a deep embedding in Coq (Sect. 2.3). This allows us to use reflective tactics that significantly reduce the time needed to check a proof. Deep embeddings have several advantages as opposed to shallow embeddings (such as the VCgen described in Deliverable D3.1). As shown in [108], proofs for deep embeddings are smaller, and more importantly, we can optimize and simplify the formulas generated during their construction. We exploit this fact by combining the VCgen with static analyses (Sect. 2.4).

Generated proof obligations As mentioned before, to verify a program, one needs to prove that is correct with respect to its specification for *every* path in the control flow graph. In Java bytecode programs, there are several instructions whose execution may result in an exception being thrown. Hence, the control flow graph of a Java bytecode program contains many edges that are related to exceptional execution. For instance, instructions like *Getfield* and *Invokevirtual*, will result in a *NullPointerException* being thrown if applied to a null pointer. For these instructions, one needs to verify that both paths of execution (normal for non-null pointers, exceptional for null pointers) satisfy the specification. This means that the VCgen will generate a number of proof obligations that is very large compared to the size of the program. However, a null-pointer analysis can be used to prove that most accesses to pointers are safe (i.e. an exception cannot be thrown). In those cases, it is not necessary to generate two proof obligations. As an example, consider the following excerpt of Java bytecode:

$$\begin{array}{ll} pc_1 & \mathit{Vstore}\ x \\ pc_2 & \mathit{Getfield}\ f \\ pc_3 & \ldots \end{array}$$

A VCgen (denoted by VC) generates two proof obligations for the program point pc_2, corresponding to normal and exceptional execution:

$$\begin{array}{c} lv(x) \neq \mathtt{null} \Rightarrow VC(pc_3) \\ \wedge\ lv(x) = \mathtt{null} \Rightarrow VC(pc_{\mathrm{exc}}), \end{array}$$

where lv access the local variable array, and pc_{exc} is the program point corresponding to the exception handler. If an static analysis can ensure that the pointer will be non-null at pc_2, then the VCgen can generate only one proof obligation:

$$lv(x) \neq \mathtt{null} \Rightarrow VC(pc_3)$$

In Sect. 2.4 we show a way to combine a VCgen with static analysis with the purpose of removing proof obligations corresponding to unfeasible paths. The proof of correctness of the VCgen depends on the soundness of the analysis.

2.3 Deep embedding of the VCgen

We present an overview of the design of the deeply embedded VCgen. The overall design of the VCgen follows closely to the VCgen described in Deliverable D3.1. To differentiate them, we will refer to the one described here as the deep VCgen, and to the one described in Deliverable D3.1 as the shallow VCgen.

2.3.1 Overview

The design of the deep VCgen is similar to the one of the shallow VCgen, and is also applied to annotated programs. Let us recall that an annotated program is a triple $(p, \mathit{subclass}, \mathit{MST})$, where p is a JVM program, $\mathit{subclass}$ is the subclass relation, and MST is the method specification table of p. A method specification table is a partial mapping from methods to pairs of global method specifications, and local specification tables of a method. A global method specification contains the precondition and postcondition of a method. A local specification table is a partial mapping from program points to assertions (also called invariants). The actual definition of the method specification table will be given later.

The definition of the deep VCgen is based on two mutually dependent functions:

$$\begin{array}{lll} \mathsf{swp_i} & : & \mathit{AnnotProg} \to \mathcal{M} \to \mathit{PC} \to \mathit{SInitState} \to \mathit{SLocalState} \to \mathit{Assertion} \\ \mathsf{swp_l} & : & \mathit{AnnotProg} \to \mathcal{M} \to \mathit{PC} \to \mathit{SInitState} \to \mathit{SLocalState} \to \mathit{Assertion} \end{array}$$

where $\mathit{Assertion}$, $\mathit{SInitState}$, $\mathit{SLocalState}$, are types that describe assertions and states; their definition is given below. The function $\mathsf{swp_i}\ p\ m\ pc$ computes the weakest precondition at program point pc of program p, while $\mathsf{swp_l}\ p\ m\ pc$ does the same, but also using the local specification table of m. These functions corresponds to functions $\mathsf{wp_i}$ and $\mathsf{wp_l}$ of the shallow VCgen.

2.3.2 Syntax of the deep VCgen

The deep VCgen defines a number of data structures that allows us to write expressions that represent program properties. These properties involve the different domains defined in the Bicolano semantics, namely, heaps (*Heap*), local variables (*LocalVar*), operand stacks (represented in Bicolano using *list value*), values (*value*), and auxiliary types such as locations in the heap (*AddressingMode*), and type of locations (*LocationType*), references (*Location*), integers (*Int*), return values (*ReturnVal*), and classes (*ClassName*). We will define data structures that represent these domains and the operations we can use on them. These structures should be expressive enough so that we are able to describe functional properties about programs.

Program specifications is described by three kind of assertions: preconditions, invariants and postconditions. There are some restrictions that apply to each kind of assertion. Preconditions are assertions that should be valid in the program state just before executing a method. They are allowed to refer to the initial state of the method (composed by the local variables array, and the heap).

Invariants are propositions that should be valid at particular program points during the execution of the method. They can refer to the current local state (local variables array, heap, and operand stack), but also they can refer to the initial state of the method, that is, the value of the local variables array and the heap at the beginning of the method.

Postconditions should be valid when a method finishes its execution (normally or abnormally because of an uncaught exception). They refer to the initial state of the method, and to the *return* state, which consists of the value of the heap, and the return value of the method. In case of normal termination, the return value is the value returned by the method, and in case of abnormal termination, the return value is the location of the exception object.

We propose a data type for expressing assertions that is generic in the "values" that the assertions can refer to. Then, we obtain the data type representing preconditions, invariants and postconditions by a suitable instantiation of the generic data type to the type of values that each kind of assertion can refer to. For instance, the values used for preconditions permit to refer to the initial state, that is, the heap and local variables array at the beginning of execution of the method.

As it turns out, the representation of some of the types defined in Bicolano will also be generic in the kind of values.

We divide the presentation of the deep VCgen in three parts. In the first part, we present the data types for the domains defined in Bicolano; in the second part, we present the data type for representing assertions; finally, in the third part, we instantiate the data type of the first two parts with the corresponding values to obtain preconditions, invariants and postconditions.

Types of Bicolano Each of the types defined in Bicolano has its own data type to represent the possible operations on them. To give a detailed example, we show the data types for representing local variables, operand stacks, and heaps. All these types are parametric in the type of values, which we call V here.

$$\begin{array}{rcl} LV(V) & := & \texttt{LVvar nat} \\ & | & \texttt{LVupd}\ (LV\ \ V)\ \ Var\ \ V \\ & | & \texttt{LVStack2LV}\ (St\ \ V)\texttt{nat} \end{array}$$

$$\begin{array}{rcl} St(V) & := & \texttt{StVar nat} \\ & | & \texttt{StEmpty} \\ & | & \texttt{StPop}\ (St\ \ V) \\ & | & \texttt{StPush}\ V\ (St\ \ V) \end{array}$$

$$\begin{array}{rcl} H(V) & := & \texttt{Hvar nat} \\ & | & \texttt{Hupd}\ (H\ \ V)\ (AM\ \ V)\,V \end{array}$$

Var is the type of variables identifiers from Bicolano. The constructor `LVvar` is used to refer to a value that is universally quantified. As we will see later, the data type for assertions includes a construction for universal quantification. The argument of `LVvar` is used as a de Bruijn index.

The definition of *LocalVar* in Bicolano specifies one accessor (*get*) and one modifier (*update*). In the definition above, the constructor `LVupd` is a representation of the modifier *update*. The accessor (*get*) returns a value, so it is not represented here.

For operand stacks, we have constructors to push and pop values, and `StVar` for universal quantification over operand stacks, and a constructor that represents an empty stack.

The data type for heaps is similar, allowing universal quantification (with `Hvar`) and updates to a location in the heap, where *AM* is the type of locations in the heap, called *AddressingMode* in Bicolano.

$$\begin{array}{rcl} AM(V) & := & \texttt{AMvar nat} \\ & | & \texttt{AMStaticField}\ \mathit{FieldSignature} \\ & | & \texttt{AMDynamicField}\ V\ \mathit{FieldSignature} \\ & | & \texttt{AMArrayElement}\ V\ V \end{array}$$

As with the cases above, we have a constructor for universal quantification, constructors to access static and dynamic fields, and array elements. In the case of dynamic fields, the first argument represents the object we are accessing; for array accesses, the first parameter is the object, and the second is the index, both being values.

Assertions The data type representing assertions is given below. It is generic on the type of values (V) and in the type of predicates (P). This way the assertions can be extended by instantiating the type P to an appropriate data type that represents the additional predicates.

$$\begin{array}{rcl} expr(V,P) & ::= & \textit{(* Logic operators *)} \\ & | & \texttt{False} \\ & | & \texttt{True} \\ & | & expr(V,P) \wedge expr(V,P) \\ & | & expr(V,P) \vee expr(V,P) \\ & | & expr(V,P) \Rightarrow expr(V,P) \\ & | & \neg expr(V,P) \\ & | & \forall\ Type\ (expr(V,P)) \\ & & \textit{(* Comparison of values *)} \\ & | & \texttt{CompRef}\ CompRefOp\ V\ V \\ & | & \texttt{CompInt}\ CompIntOp\ V\ V \\ & & \textit{(* Other predicates *)} \\ & | & \texttt{Pred}\ P \end{array}$$

The constructors are divided in three groups. The first group contains the logical operators. Note that we have a constructor for universal quantification. It takes an element of *Type* and an assertion, where *Type* is an enumeration type containing all the different types of objects that we can quantify over (local variables, operand stacks, heaps, values, etc).

The second group contains comparison operations for references and numerical values.

The third group contains a single construction used to extend the data type of assertion with arbitrary predicates, as mentioned above.

Values Now we are going to define the different types of values with which we instantiate the data types defined above. First, we define a generic type of expressions for values

$$\begin{array}{rcl} Expr(V) & ::= & \texttt{EVar nat} \\ & | & \texttt{EFrom}\ V \\ & | & \texttt{EConst}\ value \\ & | & \texttt{EBinop}\ Op\ Expr\ Expr \\ & | & \texttt{ENeg}\ Expr \end{array}$$

The type of expressions contains a constructor for universal quantification over values (`EVar`), a lifting from values to expressions (`EFrom`) and from Bicolano values to expressions (`EConst`), and constructors that represent arithmetic operations on expressions (`EBinop` and `ENeg`).

Now we define the different possibilities for V in order to represent preconditions, postconditions and invariants. For preconditions, we define the data type *Vpre* that includes constructions to access the local variables array and the heap.

$$\begin{array}{lcl} \mathit{Vpre} & ::= & \texttt{preLget}\ \mathit{Var} \\ & | & \texttt{preHget}\ (\mathit{AM}\ (\mathit{Expr}\,\mathit{Vpre})) \end{array}$$

Values used in preconditions can be obtained from the local variables array (using `preLget`) and from the initial heap (using `preHget`).

In the same way, we define values for the invariants and the postcondition. In the case of invariants, we can access the current heap (`invHget`), the initial heap (`invHget0`), the current and initial local variables (`invLget` and `invLget0`, resp.) and the current operand stack (`invSt`). The corresponding data structure is defined as follows

$$\begin{array}{lcl} \mathit{Vinv} & ::= & \texttt{invHget}\ (\mathit{AM}\ (\mathit{Expr}\,\mathit{Vinv})) \\ & | & \texttt{invHget0}\ (\mathit{AM}\ (\mathit{Expr}\,\mathit{Vinv})) \\ & | & \texttt{invLget}\ \mathit{Var} \\ & | & \texttt{invLget0}\ \mathit{Var} \\ & | & \texttt{invSt}\ \texttt{nat} \end{array}$$

For postconditions, we can access the current heap (`postHget`), the initial heap (`postHget0`) and the return value (`postRes`).

$$\begin{array}{lcl} \mathit{Vpost} & ::= & \texttt{postHget}\ (\mathit{AM}\ (\mathit{Expr}\,\mathit{Vpost})) \\ & | & \texttt{postHget0}\ (\mathit{AM}\ (\mathit{Expr}\,\mathit{Vpost})) \\ & | & \texttt{postRes} \end{array}$$

Method specification Now we can instantiate the assertions with the values above in order to obtain the data types for expressing preconditions, invariants and postconditions. For preconditions and invariants, we simply instantiate with the values above, and using a dummy data type for extended predicates

$$\begin{array}{lcl} \mathit{PreCondition} & := & \mathit{expr}(\mathit{Expr}\ \mathit{Vpre}, \mathit{PredPre}) \\ \mathit{Invariant} & := & \mathit{expr}(\mathit{Expr}\ \mathit{Vinv}, \mathit{PredInv}) \end{array}$$

Note that we also instantiate with predicate types, *PredPre* and *PredInv*. We do not give their complete definitions, but only show some constructors when they are used.

The case of postcondition is not as simple, since the postcondition of a method is not a single assertion. Instead, the postcondition is a pair, consisting of a *normal* postcondition (that should be valid when the method terminates normally), and an *exceptional* postcondition (that should be valid when the method terminates abnormally). Further, the exceptional postcondition is also a pair: the first component consists of a list of pairs that associates assertions to exceptions, while the second component is an assertion. It functions as a kind of try-catch block, where we have assertions corresponding to some exceptions that may escape the method, and a default assertion in case the exception is not "catched".

Therefore, to define the postcondition of a method, we need first to define a simple assertion:

$$\mathit{SimplePost} \ := \ \mathit{expr}(\mathit{Expr}\ \mathit{Vpost}, \mathit{PredPost})$$

Then, the postcondition is defined as

$$\mathit{PostCondition} \ := \ \mathit{SimplePost} \times (\mathit{list}\ (\mathit{Exception} \times \mathit{SimplePost}) \times \mathit{SimplePost})$$

2.3.3 Interpretation of the deep VCgen

The meaning of an assertion in the deep VCgen is given by an interpretation function that translates an assertion into a Coq proposition. For each of the data types defined above, we have an interpretation function that translate them to the corresponding types in Bicolano. Since terms in these data types contain free variables, the interpretation functions are parametrized by a context that assigns a meaning to free variables. We won't give here the complete definition of the context type, *Cnxt*, but only show two operations that can be applied to a context: the function *get* takes a context and a natural number, and returns the corresponding term (remember that variables are represented by natural numbers using de Bruijn indices), and the operator $x \gg c$ adds variable x to the context c.

To exemplify, we show the definition of the interpretation function for local variables. It takes an element of type *LV* and a context, and returns a *LocalVar*. It is also parametrized by a function that interprets values.

$$\mathit{interp}_{\mathrm{LV}} \quad : \quad \mathit{LV} \times \mathit{Cnxt} \to \mathit{LocalVar}$$

$$\mathit{interp}_{\mathrm{LV}}(c,l) = \begin{cases} \mathit{get}(c,n) & \text{if } l = \texttt{LVvar}\ n \\ \mathit{LocalVar.update}\ (\mathit{interp}_{\mathrm{LV}}(c,l'))\ x\ (\mathit{interp}_{\mathrm{V}}(c,v)) & \text{if } l = \texttt{LVupd}\ l'\ x\ v \\ \mathit{stack2localvar}(\mathit{interp}_{\mathrm{St}}(c,s)) & \text{if } l = \texttt{LVStack2LV}\ s \end{cases}$$

The function *stack2localvar* is defined in Bicolano and is used to convert the elements of the stack into local variables when calling a method. That is, the arguments passed in the stack are converted into local variables in the called method. The function $\mathit{interp}_{\mathrm{St}}$ interprets a stack, returning a list of values, which is the Bicolano representation for a stack. Below are its type and definition

$$\mathit{interp}_{\mathrm{St}} \quad : \quad \mathit{St} \times \mathit{Cnxt} \to \mathit{list\ value}$$

$$\mathit{interp}_{\mathrm{St}}(s) = \begin{cases} v & \text{if } s = \texttt{LVvar}\ v \\ vs & \text{if } s = \texttt{StPop}\ s' \wedge \mathit{interp}_{\mathrm{St}}(s') = v :: vs \\ \mathit{interp}_{\mathrm{V}}(v) :: \mathit{interp}_{\mathrm{St}}(s') & \text{if } s = \texttt{StPush}\ v\ s' \end{cases}$$

For assertions, the interpretation function is the following (selected cases):

$$\mathit{interp}_{\mathrm{WP}}(c,e) = \begin{cases} \mathit{True} & \text{if } e = \texttt{True} \\ \mathit{False} & \text{if } e = \texttt{False} \\ \mathit{interp}_{\mathrm{WP}}(c,e_1) \wedge \mathit{interp}_{\mathrm{WP}}(c,e_2) & \text{if } e = e_1 \wedge e_2 \\ \mathit{forall}\ x : \mathit{mkType}\ T, \mathit{interp}_{\mathrm{WP}}(x \gg c, f) & \text{if } e = \forall (x:T)\ f \\ \vdots & \end{cases}$$

Note that in the translation of the universal quantifier, we extend the context by introducing the newly created variable. The function *mkType* takes as argument an element from the enumeration of types defined in the deep VCgen, and returns the corresponding *real* type defined in the Bicolano semantics.

2.3.4 Example Program

In this section, we present a simple program to exemplify how to write specifications with the deep VCgen. We are not concerned with its verification here, and postpone it to Sect. 2.4.6, after the description of the weakest precondition calculus.

Figure 2.4 contains the Java source code together with the JML specification of the example. The program consists of a class `Point` that represents point in the plane by two integer coordinates (fields `x` and `y`). We have defined only one method, `translate`, that translates a point using another point as parameter. The precondition states that the parameter of `translate` is non-null, and the postcondition states that the

```
class Point {
  private int x, y;

  //@ require t != null;
  //@ ensures \result .x == \old(this.x) + t.x &&
  //@         \result .y == \old(this.y) + t.y;
  public Point translate (Point t) {
    x += t.x;
    y += t.y;
    return this;
  }
}
```

Figure 2.4: Java source code of the example program

result is in fact the translation. The specification doesn't contains any internal invariants (they are not necessary, since the code doesn't contain any loop). Figure 2.5 shows the corresponding bytecode.

We translate the specification of method `translate` into Coq propositions. The precondition states that pointer `this` and the argument of the method are non-null references. This is stated using a predicate for preconditions, `isValidRef`. Note that the pointer `this` and the argument are located, respectively, in the first and second position in the local variables array. The precondition is stated as follows:

```
 Definition translate_pre :=
   isValidRef (preLget 0%N) /\ isValidRef (preLget 1%N).
```

The postcondition states the functional behavior of the function. Its definitions is divided two parts, corresponding to normal termination and exceptional termination of the method. The normal postcondition is defined as

```
Definition translate_post_normal (s0:InitState) (t:ReturnState) :=
  CompInt EqInt (postHget (postRes (AMDynamicField x)))
                (EBinop AddInt (postHget0 (AMDynamicField (postLget0 0%N)) x)
                               (postHget0 (AMDynamicField (postLget0 1%N)) x))
  /\ CompInt EqInt (postHget (postRes (AMDynamicField x)))
                   (EBinop AddInt (postHget0 (AMDynamicField (postLget0 0%N)) y)
                                  (postHget0 (AMDynamicField (postLget0 1%N)) y)).
```

where `x` and `y` are the definitions of the fields of the class `Point`.

Since the argument is not null, the method `translate` will always terminate normally. We state this fact by defining the exceptional postcondition as a contradiction:

```
Definition translate_post_exc (s0:InitState) (t:ReturnState) := False.
```

Finally, the postcondition of the method is defined by composing both definitions above:

```
Definition translate_post := (translate_post_normal, [], translate_post_exc).
```

As mentioned above, the verification of this method is described in Sect. 2.4.6.

2.3.5 Weakest precondition

We present the weakest precondition (WP) calculus. The main structure is similar to the way the WP is computed in the shallow VCgen: it's defined based on two mutually recursive functions, swp_i and swp_l, that

```
class Point extends java.lang.Object{
Point();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public Point translate(Point);
  Code:
   0:   aload_0
   1:   dup
   2:   getfield        #2; //Field x:I
   5:   aload_1
   6:   getfield        #2; //Field x:I
   9:   iadd
   10:  putfield        #2; //Field x:I
   13:  aload_0
   14:  dup
   15:  getfield        #3; //Field y:I
   18:  aload_1
   19:  getfield        #3; //Field y:I
   22:  iadd
   23:  putfield        #3; //Field y:I
   26:  aload_0
   27:  areturn

}
```

Figure 2.5: Java bytecode of the example program

have the same meaning as the functions $\mathsf{wp_i}$ and $\mathsf{wp_l}$ of the shallow VCgen. Also, to simplify the presentation, we've defined assertion transformers that have the same role as the ones defined for the shallow VCgen.

The result type of these functions is *Assertion*, which is defined from *expr* by instantiating to an appropriate type for values and predicates. The type of values, *Vassert* is general enough to represent values that can come from preconditions, invariants and postconditions. We won't state here its definition since it will not be necessary. Using *Vassert*, we define the following types for representing the state when we compute the WP:

$$\begin{array}{rcl} \mathit{SHeap} & ::= & H(\mathit{Expr}\ \mathit{Vassert}) \\ \mathit{SLocalVar} & ::= & LV(\mathit{Expr}\ \mathit{Vassert}) \\ \mathit{SStack} & ::= & St(\mathit{Expr}\ \mathit{Vassert}) \\ \mathit{SInitState} & ::= & \mathit{SLocalVar} \times \mathit{SHeap} \\ \mathit{SLocalState} & ::= & \mathit{SHeap} \times \mathit{SStack} \times \mathit{SLocalVar} \end{array}$$

The type of predicates used allows us to express the conditions needed for some instructions and that cannot be expressed in the base type *expr*. The definition is the following (excerpt)

$$\begin{array}{rcl} \mathit{PredAssert} & ::= & \texttt{HTypeObject}\ \mathit{SHeap}\ \mathit{Vassert}\ \mathit{SClassName} \\ & | & \texttt{HnewObj}\ \mathit{SHeap}\ \mathit{SClassName}\ \mathit{Vassert}\ \mathit{SHeap} \\ & | & \texttt{SubClass}\ \mathit{SClassName}\ \mathit{SClassName} \\ & & \vdots \end{array}$$

The assertions that are used when computing the WP are defined by the type

$$\mathit{Assertion} ::= \mathit{Expr}(\mathit{Vassert}, \mathit{PredAssert})$$

Finally, we need translations functions that take a precondition (invariant, postcondition) and return an *Assertion*. They are defined with the following types:

$$\begin{array}{rcl} \mathit{substPre} & : & \mathit{PreCondition} \to \mathit{SInitState} \to \mathit{Assertion} \\ \mathit{substInv} & : & \mathit{Invariant} \to \mathit{SInitState} \to \mathit{SLocalState} \to \mathit{Assertion} \\ \mathit{substPost} & : & \mathit{PostCondition} \to \mathit{SInitState} \to \mathit{SHeap} \to \mathit{Expr}(\mathit{Vassert}) \to \mathit{Assertion} \end{array}$$

Basically, they are defined as substitutions. For instance, the definition of $\mathit{substPre}\ p\ s_0$ consists of substituting every occurrence of *preHget* in p with the heap from s_0, and every occurrence of *preLget* in p with the local variables from s_0. Similar definitions apply to *substInv* and *substPost*.

While the way the WP is computed is similar on both VCgen, there are two main differences. First is the treatment of exceptions. To be able to decide whether an exception is caught or not, a subclass relation was added to the annotated program. Using this relation, it's possible to write a function *lookuphandlers* that returns, for a given program point and exception class, the program point where the handler for that exception is located. The definition of the function *lookuphandlers* in the shallow VCgen is cumbersome to work with. For the deep VCgen we propose a different definition of *lookuphandlers* that generates a proof obligation for each possible exception. While this choice increase the number of proof obligations, its simpler to work with when it comes to build proofs using the Coq proof assistant. On the other hand, the size can be further reduced using the techniques described in Sect. 2.4.

Second, the result of the WP is now a syntactic term in the language of the deep VCgen. This means that we can analyze and manipulate it, something not possible with a shallow embedding. We exploit this fact by defining *wrapper* functions around the constructors of the types defined in the language. These wrapper functions allows to simplify and remove unneeded constructors from the resulting term.

As an example, we show wrapper function corresponding to updating the local variables.

$$\mathit{mkLVupd}\ x\ v\ l = \begin{cases} \texttt{LVupd}\ x\ v\ l & \text{if } l = \texttt{LVvar}\ l' \\ \texttt{LVupd}\ y\ v'\ (\mathit{mkLVupd}\ x\ v\ l') & \text{if } l = \texttt{LVupd}\ y\ v'\ l' \text{ and } y \neq x \\ \texttt{LVupd}\ x\ v\ l' & \text{if } l = \texttt{LVupd}\ x\ v'\ l' \end{cases}$$

For all constructors in the data types defined above, we define a wrapper function. For updating the state, besides *mkLVupd*, we define functions for updating the heap (*mkHupd*), pushing values in the stack (*mkSPush*) among others. We also define wrapper functions to obtain values from the state: to read a value from the heap (*mkHget*), from the local variables array (*mkHget*) and from the stack (*mkStget*).

When building the WP for a method, we don't use directly the constructors, for instance `VLVget` and `LVupd`, but instead use the wrapper functions, *mkLVupd* and *mkLVget*. As a result, the proof obligations are simplified, and also the resulting proof term is smaller.

We have defined wrapper functions for all other constructors, including some of *expr*, such as:

$$\mathit{mkAnd}(e_1, e_2) = \begin{cases} e_1 & \text{if } e_2 = \texttt{True} \\ e_2 & \text{if } e_1 = \texttt{True} \\ \texttt{False} & \text{if } e_1 = \texttt{False} \text{ or } e_2 = \texttt{False} \\ e_1 \wedge e_2 & \text{otherwise,} \end{cases}$$

that is used instead of $\wedge$. For readability, we are going to use $\wedge$ instead of *mkAnd*.

Assertion transformers

The assertion transformers are mainly similar to the ones defined for the shallow VCgen, and allows us to make a clearer presentation. The only big difference is in the exception handling. We introduce a new assertion transformer to handle exceptions, $\mathsf{swp}_{\mathrm{ExcUnknown}}$, used for instructions where the type of the (possible) raised exception is not known statically. These instructions include the *Invoke* family, where is not possible to know statically the class of exception that they can throw.

For completeness, we show all the assertion transformers. As in the case with the shallow VCgen, all these functions are parametrized by an annotated program p, a method m and a function $\mathsf{swp}_{\mathrm{l}}$.

- $\mathsf{swp}_{\mathrm{next}}\ pc$ computes the WP of the successor of pc:

$$\mathsf{swp}_{\mathrm{next}}\ pc = \begin{cases} \mathsf{swp}_{\mathrm{l}}\ pc' & \text{if } \mathit{next}_m\,(pc) = pc', \\ \lambda s_0\, s.\texttt{True} & \text{if } \mathit{next}_m\,(pc) = \bot. \end{cases}$$

- $\mathsf{swp}_{\mathrm{Bound}}\ i\ \mathit{length}\ T\ pc = \mathsf{swp}_{\mathrm{JvmCond}}(\texttt{isBound}\ i\ \mathit{length})\ T\ pc\ \mathit{ArrayIndexOutOfBoundsException}$
- for handling exceptions:

$$\mathsf{swp}_{\mathrm{Exc}}\ pc\ loc\ cn\ h\ l\ s_0 = \begin{cases} \texttt{Post}\ m\ s_0\ (h, loc) & \text{if } \mathit{lookuphandlers}\ m\ pc\ cn = \bot, \\ \mathsf{swp}_{\mathrm{l}}\ pc'\ (s_0, (h, loc :: \emptyset, l)) & \text{if } \mathit{lookuphandlers}\ m\ pc\ cn = pc' \end{cases}$$

$$\begin{array}{l} \mathsf{swp}_{\mathrm{JvmExc}}\ pc\ e\ h\ l\ s_0 = \\ \quad \forall\, h'\ \forall loc.\ \texttt{HnewObj}\ h\ (\mathit{javaLang}, e)\ loc\ h' \Rightarrow \mathsf{swp}_{\mathrm{Exc}}\ pc\ loc\ (\mathit{javaLang}, e)\ h'\ l\ s_0 \end{array}$$

  The function `Post` $m\ s_0\ s_r$ transforms the postcondition of method m using the initial and return states s_0 and s_r (by applying the function *substPost*).

- $\mathsf{swp}_{\mathrm{Null}}\ pc = \mathsf{swp}_{\mathrm{JvmExc}}\ pc\ \mathit{NullPointerException}.$
- $\mathsf{swp}_{\mathrm{Cond}}\ C\ T\ F = (C \Rightarrow T) \wedge (\neg C \Rightarrow F).$
- $\mathsf{swp}_{\mathrm{JvmCond}}\ C\ T\ pc\ e\ h\ l\ s_0 = \mathsf{swp}_{\mathrm{Cond}}\ C\ T\ (\mathsf{swp}_{\mathrm{JvmExc}}\ pc\ e\ h\ l\ s_0).$

$\mathsf{swp}_{\mathrm{Exc}}$ is not directly used when computing the WP. Instead, we use $\mathsf{swp}_{\mathrm{Bound}}$, $\mathsf{swp}_{\mathrm{Null}}$, and $\mathsf{swp}_{\mathrm{JvmExc}}$ for instruction where the type of exceptions that can be (possibly) raised is known statically. These include instructions such as *Getfield*, *Putfield* (that can raise a *NullPointerException*), *Vaload*, *Vastore* (that can raise a *ArrayIndexOutofBoundsException*) and *Checkcast* (that can raise a *CastClassException*).

However, for instructions in the *Invoke* family is not possible to know statically (in most of the cases) what type of exception can be raised. For these instructions, we introduce a new assertion transformer:

$$\mathsf{swp}_{\text{ExcUnknown}}\ pc\ loc\ cn\ h\ l\ s_0 = \mathsf{swp}'_{\text{ExcUnknown}}(handlers_m(pc))\ pc\ loc\ cn\ h\ l\ s_0,$$

where $handlers_m(pc)$ is the list of exception handlers of m whose range includes pc. Function $\mathsf{swp}'_{\text{ExcUnknown}}$ is defined by case analysis on the list of handlers. We have two cases:

- if $exl = []$, then
$$\mathsf{swp}'_{\text{ExcUnknown}}\ exl\ pc\ loc\ cn\ h\ l\ s_0 = \texttt{Post}\ s_0\ (h, loc)$$
- if $exl = ex :: exl'$, and cn' is the type of exceptions that are caught by the handler ex, then
$$\mathsf{swp}'_{\text{ExcUnknown}}\ exl\ pc\ loc\ cn\ h\ l\ s_0 = \begin{cases} \ \ \texttt{SubClass}\ cn\ cn' \Rightarrow \mathsf{swp}_{\mathrm{l}}\ pc'\ s_0(h, \texttt{StPush}\ loc\ \texttt{StEmpty}, l) \\ \wedge \neg\texttt{SubClass}\ cn\ cn' \Rightarrow \mathsf{swp}'_{\text{ExcUnknown}}\ exl'\ pc\ loc\ cn\ h\ l\ s_0 \end{cases}$$

Note that we have a conjunction for each exception handler in range. For example, suppose that we have two exception handlers for classes c_1 and c_2 located at pc_1 and pc_2. Then, we have the following proof obligations:

$$\begin{gathered}(\texttt{SubClass}\ cn\ c_1 \Rightarrow \mathsf{swp}_{\mathrm{l}}\ pc_1 \ldots) \\ \wedge (\neg\texttt{SubClass}\ cn\ c_1 \Rightarrow \texttt{SubClass}\ cn\ c_2 \Rightarrow \texttt{Post}\ \ldots)\end{gathered}$$

There are two proof obligations, depending on whether the exception is caught by the first handler or the second one.

Main functions: $\mathsf{swp}_{\mathrm{i}}$ and $\mathsf{swp}_{\mathrm{l}}$

These are the two main functions that compute the WP for a given method. The function $\mathsf{swp}_{\mathrm{i}}$ takes as parameter the program counter pc, the initial state s_0 and the current state s, and proceeds by case analysis on the instruction at pc. Roughly, it computes the conjunction of the WP of all successors of pc. If pc' is a successor of pc, it computes an expression of the form:

$$C_{(pc,pc')}(s) \Rightarrow P_{(pc,pc')}(\mathsf{swp}_{\mathrm{l}}(pc, s_0, s')),$$

where $C_{(pc,pc')}(s)$ is the condition that needs to be satisfied for the program to go from pc to pc', s' is the state derived from s after the execution of the instruction at pc, and P is a predicate transformer. The final result of $\mathsf{swp}_{\mathrm{i}}\ pc$ is the conjunction of the expression above taken over all successor of pc:

$$\mathsf{swp}_{\mathrm{i}}\ pc\ s_0\ s = \bigwedge_{pc' \in succ(pc)} C_{(pc,pc')}(s) \Rightarrow P_{(pc,pc')}(\mathsf{swp}_{\mathrm{l}}(pc, s_0, s'_{(pc,pc')})),$$

We show some relevant cases. The definition of $\mathsf{swp}_{\mathrm{i}}\ pc\ s_0\ s$ proceeds by case analysis in the instruction at pc, where $s = (h, os, l)$:

- if the instruction is *AconstNull*, then
$$\mathsf{swp}_{\mathrm{i}}\ pc\ s_0\ (h, os, l) = \mathsf{swp}_{\text{next}}\ pc\ s_0\ (h, mkStPush\ (\texttt{EConst null})\ os, l)$$

  In this case, we can consider that the condition C for this instruction is simply `True`.

- if the instruction is *Getfield f*, then

$$\begin{aligned}
\mathsf{swp_i}\ pc\ s_0\ (h, os, l) = \mathtt{let}\ loc = \mathit{mkStget}\ os\ 0\ \mathtt{in} \\
loc = \mathtt{EConst\ null} \Rightarrow \mathsf{swp_{Null}}\ pc\ h\ l\ s_0 \\
\wedge\, loc \neq \mathtt{EConst\ null} \Rightarrow \mathsf{swp_{next}}\ pc\ s_0\ (h, \mathit{mkStPush}\ (\mathit{mkHget}\ h\ (\mathtt{DynamicField}\ loc\ f)\ os, l)
\end{aligned}$$

  In this case, the condition C is the following:

$$\begin{aligned}
C_{(pc,pcNull)}(s) &= \mathit{mkStget}\ os\ 0 = (\mathtt{EConst\ null}) \\
C_{(pc,pcNext)}(s) &= \mathit{mkStget}\ os\ 0 \neq (\mathtt{EConst\ null})
\end{aligned}$$

  where *pcNull* and *pcNext* are the successor program points corresponding to exceptional execution and normal execution.

- if the instruction is *Return*, then

$$\mathsf{swp_i}\ pc\ s_0\ (h, os, l) = \mathtt{Post}\ m\ s_0\ (h, \mathit{mkStget}\ os\ 0)$$

  Again, in this case, the condition C is simply `True`.

Finally, the definition of $\mathsf{swp_l}$ is:

$$\mathsf{swp_l}\ pc\ s_0\ s = \begin{cases} \mathtt{Assert}\ m\ pc\ s_0\ s & \text{if } S.(pc) = A, \\ \mathsf{swp_i}\ pc\ s_0\ s & \text{if } S.(pc) = \bot \end{cases}$$

The function `Assert` behaves in a similar way to `Post` by transforming an invariant into an element of $Assertion$.

2.3.6 Correctness of the VCgen

The theorem stating correctness of the VCgen is very similar to the correctness of the shallow VCgen. First, we define the notions of certified method, and certified program.

Definition 2.3.1 (Certified methods) *Given an annotated program p, a precondition R, a postcondition T and a local specification table $\mathcal{S}$, a method m is certified if $p.MST(m) = (R, T, \mathcal{S})$ and the following property holds:*

$$\begin{aligned}
&\mathtt{certifiedMethod}\ p\ \mathcal{S}\ m \equiv \\
&\qquad (\forall h_0 \forall l_0 . \mathtt{Pre}\ m\ (h_0, l_0) \Rightarrow \mathsf{swp_l}\ p\ m\ \mathsf{init_m}\ ((h_0, l_0), (h_0, \mathtt{StEmpty}, l_0))) \\
&\quad \wedge \bigwedge_{\mathcal{S}(pc)=A} \forall h_0 \forall h \forall l_0 \forall l \forall os . \mathtt{Assert}\ m\ pc\ (h_0, l_0)\ (h, os, l) \Rightarrow \mathsf{swp_i}\ p\ m\ pc\ ((h_0, l_0), (h, os, l)),
\end{aligned}$$

Definition 2.3.2 (Certified programs) *An annotated program p is certified whenever the following property holds:*

$$\mathtt{certifiedProg}\ p \equiv \bigwedge_{p.MST(m)=(R,T,\mathcal{S})} \mathtt{certifiedMethod}\ p\ \mathcal{S}\ m.$$

Theorem 2.3.3 (Correctness of the VCgen) *For each annotated program p and method m, such that the proof obligation generated by the VC generator is satisfied (i.e. a proof of $interp_{\mathrm{WP}}(\mathtt{certifiedProg}\ p)$ exists) and $p.MST(m) = (R, T, \mathcal{S})$ we have that if the proposition $interp_{\mathrm{WP}}(\mathsf{swp_l}\ p\ m\ pc\ s_0\ s)$ is valid and the state (pc, s) evaluates to:*

- *the state (pc', s'), then the proposition $interp_{\mathrm{WP}}(\mathsf{swp_l}\ p\ m\ pc'\ s_0\ s')$ is valid,*
- *the return state (h, rv), then the proposition $interp_{\mathrm{WP}}(\mathtt{Post}\ m\ s_0\ (h, rv))$ is valid.*

2.4 Reducing Proof Obligations

The deep VCgen presented in the previous section has two main advantages with respect to the shallow VCgen of Deliverable D3.1, namely, a reduction in the size of proof terms, and a significant reduction in the time needed to check such a proof. However, the number of proof obligations is not greatly reduced. In this section we show a technique to reduce the number of proof obligations using static analyses.

2.4.1 Preliminary definitions

We will consider a fixed annotated program p, and a method m. PC is the set of program points of m, $\mathcal{G} \subseteq PC \times PC$ will denote the set of edges in the control flow graph of m (i.e. $\mathcal{G}$ is the set of ordered pairs (pc, pc') such that the execution of m can pass from program point pc to pc'). We have two special nodes, $pc_{\mathrm{N}}, pc_{\mathrm{E}} : PC$, that correspond to normal termination of m, or abnormal termination (due to an exception uncaught in m). These nodes have no associated instruction, and are introduced to simplify the following definitions. pc_0 is the initial point of m. *State* represent the states of the virtual machine; each $s \in State$ is actually a triple (h, l, s), where h is the heap, l the local variables, and s the operand stack. $State_0$ represents the set of initial states; each $s_0 \in State_0$ is a pair (h, l), where h is the heap and l the local variables. $\_ \rightsquigarrow \_ \subseteq (PC, State) \times (PC, State)$ is the operational semantics (note that the operational semantics has as implicit parameters the program p and the method m). This is a simplified version of the big-step semantics defined in Deliverable D3.1. The difference lies in the use of the special nodes pc_{N} and pc_{E}. In the big-step semantics, a special relation $\downarrow$: $(PC, State) \times ReturnState$, is considered for returning a value from a method, where $(pc, s) \downarrow r$ means that from the state (pc, s) the method returns the value r that can be either a normal value or an exception. Here, we consider, instead of the relation $\downarrow$, the steps $(pc, s) \rightsquigarrow (pc_{\mathrm{N}}, s)$ for returning a normal value, and $(pc, s) \rightsquigarrow (pc_{\mathrm{E}}, s)$ for returning a exception. In both cases, the returning value is in the top of the operand stack of state s.

Definition 2.4.1 *A* static analysis $\mathcal{A}$ *is a tuple* (D, t, I, f), *where*

- $D = (D, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$ *is a complete lattice denoting the domain of the analysis;*
- $t : \mathcal{G} \to (D \to D)$ *is the transfer function, such that for each* $(pc, pc') \in \mathcal{G}$, $t_{(pc,pc')}$ *is a monotone function in* D*;*
- $I : PC \to D$ *is the initial value; and*
- $f \in \{\uparrow, \downarrow\}$ *denotes the direction of the analysis.*

If $f = \uparrow$ *we say the analysis is* backward*, and if* $f = \downarrow$ *we say is* forward.

Definition 2.4.2 *A* solution *(or* table*) for a forward analysis* $\mathcal{A} = (D, t, I, \downarrow)$ *is a function* $S : PC \to D$*, such that* $S(pc_0) = I(pc_0)$*, and*

$$\forall pc \in PC, \bigsqcup_{(pc',pc) \in \mathcal{G}} t_{(pc',pc)}(S(pc')) \sqsubseteq S(pc)$$

A solution *(or* table*) for a backward analysis* $\mathcal{A} = (D, t, I, \uparrow)$ *is a function* $S : PC \to D$*, such that* $S(pc_{\mathrm{N}}) = I(pc_{\mathrm{N}})$*,* $S(pc_{\mathrm{E}}) = I(pc_{\mathrm{E}})$*, and*

$$\forall pc \in PC, S(pc) \sqsubseteq \bigsqcap_{(pc,pc') \in \mathcal{G}} t_{(pc,pc')}(S(pc')).$$

We will not delve in how to find solutions for an analysis. We refer the interested reader to [52].

To illustrate the combination of analysis and the VCgen, we will define a simple null-pointer analysis. We use a technique described in [28, 107] for defining domains for bytecode analysis, where the values stored in the stack are related to their meaning.

Example 1 *The null-pointer analysis $\mathcal{A}_{\mathsf{NP}} = (D_{\mathsf{NP}}, t_{\mathsf{NP}}, I_{\mathsf{NP}}, \downarrow)$ is defined as follows. The domain D_{NP} represents the operand stack and the local variables, and is defined by:*

$$\begin{array}{rcl} D_{\mathsf{NP}} & = & (\mathit{list}\ E)_{\bot}^{\top} \times (\mathit{Var} \to \mathit{NP}) \\ \mathit{NP} & = & \{\mathit{null}, \mathit{nonnull}\}_{\bot}^{\top} \\ E & ::= & \mathit{const}\ \mathit{NP} \mid \mathit{localvar}\ \mathit{Var} \end{array}$$

We define an interpretation function $[\![.]\!]$ that depends on a function $l : \mathit{Var} \to \mathit{NP}$, such that:

$$[\![\mathit{const}\ n]\!](l) = n$$
$$[\![\mathit{localvar}\ x]\!](l) = l(x),$$

Also, we have an update *function; given $e : E$, $n : \mathit{NP}$, and $l : \mathit{Var} \to \mathit{NP}$, the expression $[\![e = n]\!](l) : \mathit{Var} \to \mathit{NP}$ is a mapping that updates l using the fact that $e = n$. It is defined by:*

$$[\![\mathit{const}\ c = n]\!](l) = l$$
$$[\![\mathit{localvar}\ x = n]\!](l) = l[x \mapsto n]$$

The transfer functions, $t_{\mathsf{NP}(pc,pc')}(d)$ is defined by case analysis in the instruction at pc and in d. Some of the rules are:

- *if the instruction is Getfield, then*

$$\begin{array}{rcl} t_{\mathsf{NP}(pc,pcNext)}(e :: s, l) & = & (\mathit{const}\ \top :: s, [\![e = \mathit{nonnull}]\!](l)) \\ t_{\mathsf{NP}(pc,pcNull)}(e :: s, l) & = & (\mathit{nonnull} :: [], [\![e = \mathit{null}]\!](l)), \end{array}$$

  where pcNext *and* pcNull *are the successor program points corresponding to normal execution and exceptional execution respectively. Note that we update the local-variables map depending on whether the top of the stack contains a null or a non-null pointer.*

- *if the instruction is Return, then*

$$t_{\mathsf{NP}(pc,pc_{\mathrm{N}})}(v :: s, l) = (v :: s, l).$$

  Note that pc_{N} is the only successor of pc.

- *if the instruction is Invokevirtual, then*

$$\begin{array}{rcl} t_{\mathsf{NP}(pc,pcNext)}(args +\!\!+\ loc :: s, l) & = & (\mathit{const}\ \top :: s, [\![loc = \mathit{nonnull}]\!](l)) \\ t_{\mathsf{NP}(pc,pcExc)}(args +\!\!+\ loc :: s, l) & = & (\mathit{const}\ \mathit{nonnull} :: [], l) \\ t_{\mathsf{NP}(pc,pc_{\mathrm{E}})}(args +\!\!+\ loc :: s, l) & = & (\mathit{const}\ \mathit{nonnull} :: [], l), \end{array}$$

  where pcExc ranges over all exception handler whose range include pc. We define it this way because, without further analysis, we cannot know which exception can be thrown by a method. Hence, we assume that any exception in range can be thrown. Also, an Invokevirtual instruction has the node pc_{E} as successor, since the called method can throw an exception that can be uncaught in the current method.

- *if the instruction is Vload, then $t_{\mathsf{NP}(pc,\mathrm{pcNext})}(s, l) = (\mathit{localvar}\ x :: s, l)$*

An static analysis simulates the execution of a program in its domain. To prove that an analysis is sound, we need to prove that a step in the operational semantics, correspond to a transfer function in the domain. We define a *correctness relation* that relates states, with the elements of the domain of the analysis.

Definition 2.4.3 *A* correctness relation *for an analysis $\mathcal{A} = (D, t, I, f)$ is a relation $\_ \vdash \_ \subseteq State \times D$, such that the following holds:*

- *for all $d_1, d_2 \in D$, if $s \vdash d_1$ and $d_1 \sqsubseteq d_2$, then $s \vdash d_2$, and*
- *if $(\forall d \in D' \subseteq D, s \vdash d)$, then $s \vdash (\bigsqcap D')$.*

The relation $s \vdash d$ should be read as: d is a safe approximation of s.

Definition 2.4.4 *A static analysis $\mathcal{A} = (D, t, I, f)$ with correctness relation $\vdash$, is* sound *if for every solution S, the following holds: $(pc, s) \leadsto (pc', s')$ and $s \vdash S(pc)$, implies $s' \vdash S(pc')$.*

The usual way to prove that an analysis is sound is to prove that the transfer functions preserve the semantics. For a forward analysis, this means that if $(pc, s) \leadsto (pc', s')$ and $s \vdash d$, then $s' \vdash t_{(pc,pc')}(d)$. For a backward analysis, the transfer functions preserve the semantics if $(pc, s) \leadsto (pc', s')$ and $s \vdash t_{(pc,pc')}(d)$ implies $s' \vdash d$.

If we prove for a given analysis that the transfers functions preserve the semantics, then the soundness of the analysis follows from the properties of the correctness relation, and the definition of a solution.

Continuing with the examples, we define a correctness relation for the null-pointer analysis and the weakest precondition.

Example 2 *For the analysis defined in Example 1, we define a correctness relation, $\vdash_{\mathsf{NP}}$, by translating the elements of D_{NP} to expr, and using the validity relation of expr. First, we define the function $tr : V \times NP \to expr$, where*

$$\begin{aligned} tr(e, \bot) &= \texttt{False} \\ tr(e, \top) &= \texttt{True} \\ tr(e, null) &= \texttt{CompInt} = e\ \texttt{EConst null} \\ tr(e, nonnull) &= \texttt{CompInt} \neq e\ \texttt{EConst null} \end{aligned}$$

This function is extended to $\overline{tr} : D_{\mathsf{NP}} \to SLocalState \to expr$. For example, if $s = (h, os, l)$, then

$$\overline{tr}(localvar\ 0 :: [], [x \mapsto nonnull, y \mapsto \top])\ s = \\ tr(mkStget\ os\, 0, nonnull) \wedge tr(mkLget\ l\, 0, nonnull) \wedge tr(mkLget\ l\, 1, \top).$$

The correctness relation is defined as: $s \vdash_{\mathsf{NP}} d = s \models \overline{tr}(d)$. It can be shown that the transfer functions for this analysis preserve the semantics, and therefore, that the analysis is sound.

2.4.2 Combining static analyses with the VCgen

We show how the VCgen can use the results of the analysis to reduce the proof obligations. The main idea is to use the solution of the analysis as a parameter for the VCgen. When computing the function $\mathsf{swp_i}$ at a particular point pc, we can use the information given by the analysis at pc to remove some branch.

Remember that the function $\mathsf{swp_i}$ defined in 2.3 can be represented in the following way:

$$\mathsf{swp_i}\ pc\ s_0\ s = \bigwedge_{(pc,pc') \in \mathcal{G}} C_{(pc,pc')}(s) \Rightarrow P_{(pc,pc')}(\mathsf{swp_l}(pc, s_0, s'_{(pc,pc')})),$$

where the $C_{(pc,pc')}$ is a condition that needs to be satisfied by s for the program to go from pc to pc', $s'_{(pc,pc')}$ is the state after the execution of the instruction, and P is a predicate transformer.

Assume we have an analysis $\mathcal{A} = (D, t, I, f)$ with correctness relation $\vdash$, and a solution $S : PC \to D$. Further, assume we have a function $\gamma : D \to expr$ that translates the results of the analysis to expressions in the VCgen language that reference to the local state, with the following property: if $s \vdash d$, then $s \models \gamma(d)$.

We redefine the function $\mathsf{swp_i}$. The general form is now

$$\mathsf{swp_i}(pc, s_0, s) = \bigwedge_{(pc,pc')\in\mathcal{G}} F_{(pc,pc')}(s_0, s)$$

where the F is defined as

$$F_{(pc,pc')}(s_0, s) = \begin{cases} \texttt{True} & \text{if } \models \gamma(S(pc)) \Rightarrow \neg C_{(pc,pc')}(s) \\ C_{(pc,pc')}(s) \Rightarrow P_{(pc,pc')}(\mathsf{swp_l}(pc'), s_0, s) & \text{otherwise} \end{cases}$$

Intuitively, if we can infer $\neg C_{(pc,pc')}(s)$ from $S(pc)$, then the path going from pc to pc' cannot be taken at runtime, since taking this path would imply that the condition $C_{(pc,pc')}(s)$ is valid. In that case, the proof obligation corresponding to this branch can be removed, replacing it by `True`.

The condition $\models \gamma(S(pc)) \Rightarrow \neg C_{(pc,pc')}(s)$ may not be decidable; in that case we have to replace it with a decidable test, $\mathit{test}(S(pc), C_{(pc,pc')}(s))$, that is a sound approximation (i.e. if $\mathit{test}(S(pc), C_{(pc,pc')}(s))$, then $\models \gamma(S(pc)) \Rightarrow \neg C_{(pc,pc')}(s)$).

The definition of F depends on the domain of the analysis, so we will illustrate with the null-pointer analysis defined above.

Example 3 *To remove proof obligations using the null-pointer analysis, we look on the instructions that could generate a null-pointer exception. For instance, let us take Getfield. If the instruction is Getfield, and* $S(pc) = (e :: s, l)$*, then* $F_{(pc,pcNull)}$ *and* $F_{(pc,pcNext)}$ *are defined by:*

$$F_{(pc,pcNull)}(s_0, s) = \begin{cases} \texttt{True} & \mathit{if}\ [\![e]\!](l) = \mathit{nonnull} \\ C_{(pc,pcNull)}(s) \Rightarrow P_{(pc,pcNull)}(\mathsf{swp_l}(pcNull), s_0, s') & \mathit{otherwise} \end{cases}$$

$$F_{(pc,pcNext)}(s_0, s) = \begin{cases} \texttt{True} & \mathit{if}\ [\![e]\!](l) = \mathit{null} \\ C_{(pc,pcNext)}(s) \Rightarrow P_{(pc,pcNext)}(\mathsf{swp_l}(pcNext), s_0, s') & \mathit{otherwise} \end{cases}$$

This says that if the analysis guarantees that the top of the stack will contain a non-null pointer, then we do not need to check the branch corresponding to the null-pointer exception handler. In the same way, we can remove the proof obligation corresponding to normal execution if the analysis guarantees that the pointer is null.

A similar definition applies to other instructions such as Putfield, Vaload, Vastore, Invokevirtual, i.e. all instructions that take a pointer parameter from the stack, and throw a NullPointerException if the pointer is null.

2.4.3 Combining static analyses and specifications

The VCgen presented above generates fewer proof obligations by using static analysis to reduce the control flow graph. However, there are situations where the analysis cannot ensure enough information to make some reduction possible. Consider the following excerpt of Java bytecode:

$$\begin{array}{lll} pc_1 & \ldots & A(pc_1) = \texttt{invLget}\, x \neq \texttt{EConst null} \wedge \ldots \\ & \ldots & \\ pc_2 & \mathit{Vload}\ x & \\ pc_3 & \mathit{Getfield}\ f & \end{array}$$

Assume that the local variable x does not change between pc_1 and pc_2, and that the annotation table contains the assertion that x is not null at pc_1. Therefore, at pc_3, the *Getfield* instruction is accessing a non-null pointer. If the analysis is not able to ensure this, then the VCgen will generate two proof obligations. The one corresponding to exceptional execution is proved by contradiction using the assertion at pc_1. If there is more that one access to x such as the one at pc_3, the VCgen will generate two proof obligations for each access.

In this section, we propose a way to transfer the assertions contained in the specification to the domain of the analysis, so that the analysis can produce more accurate results. In the example above, if the information contained in $A(pc_1)$ is transferred, the analysis can propagate it to point pc_3, where it can ensure that the object accessed is non-null. Then, only one proof obligation would have been generated.

We will assume an annotated method m with specification $\mathcal{S}$ and an analysis $\mathcal{A} = (D, t, I, f)$. In order to translate the assertions contained in the specification to the domain of the analysis, we assume a function $\alpha : \mathit{expr} \to D$, with the following property: $s_0, s \models e \Rightarrow s \vdash \alpha(e)$.

We extend the annotation table A into a total function $\overline{A} : PC \to \mathit{expr}$, where we complete with the value `True` the elements that are not in the domain.

We redefine the meaning of a solution for the analysis, to use the specification. To differentiate from the previous definition, we call this *combined solution*, and refer to the previous as *simple solution*.

Definition 2.4.5 *A* combined solution *(or* combined table*) for a forward analysis $\mathcal{A} = (D, t, I, \downarrow)$ is a function $S : PC \to D$, such that $I(pc_0) \sqcap \alpha(Pre) \sqsubseteq S(pc_0)$ and*

$$\forall pc \in PC, \bigsqcup_{(pc,pc')\in\mathcal{G}} t_{(pc,pc')}(S(pc) \sqcap \alpha(\overline{A}(pc))) \sqsubseteq S(pc') \ .$$

A combined solution *(or* combined table*) for a backward analysis $\mathcal{A} = (D, t, I, \uparrow)$ is a function $A : PC \to D$, such that $S(pc_{\mathrm{N}}) \sqcap \alpha(Post_{Nrml}) \sqsubseteq I(pc_{\mathrm{N}})$, $S(pc_{\mathrm{E}}) \sqcap \alpha(Post_{Exc}) \sqsubseteq I(pc_{\mathrm{E}})$ and*

$$\forall pc \in PC, \alpha(\overline{A}(pc)) \sqcap S(pc) \sqsubseteq \bigsqcap_{(pc,pc')\in\mathcal{G}} t_{(pc,pc')}(S(pc')) \ .$$

Note that, since transfer functions and the meet operator ($\sqcap$) are monotone, any simple solution for the analysis is also a combined solution. To find combined solutions, we can use the same methods used to find simple solutions.

Again, we will exemplify the approach using the null-pointer analysis.

Example 4 *To define the function α for the analysis $\mathcal{A}_{\mathsf{NP}}$, we first define the function split : $\mathit{expr} \to$ list expr such that $\mathit{split}(e_1 \wedge e_2) = \mathit{split}(e_1) +\!\!+\ \mathit{split}(e_2)$, and $\mathit{split}(e) = e$ if e is not of the form $e_1 \wedge e_2$.*

Then α is defined as: $\alpha(e) = \mathit{filter}(\mathit{split}(e))$, where filter looks in the list produced by split for expressions of the form $\texttt{invSt}\,k \triangleq \texttt{EConst null}$, $\texttt{invSt}\,k \neq \texttt{EConst null}$, $\texttt{invLget}\,k \triangleq \texttt{EConst null}$, $\texttt{invLget}\,k \neq \texttt{EConst null}$, or their symmetric, and translate them to the domain D_{NP}. For instance, $\alpha(\texttt{invLget}\,0 \neq \texttt{EConst null} \wedge \texttt{EConst null} \triangleq \texttt{invSt}\,1) = (\mathit{const}\ \top :: \mathit{const}\ \mathit{null} :: [], [0 \mapsto \mathit{nonnull}])$.

2.4.4 Correctness of the VCgen revisited

We review the theorem stating correctness of the VCgen. The main difference is that, in this case, we have an additional hypothesis about the analysis, that is, that the solution found is sound.

The definitions of `certifiedMethod` and `certifiedProg` are the same as Sect. 2.3.6, except that they take a new parameter which is the result of the analysis (because $\mathsf{swp_i}$ and $\mathsf{swp_l}$ take this new parameter).

Theorem 2.4.6 (Correctness of the VCgen) *For each annotated program p, method m, and solution to an analysis S, such that the proof obligation generated by the VC generator is satisfied (i.e. a proof of $\mathit{interp}_{\mathrm{WP}}(\texttt{certifiedProg}\ p\ S)$ exists) and $p.MST(m) = (R, T, \mathcal{S})$ we have that if the proposition*

$$\mathit{interp}_{\mathrm{WP}}(\mathsf{swp_l}\ p\ m\ pc\ s_0\ s)$$

is valid, $s \vdash S(pc)$ is valid and the state (pc, s) evaluates to:

- *the state* (pc', s')*, then the proposition* $interp_{\mathrm{WP}}(\mathsf{swp_l}\ p\ m\ pc'\ s_0\ s')$ *is valid, and* $s' \vdash S(pc')$ *is valid,*
- *the return state* (pc_E, s) *or* (pc_N, s)*, then the proposition* $interp_{\mathrm{WP}}(\texttt{Post}\ m\ s_0\ (h, rv))$ *is valid (where* h *is the heap of state* s *and* rv *the return value).*

Note that we have to prove two properties. One is the fact that the formula $interp_{\mathrm{WP}}(\mathsf{swp_l}\ldots)$ remains valid for a step in the operational semantics. The other is that the relation $\vdash$ remains valid for the same step. These proofs can be done independently. The latter corresponds to the fact that the analysis is sound. In order to prove it, we need to prove that $\alpha(\bar{(}A)(pc)$ is valid. This is implied by $interp_{\mathrm{WP}}(\mathsf{swp_l}\ p\ m\ pc\ s_0\ s)$.

The former correspond to proving the soundness of the WP and uses the fact that the analysis is sound in the cases where a proof obligation is removed (the soundness of the analysis guarantees that the removal is safe).

2.4.5 Coq Implementation

In this section, we are going to give some details of the Coq implementation of the VCgen described above. The description is divided in two parts, the first concerns the null-pointer analysis, and the second the VCgen proper.

Null-pointer analysis. The implementation of the null-pointer analysis, consists in the definition of the analysis domain, the transfer functions, and a predicate to check that a solution is valid. Solutions are defined by a function that returns an abstract memory for each program point

$$\texttt{Definition Solution} := \texttt{PC} \rightarrow \texttt{abstract\_memory}$$

The predicate for checking solution is

$$\texttt{soluce} : \mathit{AnnotProg} \rightarrow \mathcal{M} \rightarrow \texttt{Solution} \rightarrow \mathit{Prop}$$

whose definition is a direct translation of the definition of a combined solution (Def. 2.4.5). To be able to efficiently check solutions, we have defined a function

$$\texttt{check\_soluce} : \mathit{AnnotProg} \rightarrow \mathcal{M} \rightarrow \texttt{Solution} \rightarrow \texttt{bool}$$

together with a lemma showing that the function is correct:

$$\texttt{Lemma check\_soluce\_correct} : \texttt{forall}\ p\ m\ SL, \texttt{check\_soluce}\ p\ m\ SL = \texttt{true} \rightarrow \texttt{soluce}\ p\ m\ SL.$$

Using this lemma, verifying a solution reduces to a conversion test, that is, checking that the function `check_soluce` returns `true`, which can be done very efficiently. Also, this means that the size of the proof for checking a solution is the same for all programs.

Hybrid VCgen. Based on the null-pointer analysis, we have defined another VCgen, that uses the results of the analysis as a way to reduce proof obligations as described above. To define it, we modified the deep VCgen in three ways. First, the method specification table now contains a new component that assigns the solution of the analysis to each method.

Second, the function $\mathsf{swp_i}$ needs to be modified to use the results of the analysis. This means modifying the treatment of instructions that can raise a *NullPointerException*, so that the corresponding branch is deleted if the solution of the analysis ensures that no such exception can occur at runtime. For instance, the definition of $\mathsf{swp_i}$ for the instruction *Getfield* is as follows:

$$\begin{aligned}
\mathsf{swp_i}\ pc\ s_0\ (h, os, l) = {} & \texttt{let}\ loc = mkStget\ os\ 0\ \texttt{in} \\
& \wedge loc \neq \texttt{EConst null} \Rightarrow \mathsf{swp_{next}}\ pc\ s_0\ (h, mkStPush\ (mkHget\ h\ (\texttt{DynamicField}\ loc\ f)\ os, l) \\
& \quad \texttt{if}\ (isNotNull\ (SL\ pc)) \\
& \quad \texttt{then True} \\
& \quad \texttt{else}\ (loc = \texttt{EConst null} \Rightarrow \mathsf{swp_{Null}}\ pc\ h\ l\ s_0)
\end{aligned}$$

where *SL* is the solution of the analysis, and the function *isNotNull* checks if *SL pc* (the abstract memory at that point in the program) states that the top of the stack contains a non-null pointer. If that is the case, then remove the proof obligation by replacing it with `True`.

Finally, we modified the definition of `certifiedProg`, to check that the solutions of the analysis are correct.

2.4.6 Example Program

Continuing with the example of Sect. 2.3.4, in this section we show how the verification of this example proceeds. We are not interested in the verification itself of this program, since an example of program verification using the shallow VCgen was given in Deliverable D3.1. Instead, what concerns us here is to show, through an example, how the combination of VCgen and static analyses helps in the verification.

First, we will use the deep VCgen to prove that the specification is satisfied, without using the null-pointer analysis. As a result, there are seven proof obligations. One is related to the normal postcondition, and the other six are related to the instructions that can generate a null-pointer exception. Since pointer `this` is non-null, and the precondition states that the parameter is non-null, then, we can prove that, in fact, there can be no exception in this method. The same proof obligations are generated using the shallow VCgen.

Now we will use the null-pointer analysis, however, we do not give the method specification as input to the analysis. Therefore, the initial value states that pointer `this` (index 0 in the local variables) is non-null, and gives no information about the parameter (index 1 in the local variables); the initial stack is empty:

$$([], \{0 \mapsto \mathit{nonnull}, 1 \mapsto \top\})$$

Using this initial value, we can remove four proof obligations corresponding to the access to the pointer `this` (at program points 2, 10, 15 and 23). However, the analysis is not able to remove the proof obligations corresponding to the access to the parameter (at program points 6 and 19), since the initial solution doesn't state any information about it.

The precondition of the method states that the parameter is non-null. With an analysis that uses the specification to improve its result, the initial solution is now:

$$([], \{0 \mapsto \mathit{nonnull}, 1 \mapsto \mathit{nonnull}\})\,,$$

showing that both values in the local variables array are non-null. With this initial value, the analysis is able to ensure that all pointer accesses will be done to non-null variables. In that case, we can remove *all* proof obligations related to exceptional termination, and we are left with only one proof obligation that correspond to the normal postcondition.

2.4.7 Hybrid certificates

So far, we presented a VCgen that improves the usability and performance of the VCgen presented in Deliverable D3.1 by using a combination of deep embedding, and static analyses. Now we present the certificates for this VCgen, which we call *hybrid certificates.*

In the typical PCC architecture, the producer runs the VCgen on the annotated code. This generates proof obligations, whose proof provides the certificate that is packaged along with the code and sent to the consumer. For the VCgen described in the previous sections, this framework largely applies. The difference lies in the generation of proof obligations. The analyses are performed on the code. For this stage, any fixpoint algorithm can be used to generate the results of the analysis. The algorithm itself does not need to be verified, since we can check that the results given are correct [28].

The results of these analyses are then given to the VCgen, that returns the proof obligations. These can be proved by automatic methods or in a proof assistant (Coq in our case). The certificate given to the consumer consists on the proofs obtained and the results of the analysis.

Checking the certificate, on the consumer side, consists of three stages. First, the results of the analyses are checked. This involves a simple procedure that can be done very efficiently in one pass through the code [?]. Second, once the results are checked, they are given to the VCgen that generates the proof obligations. Third, the proofs given as part of the certificate are checked to correspond with the obligations generated by the VCgen. If all the checking goes well, the code can be safely executed.

2.4.8 Discussion

We have shown in this section a VCgen that improves, in terms of usability and performance, the VCgen presented in Deliverable D3.1, using a deep embedding and static analyses.

Using a deep embedding of the VCgen allows us to reduce significantly the time needed to check a proof, by using reflection techniques. Another advantage of a deep embedding is the possibility to manipulate and simplify the proof obligations generated by the VCgen. This possibility is considered in [109], where the authors compare shallow and a deep VCgens.

We use static analysis to reduce the number of proof obligations. This is an important factor because we want to simplify the task of the developer, specially when using an interactive proof assistant like Coq. The use of abstract interpretation as a tool to verify safety policies in PCC has been proposed by Albert, Puebla and Hermenegildo in their *Abstraction-Carrying Code* (ACC) framework [?], where abstract interpretation is used to represent safety policies. The abstraction of a program is the certificate sent to the consumer alongside the code. We do not use analysis to express safety policies, but to reduce the control flow graph of a program. Albert et al. developed a technique to compress certificates for ACC in [9]. The main idea is to remove redundant information that can be easily reconstructed in one pass through the code. See Sect. 3.1 for a description of this technique and Sect. 3.1.4 for a discussion on certificate compression. The different approaches to compression described there can be readily applied in our case to compress the results of the analyses.

We have exemplified the approach with a simple null-pointer analysis. We have chosen this type of analysis, because many instructions in the JVM can throw null-pointer exceptions, which allows for large reductions in the proof obligations. A recent study by Chalin and James [34] shows that in 2/3 of the cases, reference variables are meant to be non-null (based on design intent). However, other analysis can be useful to reduce proof obligations. Obvious candidates are interval analysis used for array-bound checking and escaping-exception analysis.

2.5 Preservation of proof obligations for hybrid verification methods

Program verification, and in particular deductive program verification, is traditionally applied to source code. Therefore, it is of interest to develop methods to transfer evidence from source code verification to the code consumers.

This section extends preservation of proof obligations to hybrid verification methods. For concreteness, we consider a small imperative language with arrays, and we focus on a hybrid method based on a generic numerical analysis [76, 29] and that can be instantiated to several numeric domains, including polyhedra.

We first define a hybrid verification method in which programs are subjected to static analysis, and then to verification condition generation. The VCgen exploits the information of the analysis in two useful ways: on the one hand, verification conditions that originate from spurious edges in the control-flow graph are discarded: more precisely, the VCgen ignores the case of out-of-bound accesses whenever the analysis ensures that accesses are within bounds. This leads to fewer, smaller verification conditions. Furthermore, the VCgen adds the results of the analysis as additional assumptions to help the user prove the verification conditions. This is particularly useful for the relational analyses considered as they can provide part of the invariants required to prove programs correct.

Then, we prove preservation of proof obligations using the techniques of Barthe et al. [23]. The proof relies on knowing that the solutions of the analysis are preserved by compilation. Although analyzing compiled programs is known to be less precise than analyzing source programs, as stated by Logozzo and

Fähndrich [68], we achieve preservation of solutions by defining at bytecode level an analysis that performs a symbolic execution of stacks [111, 110, 29].

2.5.1 Proof Carrying Code and Hybrid Methods

Hybrid methods aim to provide a tight integration of type systems and logical methods. There are two approaches to hybrid program verification. In the explicit approach, the user provides safety annotations that are used by the verification condition generator, and checked by a annotation checker. In contrast, the implicit approach advocates that the annotations are inferred by a static analyzer, and then used by verification condition generation. Both approaches are used (sometimes in conjunction) in deductive program verification, as well as in type-based analyses.

For the hybrid method developed in this paper, program verification proceeds as follows: first, the annotated program is subjected to the static analysis, and a solution is computed (note the analysis does not take advantage of the annotations). Then, the verification condition generator uses the solution to compute a smaller set of proof obligations, which must then be discharged interactively. On the consumer side, the program arrives packaged with the solution of the analysis, the program annotations, and the certificate, and the checking proceeds in four steps; first, one checks the correctness of the analysis. The remaining three steps are as in logic-based PCC.

2.5.2 Setting

This section introduces the source language (an imperative language with arrays of integers), the target language (a stack-based language with jumps), and the compiler.

We assume given two disjoint sets V_s of scalar variables and V_a of array variables, and let V denote $V_s + V_a$. Each variable in V_a has an associated size. Furthermore, we assume given two sets V_s^{old} and V_a^{old} in 1-1 correspondence with V_s and V_a, which are used to store initial values. We also consider a special variable res, which is used to represent the value of the program result. Finally, we assume given a set $\mathbf{Lab} \subset \mathbb{N}$ of labels.

Source Language

Programs are defined as commands, and are decorated with labels in order to express analysis results:

$$\begin{array}{rl} e ::= & e \ \mathtt{op} \ e \mid n \mid x \mid a[e] \\ c ::= & \mathtt{Skip} \mid [x{:=}e]^k \mid [a[e]{:=}e]^k \mid c; c \mid [\mathtt{return}\ e]^k \\ & \mid \mathtt{if}\ [e \bowtie e]^k\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \mathtt{while}\ [e \bowtie e]^k\ \mathtt{do}\ c \end{array}$$

where x, a, n and k respectively range over V_s, V_a, $\mathbb{Z}$ and $\mathbf{Lab}$, op ranges over (binary) arithmetic operations, and $\bowtie$ over arithmetic comparisons. We assume that labels occur at most once in commands.

The semantics of source programs is formalized by a small-step transition relation between states. States may be intermediate, in which case they consist of a statement and of a memory, or final, in which case they consist of a memory, and possibly a tag to denote abnormal termination. Memories are modeled as pairs of mappings respectively from variables to values and from arrays to indices to values. We assume that each array a comes equipped with its size $|a|$ and define the semantic domains of the source language as follows:

$$\begin{array}{rcl} \mathit{VMem} & = & V_s \to \mathbb{Z} \\ \mathit{AMem} & = & \Pi a \in V_a.\ \{i \mid 1 \leq i \leq |a|\} \to \mathbb{Z} \\ \mathit{Mem} & = & \mathit{VMem} \times \mathit{AMem} \\ \mathit{State}_I^s & = & \mathit{Stmt} \times \mathit{VMem} \times \mathit{AMem} \\ \mathit{State}_F^s & = & \mathit{VMem} \times \mathit{AMem} \times (\mathbb{Z} + \{\mathbf{AOB}\}) \\ \mathit{State}^s & = & \mathit{State}_I^s + \mathit{State}_F^s \end{array}$$

The operational semantics of programs is standard and, thus, omitted. (See the next subsection for the semantics of instructions that manipulate arrays.)

$$
\begin{array}{l}
\llbracket n \rrbracket_{\mathrm{e}} = \texttt{push}\ n \\
\llbracket x \rrbracket_{\mathrm{e}} = \texttt{load}\ x \\
\llbracket x[e] \rrbracket_{\mathrm{e}} = \llbracket e \rrbracket_{\mathrm{e}};\ \mathsf{aload}\ x \\
\llbracket e_1\ \texttt{op}\ e_2 \rrbracket_{\mathrm{e}} = \llbracket e_2 \rrbracket_{\mathrm{e}};\ \llbracket e_1 \rrbracket_{\mathrm{e}};\ \texttt{prim op}
\end{array}
$$

$$
\begin{array}{l}
\llbracket [x{:=}e]^k \rrbracket = k : \llbracket e \rrbracket_{\mathrm{e}}; \texttt{store}\ x \\
\llbracket [a[e_1]{:=}e_2]^k \rrbracket = k : \llbracket e_2 \rrbracket_{\mathrm{e}};\ \llbracket e_1 \rrbracket_{\mathrm{e}};\ \texttt{astore}\ a \\
\llbracket s1; s2 \rrbracket = \llbracket s_1 \rrbracket;\ \llbracket s_2 \rrbracket \\
\llbracket [\texttt{return}\ e]^k \rrbracket = k : \llbracket e \rrbracket_{\mathrm{e}};\ \texttt{return} \\
\llbracket [\texttt{Skip}]^k \rrbracket = k : \texttt{nop} \\
\llbracket \texttt{if}\ [e_1 \bowtie e_2]^k\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rrbracket = k : \llbracket e_2 \rrbracket_{\mathrm{e}}; \llbracket e_1 \rrbracket_{\mathrm{e}};\ \texttt{cjmp} \bowtie k_1; k_2 : \llbracket s_2 \rrbracket; \texttt{jmp}\ l; k_1 : \llbracket s_1 \rrbracket \\
\quad \text{where } k_1 = \mathit{init}(s_1) = k_2 + |\llbracket s_2 \rrbracket| + 1 \\
\qquad\qquad\ \ k_2 = \mathit{init}(s_2) = k + |\llbracket e_2 \rrbracket_{\mathrm{e}}; \llbracket e_1 \rrbracket_{\mathrm{e}}| + 1 \\
\qquad\qquad\quad l = k_1 + |\llbracket s_1 \rrbracket| \\
\llbracket \texttt{while}\ [e_1 \bowtie e_2]^k\ \texttt{do}\ s \rrbracket = k : \llbracket e_2 \rrbracket_{\mathrm{e}}; \llbracket e_1 \rrbracket_{\mathrm{e}}; \texttt{cjmp} \bowtie k_1; \texttt{jmp}\ l; k_1 : \llbracket s \rrbracket; \texttt{jmp}\ k \\
\quad \text{where } \ k_1 = k + |\llbracket e_2 \rrbracket_{\mathrm{e}}| + |\llbracket e_1 \rrbracket_{\mathrm{e}}| + 2 \\
\qquad\qquad\quad l = k_1 + |\llbracket s \rrbracket| + 1
\end{array}
$$

Figure 2.6: Compiler

Bytecode Language

A bytecode program is defined as a list of instructions. Instructions either manipulate the memory that stores the values of variables and the contents of arrays, or manipulate the operand stack, or perform a conditional or unconditional jump. The set of instructions is defined by the grammar

$$
\begin{array}{rl}
\mathit{ins} ::= & \texttt{prim op} \mid \texttt{push}\ v \mid \texttt{load}\ x \mid \texttt{store}\ x \mid \texttt{return} \\
& \mid \texttt{aload}\ a \mid \texttt{astore}\ a \mid \texttt{cjmp} \bowtie l \mid \texttt{jmp}\ l \mid \texttt{nop}
\end{array}
$$

We denote by $\dot{p}[l]$ the instruction at position l of a bytecode program $\dot{p}$. The semantics of bytecode programs is formalized using a transition relation between states. States may either be intermediate or final; intermediate states consist of a program counter, an operand stack, that stores the results of intermediate computations, and a memory. The semantic domains of the bytecode language are defined as follows, where we implicitly assume that the program counter is within the bounds of programs:

$$
\begin{array}{rcl}
\mathit{Stack} & = & \mathbb{Z}^{\star} \\
\mathit{State}^b_I & = & \mathbb{N} \times \mathit{VMem} \times \mathit{AMem} \times \mathit{Stack} \\
\mathit{State}^b_F & = & \mathit{VMem} \times \mathit{AMem} \times (\mathbb{Z} + \{\mathbf{AOB}\}) \\
\mathit{State}^b & = & \mathit{State}^b_I + \mathit{State}^b_F
\end{array}
$$

The operational semantics of programs is standard. We only provide the operational semantics of the instructions `aload` and `astore`; these instructions may cause abrupt termination if array accesses are out-of-bound. The rules are given in Figure 2.7, where we use the notation $[f \mid s \to r]$ to refer the function that is identical to f everywhere except in r that returns s, for any sets R and S and any function $f : R \to S$.

Compiler

The compiler is standard, and defined in Figure 2.6; we use the function $\mathit{init} : \mathbf{Stm} \to \mathbf{Lab}$ to associate to each statement its initial label. We assume *label compatibility*, i.e. that the label of a source statement is the same as the label of the program point for its compilation.

Throughout the rest of the paper we let P be a source program, and the bytecode program $\dot{p}$ the result of the compilation of program P.

$$\frac{P[i] = \mathtt{aload}\ a \qquad 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: s\rangle \rightsquigarrow \langle i+1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, \rho_{\mathsf{a}}\ a\ n :: s\rangle} \qquad \frac{P[i] = \mathtt{aload}\ a \qquad \neg\ 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: s\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, \mathsf{AOB}\rangle}$$

$$\frac{P[i] = \mathtt{astore}\ a \qquad 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: v :: s\rangle \rightsquigarrow \langle i{+}1, \rho_{\mathsf{v}}, [\rho_{\mathsf{a}} \mid a \to [\rho_{\mathsf{a}}\ a \mid n \to v]], s\rangle} \qquad \frac{P[i] = \mathtt{astore}\ a \qquad \neg\ 0 \le n < |a|}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: v :: s\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, \mathsf{AOB}\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n : s\rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n\rangle}$$

Figure 2.7: Semantics of bytecode (excerpts)

2.5.3 Preservation of solutions

It is folklore that compilation potentially yields a loss of precision for relational analyses. The purpose of this section is to show that solutions of abstract interpretations are preserved by compilation, provided one uses symbolic expressions, as done in [111, 110, 29], to mitigate the presence of the operand stack and to recover the loss of precision incurred by compilation.

Symbolic Expressions

Expressions and guards serve as the interface with the numerical relational domain in the analysis for bytecode. Below we let x range over V.

$$\begin{array}{rl} \mathit{Expr} \ni e ::= & n \mid x \mid x[e] \mid ? \mid ?[e] \mid e\ \mathsf{op}\ e \quad x \in V \\ \mathit{Guard} \ni t ::= & e \bowtie e \end{array}$$

The expression ? represents an unknown value; therefore, expressions are interpreted as sets of possible values. Formally, the semantics $[\![e]\!]_\rho$ and $[\![t]\!]_\rho$ of expressions with respect to an environment $\rho = \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}\rangle$ are defined by the clauses:

$$\begin{array}{rcl} [\![n]\!]_\rho & = & \{n\} \\ [\![x]\!]_\rho & = & \rho_{\mathsf{v}}\ x \\ [\![?]\!]_\rho & = & \mathbb{Z} \\ [\![?[e]]\!]_\rho & = & \mathbb{Z} \\ [\![x[e]]\!]_\rho & = & \{\rho_{\mathsf{a}}\ x\ v \mid v \in [\![e]\!]_\rho\} \\ [\![e_1\ \mathbf{op}\ e_2]\!]_\rho & = & \{n_1\ \mathbf{op}\ n_2 \mid n1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho\} \\ [\![e_1 \bowtie e_2]\!]_\rho & \iff & \exists\ n_1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho \bullet n_1 \bowtie n_2 \end{array}$$

Note that the expression ? is not required for analyzing bytecode programs that are achieved by compilation of the source program, since the stack is empty after storing a value in an array. However, it provides more precision when dealing with programs that are not obtained by compilation.

Abstract domain

Following Miné [76], we assume given an abstract numerical domain interface, which can be instantiated with standard relational abstract domains. The interface consists of a domain $\mathbb{D}$ equipped with a partial order $\sqsubseteq\ \subseteq \mathbb{D} \times \mathbb{D}$, meet and join operators $\sqcap, \sqcup : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, a least element $\bot$ and a greater element $\top$. We also assume given abstract assignment functions $[\![x{:=}e]\!]^\sharp, [\![x[e_1]{:=}e_2]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, and a function $\mathit{assume}^\sharp$ that maps guards to abstract elements.

Finally, we assume given a monotone concretization function $\gamma : \mathbb{D} \to\ \mathcal{P}(\mathit{VMem} \times \mathit{AMem})$ mapping abstract elements to sets of environments in $\mathit{VMem} \times \mathit{AMem}$, and satisfying the properties in Figure 2.8.

We define the abstract test $[\![t]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ of a guard $t \in \mathit{Guard}$ by $[\![t]\!]^\sharp(l^\sharp) = \mathit{assume}^\sharp(t) \sqcap l^\sharp$.

$$\begin{array}{rcl}
\gamma(d_1 \sqcap d_2) & \supseteq & \gamma(d_1) \cap \gamma(d_2) \\
\gamma(d_1 \sqcup d_2) & \supseteq & \gamma(d_1) \cup \gamma(d_2) \\
\gamma([\![x{:=}e]\!]^\sharp(d)) & \supseteq & \{\langle \rho_{\mathsf{v}}[x \mapsto\ v], \rho_{\mathsf{a}}\rangle \mid \rho \in \gamma(d) \wedge \quad v \in [\![e]\!]_\rho\} \\
\gamma([\![x[e_1]{:=}e_2]\!]^\sharp(d)) & \supseteq & \{\langle \rho_{\mathsf{v}}, [\rho_{\mathsf{a}} \mid x \rightarrow [\rho_{\mathsf{a}}\ a \mid v_1 \rightarrow v_2]]\rangle \mid \rho \in \gamma(d) \wedge v_1 \in [\![e_1]\!]_\rho \wedge v_2 \in [\![e_2]\!]_\rho\} \\
\gamma(assume^\sharp(t)) & \supseteq & \{\rho \mid [\![t]\!]_\rho\}
\end{array}$$

where $\rho = \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}} \rangle$.

Figure 2.8: Requirements over function γ

$$\frac{stm \notin \{\texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ \texttt{while}\ t\ \texttt{do}\ c,\ s_1\,;\, s_2, \texttt{return}\ e\}}{Loc \vdash \{Loc(i)\}\ [stm]^i\ \{F_{stm}(Loc(i))\}} \qquad \frac{}{Loc \vdash \{Loc(i)\}\ [\texttt{return}\ e]^i\ \{\bot\}}$$

$$\frac{Loc \vdash \{[\![t]\!]^\sharp(Loc(i))\}\ s_1\ \{l_1^\sharp\} \qquad Loc \vdash \{[\![\neg t]\!]^\sharp(Loc(i))\}\ s_2\ \{l_2^\sharp\}}{Loc \vdash \{Loc(i)\}\ \texttt{if}\ [t]^i\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{l_1^\sharp \sqcup l_2^\sharp\}}$$

$$\frac{Loc \vdash \{[\![t]\!]^\sharp(Loc(i))\}\ s\ \{l^\sharp\} \qquad l^\sharp \sqsubseteq Loc(i)}{Loc \vdash \{Loc(i)\}\ \texttt{while}\ [t]^i\ \texttt{do}\ s\ \{[\![\neg t]\!]^\sharp(Loc(i))\}} \qquad \frac{Loc \vdash \{l^\sharp\}\ s_1\ \{l_1^\sharp\} \qquad Loc \vdash \{l_1^\sharp\}\ s_2\ \{l_2^\sharp\}}{Loc \vdash \{l^\sharp\}\ s_1\,;\, s_2\ \{l_2^\sharp\}}$$

$$\frac{Loc \vdash \{\top\}\ P\ \{l^\sharp\}}{Loc \vdash P}$$

$$\text{where}\quad F_{stm}(l^\sharp) = \begin{cases} l^\sharp & \text{if}\ \ stm = \texttt{Skip} \\ [\![x{:=}e]\!]^\sharp(l^\sharp) & \text{if}\ \ stm = x{:=}e \\ [\![a[e_1]{:=}e_2]\!]^\sharp(l^\sharp) & \text{if}\ \ stm = a[e_1]{:=}e_2 \end{cases}$$

Figure 2.9: Definition of the constraint system for the source code analysis.

Source Code Analysis

The source code analysis is specified by abstract transfer functions that map elements of the abstract domain into elements of the abstract domain.

Definition 2.5.1 (Abstract Domain for High-Level) *A result of the analysis for the source program P is described by a mapping Loc in the lattice $State^\sharp = \mathbf{Lab} \rightarrow \mathbb{D}$.*

Definition 2.5.2 (Solution) *A mapping Loc for the source program P is a solution of the analysis if it verifies the constraint system defined in Figure 2.9, i.e. $Loc \vdash P$ holds.*

Byte Code Analysis

As for the source code analysis, the bytecode analysis is defined by abstract transfer functions that map abstract states into abstract states. In this case, the abstract states are pairs of the form $(s^\sharp, l^\sharp)$ where $l^\sharp$ is an element of the abstract domain, and the list of symbolic expressions $s^\sharp$ abstracts the operand stack. The symbolic abstract domain for stacks is $Expr^\star$, where for any set A, $A^\star$ denotes the domain of lists with elements in A. The set of variables considered by the bytecode analysis is the same as in the source code analysis.

Definition 2.5.3 (Bytecode Abstract Domain) *A result of the analysis for $\dot{p}$ is described by a mapping $\dot{loc}$ in the lattice*

$$\dot{state}^\sharp = \mathbf{Lab} \rightarrow (Expr_L^\star \times \mathbb{D})$$

$instr$	$\dot{f}_{instr}$	$instr$	$\dot{f}_{instr}$
`prim` op	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (\llcorner e_1\ \underline{op}\ e_2 \lrcorner :: s^\sharp, l^\sharp)$	`push` n	$(s^\sharp, l^\sharp) \to (n :: s^\sharp, l^\sharp)$
`load` r	$(s^\sharp, l^\sharp) \to (\llcorner r \lrcorner :: s^\sharp, l^\sharp)$	`store` r	$(e :: s^\sharp, l^\sharp) \to (s^\sharp[?/r], [\![r{:=}e]\!]^\sharp(l^\sharp))$
`aload` a	$(e :: s^\sharp, l^\sharp) \to (\llcorner a[e] \lrcorner :: s^\sharp, l^\sharp)$	`astore` a	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (s^\sharp[?/a], [\![a[e_1]{:=}e_2]\!]^\sharp(l^\sharp))$
`nop`	$(s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$		

$$\frac{\mathsf{Instr} \notin \{\,\texttt{jmp}\ i', \texttt{cjmp} \bowtie i', \texttt{return}\} \qquad \dot{f}_{instr}(l\dot{o}c(i)) \sqsubseteq l\dot{o}c(i+1)}{l\dot{o}c \vdash i : \mathsf{Instr}}$$

$$\frac{}{l\dot{o}c \vdash i : \texttt{return}} \qquad \frac{l\dot{o}c(i) \sqsubseteq l\dot{o}c(j)}{l\dot{o}c \vdash i : \texttt{jmp}\ j}$$

$$\frac{l\dot{o}c(i) = (e_1 :: e_2 :: s^\sharp, l^\sharp) \qquad (s^\sharp, [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(l^\sharp)) \sqsubseteq l\dot{o}c(i+1) \qquad (s^\sharp, [\![e_1 \bowtie e_2]\!]^\sharp(l^\sharp)) \sqsubseteq l\dot{o}c(j)}{l\dot{o}c \vdash i : \texttt{cjmp} \bowtie j}$$

$$\frac{\top \sqsubseteq l\dot{o}c(0) \qquad \forall i \in \mathsf{dom}(\dot{p}) \bullet\ l\dot{o}c \vdash i : \dot{p}[i]}{l\dot{o}c \vdash \dot{p}}$$

Figure 2.10: Definition of the constraint system for the byte code analysis.

An analysis result is a solution of the analysis if it satisfies the constraint system associated to each program. The constraint system is defined in Figure 2.10. For instructions other than branching or return instructions, the constraint is defined by partial transfer functions in $Expr^\star \times \mathbb{D} \rightharpoonup (Expr^\star \times \mathbb{D})$, most of them defined as a symbolic execution affecting the abstract representation of the operand stack.

Definition 2.5.4 (Solution) *A mapping $l\dot{o}c$ for the bytecode program $\dot{p}$ is a solution of the analysis if it satisfies the constraint system of Figure 2.10, i.e. if $l\dot{o}c \vdash \dot{p}$ holds.*

Preservation of Solutions

We define first the compilation of a source code analysis solution and then show that it is a solution for the byte code analysis. For notational convenience, we denote by $\dot{f}_{s_1;\cdots;s_n}(s^\sharp, l^\sharp)$ the composition $\dot{f}_{s_n}(\cdots(\dot{f}_{s1}(s^\sharp, l^\sharp))\cdots)$, where $s_1;\cdots;s_n$ is a sequence of byte code instructions. Let $\mathsf{succ}(l)$ denotes the set of successors of a label l, e.g. $\mathsf{succ}(l) = \emptyset$ and $\mathsf{succ}(l) = \{l+1, l'\}$ respectively for $\dot{p}[l] = \texttt{return}$ and $\dot{p}[l] = \texttt{cjmp} \bowtie l'$. The set $\mathsf{pred}(l)$ is defined as $\{l' \mid l \in \mathsf{succ}(l')\}$.

Remark 2.5.5 *For each byte code program $\dot{p}$, we can extract from the previous constraint system a set of transfer functions $(\dot{g}_{i,j})_{(i,j)\in\mathbf{Lab}^2}$ such that $l\dot{o}c \vdash \dot{p}$ if and only if $\bigsqcup_{k'\in\mathsf{pred}(k)} \dot{g}_{k',k}(l\dot{o}c(k')) \sqsubseteq l\dot{o}c(k)$ for all $k \in \mathsf{dom}(\dot{p})$.*

We can extend a partial function $l\dot{o}c_{partial} \in st\dot{a}te^\sharp$ to a total function $l\dot{o}c$ on $\mathsf{dom}(\dot{p})$ if we set

$$l\dot{o}c(k) = \begin{cases} l\dot{o}c_{partial}(k) & \text{if } k \in \mathsf{dom}(l\dot{o}c_{partial}) \\ \bigsqcup_{k'\in\mathsf{pred}(k)} \dot{g}_{k',k}(l\dot{o}c(k')) & \text{otherwise} \end{cases}$$

This definition only makes sense if, by considering the control flow graph of $\dot{p}$ whose edges are

$$\{(i,j) \mid i \in \mathsf{dom}(\dot{p}) \wedge j \in \mathsf{succ}(i)\}$$

every loop contain a label in $\mathsf{dom}(l\dot{o}c_{partial})$. We refer to the function $l\dot{o}c$ as the completion of *the partial function $l\dot{o}c_{partial}$.*

Definition 2.5.6 (Compiled analysis results) *Given an analysis result Loc for the program P, an analysis result compiled from Loc is the completion of the function* $\dot{loc}_{partial}$ *defined on each* $k \in \mathsf{dom}(Loc)$ *by* $\dot{loc}_{partial}(k) = ([\,], Loc(k))$*.*

This definition can be shown to be well defined from the facts that Loc annotates every loop in P and each loop in the control flow graph of $\dot{p}$ contains a label of a loop in P.

Lemma 2.5.7 *Let* $\dot{p}_1$*,* $\dot{p}_2$ *and e s.t.* $\dot{p} = \dot{p}_1 :: l : [\![e]\!]_{\mathrm{e}} :: l' : \dot{p}_2$*. Then,* $\dot{loc}(l') = f_{i_1;\ldots;i_k}(s^\sharp, l^\sharp) = (e :: s^\sharp, l^\sharp)$ *where* $(s^\sharp, l^\sharp) = \dot{loc}(l)$ *and* $[i_1; \ldots; i_k] = [\![e]\!]_{\mathrm{e}}$*.*

The following lemma states the main result of this section: compilation preserves analysis solutions.

Lemma 2.5.8 *If Loc is s.t.* $Loc \vdash P$*, then the analysis result* $\dot{loc}$ *compiled from Loc is s.t.* $\dot{loc} \vdash \dot{p}$*, i.e. it is a solution of the bytecode analysis.*

2.5.4 Preservation of proof obligations

In this section we define two verification frameworks, respectively for source programs and for unstructured bytecode of previous sections. As a specification language we consider first order formulae, namely the domain of assertions $\mathcal{A}$. The validity of an assertions in a particular execution state $\eta \in State^s$ is standard. In particular, an assertion that contains the expression $a[e]$ is invalid in those execution states in which e is out of the bounds of the array a.

We consider as a program specification a tuple $(Pre, \mathsf{annot}, Post, \chi)$, where the assertion Pre is a precondition, $Post$ and $Post$ are respectively normal and abnormal postconditions, and the partial function $\mathsf{annot} : \mathbf{Lab} \rightharpoonup \mathcal{A}$ maps program labels to internal points specifications. The special variable res may only occur in $Post$, and Pre only refers to variables from V. When specifying a bytecode program, assertions may refer to the special variable os representing the operand stack.

We say that a program satisfies the specification $(Pre, \mathsf{annot}, Post, \chi)$, if every execution starting in a state that satisfies Pre only reaches normal final states satisfying $Post$ or abnormal states satisfying χ, and only reaches intermediate l-labeled points satisfying $\mathsf{annot}(l)$. Given a program specification $(Pre, \mathsf{annot}, Post, \chi)$, a verification condition generator (VCgen) framework provides a set of sufficient proof obligations that ensures that the program satisfies the specification.

The VCgens defined in this section are hybrid in the sense that they take advantage of a previously computed analysis to reduce the size of proof obligations. We assume that the result of a relational analysis (Loc and $\dot{loc}$ respectively for source and bytecode programs) is given as input to the VCgen. For the abstract domain $\mathbb{D}$, we consider a relation $\models \subseteq \mathbb{D} \times \mathcal{A}$ such that for any guard b and any $d \in \mathbb{D}$, $d \models b$ indicates that the interpretation of the abstract element d ensures the validity of the condition b. For example, when accessing an array in the expression $a[x]$ we shall check that the value of the variable x is within the bounds of the array a. If we instantiate $\mathbb{D}$ with the domain of convex polyhedra, each element $d \in \mathbb{D}$ represents a set of linear constraints from which we can discover whether the condition $0 \leq x < |a|$ is satisfied.

A further improvement over standard VCgens consists of reusing the result of the analysis to strengthen loop invariants. This technique helps reducing the size of annotations and the burden of interactive specification. To that end, we assume a concretization function $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ to interpret abstract elements $d \in \mathbb{D}$ as assertions.

VCgen for Source Programs

Consider a specification $(Pre, \mathsf{annot}, Post, \chi)$ for the source program P. In this section, we assume that annot sufficiently annotates the program P, that is, for every subprogram `while` $[t]^l$ `do` c of P, we have that $l \in \mathsf{dom}(\mathsf{annot})$.

A VCgen for source programs is defined by the set of proof obligations PO defined as $\{Pre \Rightarrow \phi[{}^{\vec{V}}\!/_{\vec{V}_{old}}]\} \cup \theta$, where $\langle \phi, \theta \rangle = \mathsf{WP}(P, Post)$, $\phi[{}^{\vec{V}}\!/_{\vec{V}_{old}}]$ represents the result of substituting in ϕ any array or scalar variable

$$\frac{}{\mathsf{WP}(\texttt{Skip}, \phi) = \langle \phi, \emptyset \rangle} \qquad \frac{}{\mathsf{WP}([\texttt{return}\ e]^l, \phi) = \langle \mathsf{ckB}(e, \mathit{Post}[{}^{e}\!/_{\mathsf{res}}]), \emptyset \rangle}$$

$$\frac{}{\mathsf{WP}([x{:=}e]^l, \phi) = \langle \mathsf{ckB}(e, \phi[{}^{e}\!/_{x}]), \emptyset \rangle}$$

$$\frac{}{\mathsf{WP}([a[e_1]{:=}e_2]^l, \phi) = \langle \mathsf{ckB}(e_2, \mathsf{ckB}(a[e_1], \phi[{}^{\mathsf{upd}(a,e_1,e_2)}\!/_{a}])), \emptyset \rangle}$$

$$\frac{\mathsf{WP}(c_1, \phi) = \phi_1, \theta_1 \qquad \mathsf{WP}(c_2, \phi) = \phi_2, \theta_2}{\mathsf{WP}(\texttt{if}\ [t]^l\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \phi) = \langle \mathsf{ckB}(t, t \Rightarrow \phi_1 \wedge \neg t \Rightarrow \phi_2), \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\mathsf{WP}(c, \phi) = \langle \phi_1, \theta \rangle \qquad \Phi = (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \phi)}{\mathsf{WP}(\texttt{while}\ [t]^l\ \texttt{do}\ c, \phi) = \langle \mathsf{annot}(l), \{\mathsf{annot}(l) \wedge \gamma_{\mathsf{a}}(\mathit{Loc}(l)) \Rightarrow \mathsf{ckB}(t, \Phi)\} \cup \theta_1 \rangle}$$

$$\frac{\mathsf{WP}(c_1, \phi_2) = \langle \phi_1, \theta_1 \rangle \qquad \mathsf{WP}(c_2, \phi) = \langle \phi_2, \theta_2 \rangle}{\mathsf{WP}(c_1; c_2, \phi) = \langle \phi_1, \theta_1 \cup \theta_2 \rangle}$$

where the expression $\mathsf{ckB}(e, \varphi)$ stands for φ if $\mathit{Loc}(l) \models \mathsf{inB}(e)$ and the formula $\mathsf{inB}(e) \Rightarrow \varphi \wedge \neg\mathsf{inB}(e) \Rightarrow \chi$ otherwise.

Figure 2.11: Definition of WP function

x^{old} in $V_s^{old} + V_a^{old}$ by x, and the function WP is defined in Figure 2.11. In the figure, the assertion $\mathsf{inB}(e)$ stands for the condition that must satisfy an execution state to ensure that every array access in e is within bounds. For instance, if e does not contain array expressions $\mathsf{inB}(e)$ is defined as *True* and $\mathsf{inB}(a[e])$ as $0 \leq e < |a|$. We follow the simplifying assumption that expressions contain no more than one array access. For any array variable a and expressions e_1 and e_2, $\mathsf{upd}(a, e_1, e_2)$ is interpreted as the array a' such that $a'[e]$ is evaluated to e_2 if $e_1 = e$ and to $a[e]$ otherwise. To simplify the presentation of examples, proof obligations for `while` statements are split into two assertions corresponding to the *True* and *False* branches.

The function WP considers the result of the analysis *Loc* to reduce the size of proof obligations. That is, if the abstract value $\mathit{Loc}(l)$ associated to the program point under consideration indicates that any array access in the statement is within bounds, the returned predicate is simplified by omitting the exceptional postcondition. Consider the program of Figure 2.12. If the analysis is able to compute at label k_1 an abstract value d such that $d \models 0 \leq i < |A|$, the WP function will return the assertion $\mathsf{upd}(A, i, A[0])[i+1-1] = A[0]$, which together with the loop invariant at label k yields the proof obligation:

$$A[i-1] = 0 \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i+1-1] = A[0]$$

where the boxed assertion $\boxed{0 \leq i \leq |A|}$ represents the result of the analysis at the loop entry point.

In contrast, if we do not take advantage of the result of the analysis we must prove the bigger formula:

$$\begin{array}{l} A[i-1] = 0 \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \\ \qquad (0 \leq i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \wedge \neg(0 \leq i < |A|) \Rightarrow \mathit{False}) \end{array}$$

As can be seen from the definition of WP, proof obligations are of the form $\phi_1 \wedge \boxed{\gamma_{\mathsf{a}}(d)} \Rightarrow \phi_2$, whereas a standard VCgen outputs the stronger proof obligation $\phi_1 \Rightarrow \phi_2$. In consequence, one can provide the code with a weaker invariant ϕ_1 as long as the analyzer is able to eventually infer the missing information $\gamma_{\mathsf{a}}(d)$. For instance, for the simple program of Figure 2.12, a standard VCgen will return the invalid proof obligation

$$A[i-1] = A[0] \Rightarrow \neg(i < |A|) \Rightarrow A[|A|-1] = A[0]$$

for the path that does not enter the loop. It is sufficient to provide a stronger invariant, i.e. in conjunction with the condition $i \leq |A|$, to prove the program correct. However, as an alternative to increasing the size

```
$//Pre : True,\ \chi : False$
$[i := 1]^{k_0};$
$//A[i-1] = A[0]$
while $[i < |A|]^k$ do{
   $[A[i] := A[0]]^{k_1};\ [i := i+1]^{k_2}$
}
$//A[|A|-1] = A[0]$
$\ldots\ldots$
```

Figure 2.12: Program example

of the program annotations, assuming the condition $i \leq |A|$ is inferred by the analysis, the hybrid VCgen generates the weaker (and valid) proof obligation:

$$A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow \neg(i < |A|) \Rightarrow A[|A|-1] = A[0]$$

VCgen for Bytecode Programs

Let $(\mathit{Pre}, \mathsf{annot}, \mathit{Post}, \chi)$ be a specification for the bytecode program $\dot{p}$. As with the VCgen for source programs defined above, the precondition Pre and the internal annotations $\mathsf{annot}(l)$ are strengthened with the result of the analysis. To that end, we interpret the result of the analysis with the aid of the concretization functions $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ and $\bar{\gamma_{\mathsf{a}}} : (\mathit{Expr}^\star \times \mathbb{D}) \to \mathcal{A}$. A VCgen for bytecode is defined by extracting the set of proof obligations:

$$\mathsf{po} = \{\mathit{Pre} \Rightarrow \mathsf{wp_i}(0)[{}^{V}\!/_{\vec{V_{old}}}]\} \cup \{\mathsf{annot}(l) \wedge \bar{\gamma_{\mathsf{a}}}(\dot{loc}(l)) \Rightarrow \mathsf{wp_i}(l) \mid l \in \mathsf{dom}(\mathsf{annot})\}$$

where the predicate transformer $\mathsf{wp_l}$ is shown in Figure 2.13. If the program point is annotated, the function $\mathsf{wp_l}$ returns $\mathsf{annot}(l)$. Otherwise it applies the weakest precondition transformer $\mathsf{wp_i}$, defined in terms of the instruction at program point l, taking as parameters the annotations computed for the successor program points. The definition of $\mathsf{wp_l}$ and $\mathsf{wp_i}$ is done by induction along the control flow paths of the program. A program $\dot{p}$ is sufficiently annotated if the control flow graph of the program $\dot{p}$ does not contain unannotated loops. The induction principle following from the definition of sufficiently annotated programs is sufficient to ensure that $\mathsf{wp_l}$ and $\mathsf{wp_i}$ are well defined For a list s, $s[0]$ and $s[1]$ represent the first and second element of s, and $\uparrow s$ denotes the result of removing the first element from s.

Preservation of Proof Obligations

Consider the specification $(\mathit{Pre}, \mathsf{annot}, \mathit{Post}, \chi)$ for source program P, and assume that annot is a sufficient annotation for P, i.e. every loop is annotated. Let $(\mathit{Pre}, \mathsf{annot}, \mathit{Post}, \chi)$ define as well the specification for the bytecode program $\dot{p}$. From previous results, we know that if annot is a sufficient annotation for P then it is also a sufficient annotation for the result of the compilation $\dot{p}$. Let Loc be a solution of the analysis for the source program P, and $\dot{loc}$ a solution of the analysis for the bytecode program $\dot{p}$, compiled from Loc as described in Section 2.5.3.

We assume that the concretization functions satisfy the property $\bar{\gamma_{\mathsf{a}}}([], d) = \gamma_{\mathsf{a}}(d)$, so that the interpretation of abstract analysis results in the source and bytecode sides coincides (recall that by definition $\dot{loc}(l) = ([\,], \mathit{Loc}(l))$ for every l in $\mathsf{dom}(\mathit{Loc})$.) In addition, for any expression e and any $d \in \mathbb{D}$, if e does not contain array expressions, i.e. $\mathsf{inB}(e) = \mathit{True}$, then $d \models \mathsf{inB}(e)$.

The following result about the compilation of expressions is helpful to prove preservation of proof obligations:

Lemma 2.5.9 *Let $\dot{p}$ be equal to $\dot{p}_1 :: l_1{:}[\![e]\!]_{\mathrm{e}} :: l_2{:}\dot{p}_2$. Then $\mathsf{wp_i}(l_1)$ is equal to $\mathsf{wp_i}(l_2)[{}^{e::os}\!/_{os}]$ if $\dot{loc}(l_1) \models \mathsf{inB}(e)$ and equal to $\mathsf{inB}(e) \Rightarrow \mathsf{wp_i}(l_2)[{}^{e::os}\!/_{os}] \wedge \neg\mathsf{inB}(e) \Rightarrow \chi$ otherwise.*

$$\begin{array}{ll}
\mathsf{wp_i}(l) = \mathsf{wp_l}(l+1)[{}^{os[0]\ \mathsf{op}\ os[1]::\uparrow^2 os}/_{os}] & \dot{p}[l] = \mathtt{prim\ op} \\
\mathsf{wp_i}(l) = \mathsf{wp_l}(l+1)[{}^{v::os}/_{os}] & \dot{p}[l] = \mathtt{push}\ v \\
\mathsf{wp_i}(l) = \mathsf{wp_l}(l+1)[{}^{os[0],\uparrow os}/_{x,os}] & \dot{p}[l] = \mathtt{store}\ x \\
\mathsf{wp_i}(l) = \mathsf{wp_l}(l+1)[{}^{x::os}/_{os}] & \dot{p}[l] = \mathtt{load}\ x \\
\mathsf{wp_i}(l) = \mathsf{ckB}(\mathsf{wp_l}(l+1)[{}^{\mathsf{upd}(a,os[0],os[1]),\uparrow^2 os}/_{a,os}]) & \dot{p}[l] = \mathtt{astore}\ a \\
\mathsf{wp_i}(l) = \mathsf{ckB}(\mathsf{wp_l}(l+1)[{}^{a[os[0]]::\uparrow os}/_{os}]) & \dot{p}[l] = \mathtt{aload}\ a \\
\mathsf{wp_i}(l) = os[0] \bowtie os[1] \Rightarrow \mathsf{wp_l}(l')[{}^{\uparrow^2 os}/_{os}] & \dot{p}[l] = \mathtt{cjmp} \bowtie l' \\
\qquad\qquad \wedge\neg(os[0] \bowtie os[1]) \Rightarrow \mathsf{wp_l}(l+1)[{}^{\uparrow^2 os}/_{os}] & \\
\mathsf{wp_i}(l) = \mathsf{wp_l}(l') & \dot{p}[l] = \mathtt{jmp}\ l' \\
\mathsf{wp_i}(l) = \mathsf{wp_l}(l+1) & \dot{p}[l] = \mathtt{nop} \\
\mathsf{wp_i}(l) = Post[{}^{os[0]}/_{\mathsf{res}}] & \dot{p}[l] = \mathtt{return}
\end{array}$$

where $\mathsf{ckB}(\psi)$ stands for ψ if $\dot{loc}(l) \models \mathsf{inB}(x[os[0]])$ and $\mathsf{inB}(x[os[0]]) \Rightarrow \psi \wedge \neg\mathsf{inB}(x[os[0]]) \Rightarrow \chi$ otherwise.

$$\mathsf{wp_l}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \mathsf{wp_i}(l) & \text{otherwise} \end{cases}$$

Figure 2.13: VCgen for bytecode programs

```
k0 :push 1          load i
    store i         prim +
 k :jmp k'          store i
k1 :push 0      k' :push |A|
    aload A         load i
    load i          cjmp < k1
    astore A    k'' :...
k2 :push 1
```

Figure 2.14: Program example

The coincidence of the sets of proof obligations PO and po is stated in the following lemma, provided the bytecode program $\dot{p}$ is the result of compiling the source program P.

Proposition 2.5.10 *For every subprogram c of P, proof obligations corresponding to c are equal to the proof obligations in $\dot{p}$ that correspond to the subsequence $[\![c]\!]$.*

Consider, the bytecode program of Figure 2.14 compiled from the example in Figure 2.12. One can see that the proof obligation at label k is

$$\begin{array}{l}
A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow \\
\quad (i < |A| \Rightarrow (A[i-1] = A[0])[{}^{\mathsf{upd}(A,i,A[0]),i+1}/_{A,i}]) \ \wedge (\neg(i < |A|) \Rightarrow A[|A|-1] = A[0])
\end{array}$$

which is equal to the proof obligation at label k for the source program of Figure 2.12.

2.6 Certificates for numeric analyses

To get foundational certificates of program safety, the result of static analyses used by hybrid verification methods (see Section 2.5) cannot be trusted. Here, we show how to design a checker for certificates asserting the truth of annotations provided by a static analysis for a bytecode language (*e.g.*, the language presented in Section 2.5.3). The cornerstone of such a checker is a checker able to discharge numeric proof obligations. In Section 2.6.1, we extract from [27] the mathematical background needed for generating and checking invariants for several fragments of arithmetic. In Section 2.6.2, we show how to design an efficient Coq checker for linear invariants in the form of convex polyhedra [29].

2.6.1 Certificates for arithmetic

We study a hierarchy of three quantifier-free fragments of integer arithmetics. We describe certificates, off-the-shelf provers and certificate checkers associated to them. We consider formulae that are conjunctions of inequalities and we are interested in proving the unsatisfiability of these inequalities. Formally, formulae of interest have the form:

$$\neg \left(\bigwedge_{i=1}^{k} e_i(x_1, \ldots, x_n) \geq 0 \right)$$

where the e_is are fragment-specific integer expressions and the x_is are universally quantified variables.

For each fragment, we shall prove a theorem of the following general form:

$$(\exists cert, Cond(cert, e_1, \ldots, e_k)) \Rightarrow \forall (x_1, \ldots, x_n), \neg \left(\bigwedge_{i=1}^{k} e_i(x_1, \ldots, x_n) \right)$$

In essence, such a theorem establish that *cert* is a certificate of the infeasibility of the e_is. We then show that certificates can be generated by off-the-shelf algorithms and that *Cond* is decidable and can be efficiently implemented by a *checker* algorithm.

Potential constraints

To begin with, consider *potential constraints.* These are constraints of the form $x - y + c \geq 0$. Deciding the infeasibility of conjunctions of such constraints amounts to finding a cycle of negative weight in a graph such that a edge $x \overset{c}{\to} y$ corresponds to a constraint $x - y + c \geq 0$ [25, 87][3].

Theorem 2.6.1

$$\exists \pi \in Path, \bigwedge \left(\begin{array}{l} isCycle(\pi) \\ weight(\pi) < 0 \\ \pi \subseteq (\bigcup_{i=1}^{k} \{x_{i_1} \overset{c_i}{\to} x_{i_2}\}) \end{array} \right) \Rightarrow \forall x_1, \ldots, x_n, \neg(\bigwedge_{i=1}^{k} x_{i_1} - x_{i_2} + c_i \geq 0)$$

Proof 2.6.2 Ad absurdum, *we suppose that we have* $\bigwedge_{i=1}^{k} x_{i_1} - x_{i_2} + c_i \geq 0$ *for some* $x_1, \ldots, x_n$*. If we sum the constraints over a cycle* π*, variables cancel and the result is the total weight of the path* $\sum_{x \overset{c}{\to} y \in \pi} x - y + c = \sum_{x \overset{c}{\to} y \in \pi} c$*. Moreover, by hypothesis, we also have that* $\left(\sum_{x \overset{c}{\to} y \in \pi} x - y + c \right) \geq 0$ *(each element of the sum being positive). We conclude that the total weight of cycles is necessarily positive. It follows that the existence of a cycle of negative weight c yields a contradiction.* □

As a result, a negative cycle is a certificate of infeasibility of a conjunction of potential constraints.

Bellmann-Ford shortest path algorithm is a certificate generator which runs in complexity $O(n \times k)$ where n is the number of nodes (or variables) and k is the number of edges (or constraints). However, this algorithm does not find the best certificate *i.e.,* the negative cycle of shortest length. Certificates, *i.e.,* graph cycles, can be coded by a list of binary indexes – each of them identifying one of the k constraints. The worst-case certificate is then a Hamiltonian circuit which is a permutation of the k constraint indexes. Its asymptotic size is therefore $k \times log(k)$:

$$size(i_1, \ldots, i_k) = \sum_{j=1}^{k} log(i_j) = \sum_{j=1}^{k} log(j) = log(\Pi_{j=1}^{k} j) = log(k!) \sim k \times log(k)$$

[3] As shown by Shostak [98], this graph-based approach generalises to constraints of the form $a \times x - b \times y + c \geq 0$.

Verifying a certificate consists in checking that:

1. indexes are bound to genuine expressions;
2. verify that expressions form a cycle;
3. compute the total weight of the cycle and check its negativity

This can be implemented in time linear in the size of the certificate.

Linear constraints

The linear fragment of arithmetics might be the most widely used. It consists of formulae built over the following expressions:

$$Expr ::= c_1 \times x_1 + \ldots + c_n \times x_n + c_{n+1}$$

A well-known result of linear programming is Farkas's Lemma which states a strong duality result.

Lemma 2.6.3 (Farkas's Lemma (Variant)) *Let $A : \mathbb{Q}^{m \times n}$ be a rational-valued matrix and $b : \mathbb{Q}^m$ be a rational-valued vector. Exactly one of the following statement holds:*

- $\exists (y \in \mathbb{Q}^n), y \geq \bar{0}, b^t \cdot y > 0, A^t \cdot y = \bar{0}$
- $\exists (x \in \mathbb{Q}^m), A \cdot x \geq b$

Over $\mathbb{Z}$, Farkas's Lemma is sufficient to provide infeasibility certificates for systems of inequalities.

Lemma 2.6.4 (Weakened Farkas's Lemma (over $\mathbb{Z}$)) *Let $A : \mathbb{Z}^{m \times n}$ be a integer-valued matrix and $b : \mathbb{Z}^m$ be a integer-valued vector.*

$$\exists (y \in \mathbb{Z}^n), y \geq \bar{0}, b^t \cdot y > 0, A^t \cdot y = \bar{0} \Rightarrow \forall (x \in \mathbb{Z}^n), \neg A \cdot x \geq b$$

Proof 2.6.5 *Ad absurdum, we suppose that we have $A \cdot x \geq b$ for some vector x. Since y is a positive vector, we have that $(A \cdot x)^t \cdot y \geq b^t \cdot y$. However, $(A \cdot x)^t \cdot y = (x^t \cdot A^t) \cdot y = x^t \cdot (A^t \cdot y)$. Because $A^t \cdot y = 0$, we conclude that $0 \geq y^t \cdot b$ which contradicts the hypothesis stating that $y^t \cdot b$ is strictly positive.*□

Over $\mathbb{Z}$, Farkas's lemma is not complete. Incompleteness is a consequence of the discreetness of $\mathbb{Z}$: there are systems that have solutions over $\mathbb{Q}$ but not over $\mathbb{Z}$. A canonical example is the equation $2.x = 1$. The unique solution is the rational $1/2$ which obviously is not an integer. Yet, the loss of completeness is balanced by a gain in efficiency. Whereas deciding infeasibility of system of integer constraints is NP-complete; the same problem can be solved over the rationals in polynomial time.

Indeed, an infeasibility certificate is produced as the solution of the *linear program*

$$min\{y^t \cdot \bar{1} \mid y \geq \bar{0}, b^t \cdot y > 0, A^t \cdot y = \bar{0}\}$$

Note that linear programming also *optimises* the certificate. To get *small* certificates, we propose to minimise the sum of the elements of the solution vector.

Linear programs can be solved in polynomial time using interior point methods [58]. The Simplex method – despite its worst-case exponential complexity – is nonetheless a practical competitive choice.

Linear programs are efficiently solved over the rationals. Nonetheless, an integer certificate can be obtained from any rational certificate.

Proposition 2.6.6 (Integer certificate) *For any rational certificate of the form $cert_{\mathbb{Q}} = [p_1/q_1; \ldots ; p_k/q_k]$, an integer certificate is*

$$cert_{\mathbb{Z}} = [p'_1; \ldots ; p'_k]$$

where $p'_i = p_i \times lcm/q_i$ and lcm is the least common multiple of the q_is.

Worst-case estimates of the size of the certificates are inherited from the theory of integer and linear programming (see for instance [96]).

Theorem 2.6.7 (from [96] Corollary 10.2a) *The bit size of the rational solution of a linear program is at most* $4d^2(d+1)(\sigma+1)$ *where*

- *d is the dimension of the problem;*
- *σ is the number of bits of the biggest coefficient of the linear program.*

Using Lemma 2.6.6 and Theorem 2.6.7, the next Corollary gives a coarse upper-bound of the bit size of integer certificates.

Corollary 2.6.8 (Bit size of integer certificates) *The bit size of integer certificates is bounded by* $4k^3(k+1)(\sigma+1)$

Proof 2.6.9 *Let* $cert_{\mathbb{Z}} = [p'_1; \ldots; p'_k]$ *be the certificate obtained from a rational certificate* $cert_{\mathbb{Q}} = [p_1/q_1; \ldots; p_k/q_k]$.

$$\begin{array}{rcl} \mid cert_{\mathbb{Z}} \mid & = & \sum_{i=1}^{k} log(p'_i) \\ & = & \sum_{i=1}^{k}(log(p_i) - log(q_i)) + \sum_{i=1}^{k} log(lcm) \\ & = & \sum_{i=1}^{k}(log(p_i) - log(q_i)) + k \times log(lcm) \end{array}$$

At worse, the q_i*s are relatively prime and* $lcm = \Pi_{i=1}^{n} q_i$.

$$\mid cert_{\mathbb{Z}} \mid \leq k \times \sum_{i=1}^{k} log(q_i) + \sum_{i=1}^{k}(log(p_i) - log(q_i))$$

As $\mid cert_{\mathbb{Q}} \mid = \sum_{i=1}^{k} log(p_i) + log(q_i)$, *we have that* $\mid cert_{\mathbb{Z}} \mid \leq k \times \mid cert_{\mathbb{Q}} \mid$. *By Theorem 2.6.7, we conclude the proof and obtain the* $4k^3(k+1)(\sigma+1)$ *bound.* □

Optimising certificates over the rationals is reasonable. Rational certificates are produced in polynomial time. Moreover, the worst-case size of the integer certificates is kept reasonable.

Checking a certificate *cert* amounts to

1. checking the positiveness of the integers in *cert*;
2. computing the matrix-vector product $A^t \cdot cert$ and verifying that the result is the null vector;
3. computing the scalar product $b^t \cdot cert$ and verifying its strict positivity

Overall, this leads to a quadratic-time $O(n \times k)$ checker in the number of arithmetic operations.

Polynomial constraints

For our last fragment, we consider unrestricted expressions built over variables, integer constants, addition and multiplication.

$$e \in Expr ::= x \mid c \mid e_1 + e_2 \mid e_1 \times e_2$$

As it reduces to solving diophantine equations, the logical fragment we consider is not decidable over the integers. However, it is a result by Tarski [103] that the first order logic $\langle \mathbb{R}, +, *, 0 \rangle$ is decidable. In the previous section, by lifting our problem over the rationals, we traded incompleteness for efficiency. Here, we trade incompleteness for decidability.

In 1974, Stengle generalises Hilbert's *nullstellenstaz* to systems of polynomial inequalities [101]. As a matter of fact, this provides a *positivstellensatz*, *i.e.,* a theorem of positivity, which states a necessary and sufficient condition for the existence of a solution to systems of polynomial inequalities. Over the integers, unlike Farkas's lemma, Stengle's *positivstellensatz* yields sound infeasibility certificates for conjunctions of polynomial inequalities.

Definition 2.6.10 (Cone) *Let $P \subseteq \mathbb{Z}[\bar{x}]$ be a finite set of polynomials. The cone of P ($Cone(P)$) is the smallest set such that*

1. $\forall p \in P, p \in Cone(P)$
2. $\forall p_1, p_2 \in Cone(P), p_1 + p_2 \in Cone(P)$
3. $\forall p_1, p_2 \in Cone(P), p_1 \times p_2 \in Cone(P)$
4. $\forall p \in \mathbb{Z}[\bar{x}], p^2 \in Cone(P)$

Theorem 2.6.11 states sufficient conditions for infeasibility certificates:

Theorem 2.6.11 (Weakened Positivstellensatz) *Let $P \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ be a finite set of polynomials.*

$$\exists cert \in Cone(P), cert \equiv -1 \Rightarrow \forall x_1, \ldots, x_n, \neg \bigwedge_{p \in P} p(x_1, \ldots, x_n) \geq 0$$

Proof 2.6.12 By adbsurdum, *we suppose that we have $\bigwedge_{p \in P} p(x_1, \ldots, x_n) \geq 0$ for some $x_1, \ldots, x_n$. By routine induction over the definition of a Cone, we prove that any polynomial $p \in Cone(P)$ is such that $p(x_1, \ldots, x_n)$ is positive. This contradicts the existence of the polynomial cert which uniformly evaluates to* -1. □

Certificate generators explore the cone to pick a certificate. Stengle's result [101] shows that only a restricted (though infinite) part of the cone needs to be considered. A certificate *cert* can be decomposed into a finite sum of products of the following form:

$$cert \in \sum_{s \in 2^P} \left(q_s \times \prod_{p \in s} p \right)$$

where $q_s = p_1^2 + \ldots + p_i^2$ is a *sum of squares* polynomial.

As pointed out by Parrilo [84], a layered certificate search can be carried out by increasing the formal degree of the certificate. For a given degree, finding a certificate amounts to finding polynomials (of known degree) that are sums of squares. This is a problem that can be solved efficiently (in polynomial time) by recasting it as a *semidefinite program* [105]. The key insight is that a polynomial q is a *sum of square* if and only if it can be written as

$$q = \begin{pmatrix} m_1 \\ \ldots \\ m_n \end{pmatrix}^t \cdot Q \cdot \begin{pmatrix} m_1 \\ \ldots \\ m_n \end{pmatrix}$$

for some positive semidefinite matrix Q and some vector $(m_1, \ldots, m_n)$ of linearly independent monomials.

An infeasibility certificates is a polynomial which belongs to the cone and is equivalent to -1. Using a suitable encoding, cone membership can be tested in linear time. Equivalence with -1 can be checked by putting the polynomial in Horner's normal form.

2.6.2 Result checking of polyhedral operations

Our current analyser instantiates the generic numeric interface of Section 2.5.3 with octagons [75] and convex polyhedra [40] – domains provided by the Apron library [56]. These implementations are highly optimised and it would be impossible to certify them entirely. In this section, we show how to develop an efficient, yet certified implementation of convex polyhedra operators in Coq using a result checking approach. Because polyhedra supersedes octagons, our checker is designed, from the start, to handle arbitrary linear invariants in the form of convex polyhedra. Our polyhedra being already very efficient, it is not worth implementing a specialised checker for octagons.

The polyhedral domain revisited

Polyhedra can be represented as sets of linear constraints. For efficiency, it is desirable to keep these sets in normal form *i.e.,* without redundant constraints. For this purpose, polyhedra libraries maintain a dual description 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.

At the origin of the efficiency (and complexity) of convex polyhedra algorithms is Chernikova's algorithm which is used to maintain the coherence of the double description of polyhedra [35]. A key insight of our approach is that we develop a checker which only uses the constraint description of polyhedra and which never needs to detect redundant constraints. 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 2.6.13 *Let V and E be disjoint sets of variables.*

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

where $Lin_X = \{c_0 + c_1 \times x_1 + \cdots + c_n \times x_n \mid c_i \in \mathbb{Z}, x_i \in X\}$.

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

$$\gamma(es) = \{\rho_{|V} \mid \rho \in (V+E) \to \mathbb{Z} \wedge \forall lc \in es, [\![lc \geq 0]\!]_\rho\}$$

Efficient Coq implementation of $\mathbb{P}_V$. We have implemented (and proved correct) a result checker for convex polyhedra based on an efficient implementation of $\mathbb{P}_V$. Since polyhedra manipulations are costly, special care must be taken to the efficiency of the checker. We have therefore carefully chosen efficient data-structures. Variables are coded by binary integers *i.e.,* the Coq positive type.

```
Inductive positive : Set
  := xH | x0 (p:positive) | xI (p:positive).
```

Variables in $v \in V$ start with a x0 constructor while existential variables $v \in E$ start with a xI constructor. A linear expression $e \in Lin$ is coded by a binary tree whose node labels record integer coefficients of the linear expression.

```
Inductive tree : Set :=
  | Leaf
  | Node (left:tree) (label:Z) (right:tree).
```

Therefore, looking-up a variable coefficient can be done by following a path in the tree. This operation executes in time linear in the length of the variable *i.e.,* logarithmic in the number of variables. For efficiency again, Coq polyhedra $p \in \mathbb{P}_V$ are not simply lists of linear expressions but are dependent records which store: i) a list lin_cstr of linear constraints coded as trees, ii) a variable fresh_v $\in V$ whose successors are fresh, iii) a variable fresh_e $\in E$ whose successors are fresh, iv) a set used_v that stores the variables $v \in V$ that are used in lin_cstr , v) and all the proofs *i.e.,* the data-structure invariants, that ensure that fresh_v and fresh_e are really fresh and that the set used_v indeed over-approximates the variables used in lin_cstr .

Checking convex polyhedra operations. 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. Using Fourier-Motzkin elimination (see *e.g.,* [96]), **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 existential variables.

To compute **intersections**, care must be taken not to mix up the existential variables. To avoid captures, existentially variables are renamed to variables that are fresh for both polyhedra. Interestingly, with our tree encoding, renaming all the existential variables is a constant time operation. Thereafter, the intersection is obtained by concatenating the lists of linear expressions.

To implement the **assume** and **ensure** operators, the expressions in the guard are linearised and normalised (see Section 2.6.1). The obtained linear constraints are then syntactically concatenated to the current polyhedron.

Assignment can be expressed in terms of the previous operators. Given x' a fresh existential variable, we have:

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

To implement **inclusion tests**, certificates are used. The form of certificates and their generation are described below.

Result certification for polyhedral inclusion

Our inclusion checker 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 ($P \sqsubseteq Q$).

In practise, we only use our checker where Q does not contain existential variables (because Q is computed by the untrusted analyser). This allows us to reduce the problem of inclusion into n problems of polyhedron emptiness where n is the number of constraints in Q. Emptiness can be checked by result certification thanks to Farkas lemma (see Lemma 2.6.3) that gives a notion of *emptiness* certificate for polyhedra.

In general, certificates can be generated by solving a linear program (see Section 2.6.1). However, if the polyhedron is actually an octagon *i.e.*, a conjunction of potential constraints, we get a degenerated form of certificate, for which the coefficient are either 0 or 1, that can be obtained by Bellmann-Ford shortest path algorithm [25] (see Section 2.6.1). As shown in Section 2.6.1, the certificate checker runs in quadratic-time in terms of arithmetic operations for each emptiness certificate.

2.7 Modifies clauses

Another interesting certification method involves frame properties, in particular modifies clauses. Modifies clauses in JML or BML specify which variables can be modified in a course of a method execution. We present here a static analysis to check the correctness of modifies clauses within a PCC framework. We closely follow the syntax and convention introduced in Definition 2.4.1.

We deal with the modifies clauses of the following form:

$$modif ::= x.* \mid a[*]$$

where x ranges over `this` and formal parameters of reference type and a ranges over formal parameters of array type. The specification of each method contains a list of such clauses. We write $mspec(x) = modifiable$ if a clause $x.*$ or $x[*]$ is in the list and $mspec(x) = unmodifiable$ otherwise.

In the original BML the syntax of modifies clauses is richer. One can also precisely specify which fields of a given object can be modified, precisely specify array boundaries within which the modification takes place and nested array and field access, e.g. $a[5].x.*$, is also allowed.

Example 5 *The modifies analysis $\mathcal{A}_{\mathsf{mod}} = (D_{\mathsf{mod}}, t_{\mathsf{mod}}, I_{\mathsf{mod}}, \uparrow)$ is defined as follows. The domain D_{mod} represents the operand stack and the local variables, and is defined by:*

$$\begin{array}{rcl} D_{\mathsf{mod}} & = & (list\ \mathsf{ModProp})_\bot^\top \times (Var \to \mathsf{ModProp}) \\ \mathsf{ModProp} & = & \{modifiable, unmodifiable\}_\bot^\top \end{array}$$

The transfer function, $t_{\mathsf{mod}(pc,pc')}(d')$ is defined in two stages, First an auxiliary function t'_{mod} is defined by case analysis in the instruction at pc and in d' and then the final t_{mod} function is obtained from t'_{mod} by modifying the egdes starting from pc_0 to take the initial method specification into account. Some of the rules for t'_{mod} are:

- *if the instruction is Getfield f, then*

$$\begin{array}{rcl} t'_{\mathsf{mod}(pc,pcNext)}(unmodifiable :: s, l) & = & (\top :: s, l) \\ t'_{\mathsf{mod}(pc,pcNext)}(modifiable :: s, l) & = & (\bot, \bot) \\ t'_{\mathsf{mod}(pc,pcNext)}(\top :: s, l) & = & (\top :: s, l) \\ t'_{\mathsf{mod}(pc,pcNext)}(\bot :: s, l) & = & (\bot, \bot) \\ t'_{\mathsf{mod}(pc,pcNull)}(s, l) & = & (\top, l), \end{array}$$

  where pcNext *and* pcNull *are the successor program points corresponding to normal execution and exceptional execution respectively. Note that we do not allow for the result of Getfieldto be modifiable.*

- *if the instruction is Putfield f, then*

$$\begin{array}{rcl} t'_{\mathsf{mod}(pc,pcNext)}(s, l) & = & (modifiable :: \top :: s, l) \\ t'_{\mathsf{mod}(pc,pcNull)}(s, l) & = & (\top, l), \end{array}$$

  where pcNext and pcNull are the successor program points corresponding to normal execution and exceptional execution respectively.

- *if the instruction is Return, then*

$$t'_{\mathsf{mod}(pc,pc_{\mathrm{N}})}(v :: s, l) = (v :: s, l).$$

  Note that pc_{N} is the only successor of pc.

- *if the instruction is Aload x, then*

$$t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(e :: s, l) = (s, l')$$

  where $l' = l[x \mapsto l(x) \sqcap e]$.

- *if the instruction is Astore x, then*

$$t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(s, l) = (l(x) :: s, l')$$

  where $l' = l[x \mapsto \top]$.

- *if the instruction is Vload x, then*

$$t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(e :: s, l) = (s, l)$$

- *if the instruction is Vstore x, then*

$$t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(s, l) = (\top :: s, l)$$

- *if the instruction is Aaload, then*

$$\begin{array}{rcl} t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(modifiable :: s, l) & = & (\bot, \bot) \\ t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(unmodifiable :: s, l) & = & (\top :: \top :: s, l) \\ t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(\top :: s, l) & = & (\top :: \top :: s, l) \\ t'_{\mathsf{mod}(pc,\mathrm{pcNext})}(\bot :: s, l) & = & (\bot, \bot) \end{array}$$

- *if the instruction is Aastore, then*

$$t'_{\mathsf{mod}\,(pc,\mathrm{pcNext})}(s,l) = (\top :: \top :: \mathit{modifiable} :: s, l)$$

- *if the instruction is Vaload, then*

$$t'_{\mathsf{mod}\,(pc,\mathrm{pcNext})}(e :: s,l) \quad = \quad (\top :: \top :: s, l)$$

- *if the instruction is Vastore, then*

$$t'_{\mathsf{mod}\,(pc,\mathrm{pcNext})}(s,l) = (\top :: \top :: \mathit{modifiable} :: s, l)$$

- *if the instruction is Invokevirtual m, then if the return type of m is a value type, then*

$$t'_{\mathsf{mod}\,(pc,pcNext)}(e :: s,l) = (args \mathbin{++} loc :: s, l)$$

otherwise (if the return type of m is a reference type):

$$\begin{array}{rcl} t'_{\mathsf{mod}\,(pc,pcNext)}(\mathit{unmodifiable} :: s,l) & = & (args \mathbin{++} loc :: s, l) \\ t'_{\mathsf{mod}\,(pc,pcNext)}(\mathit{modifiable} :: s,l) & = & (\bot, \bot) \\ t'_{\mathsf{mod}\,(pc,pcNext)}(\bot :: s,l) & = & (\bot, \bot) \\ t'_{\mathsf{mod}\,(pc,pcNext)}(\top :: s,l) & = & (args \mathbin{++} loc :: s, l) \end{array}$$

where each of args and loc is modifiable if the name of the parameter on that position is in the modifies clause of m or ⊤ otherwise.

Moreover, since the method invocation is not always successful,

$$\begin{array}{rcl} t'_{\mathsf{mod}\,(pc,pcExc)}(s,l) & = & (args \mathbin{++} loc :: s, l) \\ t'_{\mathsf{mod}\,(pc,pc_{\mathrm{E}})}(s,l) & = & (args \mathbin{++} loc :: s, l) \end{array}$$

where args and loc are as before and pcExc ranges over all exception handlers whose range include pc. We define it this way because, without further analysis, we cannot know which exception can be thrown by a method. Hence, we assume that any exception in range can be thrown. Also, an Invokevirtual instruction has the node pc_{E} as successor, since the called method can throw an exception that can be uncaught in the current method.

The final function t_{mod} is defined as follows:

$$t_{\mathsf{mod}\,(pc,pc')}(s,l) = t'_{\mathsf{mod}\,(pc,pc')}(s,l)$$

if $pc \neq pc_0$ and

$$t_{\mathsf{mod}\,(pc_0,pc')}(s,l) = ([], l'')$$

where $t'_{\mathsf{mod}\,(pc_0,pc')}(s,l) = ([], l')$ and $l''(x) = l'(x) \sqcap mspec(x)$.

Tu pewnie trzeba jeszcze cos napisac o soundness...

2.8 Specification and certificate format

In this section, we present a generic certificate format to be used in the actual class files. Additionally, we present here a description on how the generic format can be adapted to the certificates that contain proofs for the BML specifications [32].

2.8.1 Conventions for binary representation

In this section, we recall the conventions used in description of the binary format of bytecode files. These conventions are primarily described in [66]. We recall them here to make the current document more self-contained.

The class files are considered to be streams of 8-bit bytes. The values with representations taking up 16 bits, 32 bits and 64 bits are constructed by reading two, four, and eight consecutive 8-bit values, respectively. These multibyte values are always stored in the big-endian fashion in which the high bytes come first.

The format of binary structures is presented in a C-like structure notation. By convention, the elements of the structures are referred to as items. The successive items are stored in the class file sequentially with no padding or alignment. We use types `u1`, `u2`, `u4` to represent unsigned one-, two-, or four-byte quantities, respectively. We use here the array notation adopted from C to define tables. The items in the tables, however, have in this case varying size so it is impossible to deduce the offset in the stream based on an index only. In case the items of tables have fixed size we refer to them as arrays.

For example the C-like structure

```
attribute_info {
 u2 attribute_name_index;
 u4 attribute_length;
 u1 info[attribute_length];
}
```

describes the general format of a classfile attribute. This structure says that each attribute starts with a two-byte value (this value is interpreted as an index to the so called Constant Pool, which contains various constant values used in a classfile), this value is followed by a four-byte length of an array, and the array is the third item in the structure.

The certificates we propose here are located in classfile attributes. Each attribute is equipped with a textual name. The Sun naming convention suggests to build names so that they are worldwide unique. We use here the naming scheme in which each name of an attribute defined in MOBIUS is prefixed with `mobius` followed by the dot character. In the text hereafter, we omit the prefix to make the presentation cleaner so that when we describe an attribute we refer to as `PCCCert` we actually mean `mobius.PCCCert`.

The classfile attributes may be located in special attribute tables which occur in different places of the classfile. In this document, we refer to attribute tables that are associated with the class itself and with methods.

2.8.2 Design principles

There are three basic design issues to be reconciled here. The first one is the flexibility of the format — the binary format should be flexible enough to accommodate the different approaches to the certification (it should be able to contain the type-based, logic-based and hybrid certificates). The second one is the problem of efficiency — we have to be careful not to make the resulting checking obligations unnecessarily big. The third one is to ensure that the proofs of all the needed properties can be included into the certificates in a natural way.

First of all, different approaches to the certification require different certificate formats. In this light, the current generic format definition requires a notion of a *certificate type*. Each particular certificate type will have its own detailed definition.

The certificates contain proofs of the code properties so the most natural approach is to connect the proofs with the code they are related to. In this light, we need a certificate attribute which is closely attached to methods. We call these certificates *method-level* certificates. However, one needs often some common definitions or lemmas which are used in proofs of several different methods in a class. In order to avoid copying of the common information in the different method attributes, we introduce a single class certificate attribute which contains the shared definitions and proofs. We call these certificates *class-level* certificates.

We do not require both kinds of certificates to be defined for a particular type of certificate. On one hand, one may consider the proofs for methods to be so much interwoven that it is impossible or irrational to keep them separate. In this case, the method-level certificates may disappear. On the other hand, the amount of the common information shared between methods may be so little that it is not necessary to gather that in a single place. In this case, the class-level certificates may disappear.

This class-level attribute allows to introduce additional structure that relates certificates for different classes. We assume that each certificate type comes along with a definition of a *signature* the certificate exposes to the external world. The elements of the signature can be used within the other certificate attributes. The definition of the signature can be embedded in the class-level certificate, but it is not required. The signatures may be automatically generated by the final certificate checker.

We assume that once the signatures are defined one needs a way to point which external definitions are used inside a particular class. We introduce into the certificates a mechanism to point which external certificates should be imported. We do not assume any implicit rules for certificate imports. In particular, when one wants to include a signature of the superclass certificate or a signature of any class which is referenced in the bytecode, it must be done explicitly.

Sometimes, the proofs require additional external proof libraries to be included before they can be completed. We do not impose any standard way to specify the access to these libraries as this is specific to a particular proving technology.

We assume that the properties of the code which are proved in the certificates may not be located in the certificate itself. They can be located in other classfile attributes (e.g. in BML attributes) or be embedded in the final certificate checker. The way the properties descriptions are matched with the certificates is dependent on the particular certificate format. We present in Section 2.8.4 an example of how such a correspondence can be defined.

The tools which handle the certificates will evolve so we introduce a versioning mechanism to the format. This versioning mechanism is based on the major and minor version number. We assume here that a change in the minor version number means that the final checker which worked properly with the previous version of the certificate can verify the certificates with the new version as well. However, the change in the major version number means that the final checker must be upgraded to understand the new format. We include the version numbers to both class certificate attributes and method certificate attributes as we admit the possibility that only one kind of the certificates occurs actually in a classfile.

2.8.3 The description of the format

In this section we present the actual format of the certificates. The rationale behind the solutions used here is presented in the section above.

The certificate attributes use an additional attribute which is defined already within the efforts around BML. For completeness, we cite the attribute description here. This attribute occurs in the class attributes table and has the form:

```
SecondConstantPool_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 second_cp_count;
    cp_info second_cp[second_cp_count-1];
}
```

where `cp_info` is a structure that describes a particular entry in the constant pool. The general form of the structure is as follows:

```
cp_info {
    u1 tag;
    u1 info[];
}
```

where tag is a particular type of the entry and info contains the actual data of the constant. Detailed description of different types of entries can be found in the *The Java Virtual Machine Specification* [66, Section 4.4].

The class PCC certificate is one of the attributes that occur in the class attribute table. It has the form:

```
PCCClassCert {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 cert_type;
    u1 major_version;
    u1 minor_version;
    u2 imported_certs_count;
    u2 imported_certs[imported_certs_count];
    u4 proofs_section_length;
    u1 proofs_section[proofs_section_length];
}
```

The field cert_type contains index to a position in the Second Constant Pool which contains the string with the name of the particular certificate type. The fields major_version and minor_version contain the values of the major and the minor version of the certificate. The imported_certs array contains indexes to the positions in the Second Constant Pool with the fully qualified names of the classes (their format is exactly the same as the format of fully qualified class names in the Constant Pool). The section proofs_section contains the actual content of the certificate and is dependent on the particular type of the certificate.

The method PCC certificate is one of the attributes that occur in the attribute table that lays within the Java method structure. Each method should have there its own certificate attribute. This attribute has the form:

```
PCCMethodCert {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 cert_type;
    u1 cert_major_version;
    u1 cert_minor_version;
    u4 proofs_section_length;
    u1 proofs_section[proofs_section_length];
}
```

The field cert_type contains index to a position in the Second Constant Pool which contains the name of the particular certificate type. The fields major_version and minor_version contain the values of the major and the minor version of the certificate. These three values should be the same as in the PCCClassCert of the class the method belongs to. The section proofs_section contains the actual content of the certificate and is dependent on the particular type of the certificate.

2.8.4 Relation to BML

This section contains a summary of the way the generic certificate schema can be applied to the situation in which the certificates are Coq proofs of the properties expressed in BML. It serves as a case study which shows the actual utility of the format described above. In this section we sketch the way data included in certificates is transformed to a format understandable for the Coq proof assistant. The typechecking engine of Coq is the final certificate checker in this case. We assume that the classfile is equipped with both BML specifications and related certificates.

The certificate type in this case is an index to a string which has the form mobius.BMLCoq. This section presents the version 1.0 of the certificate format (i.e. the major number is 1 and the minor one is 0).

Signature of the class

The class-level certificate leads to a definition of the certificate signature. This is a Coq module type the name of which has the form `LIST_NAME` where `NAME` is the fully qualified name of the class with dots replaced with underscores and all the letters changed to the capital ones, e.g. the name of the module type that corresponds to the java.lang.Object class is `LIST_JAVA_LANG_OBJECT`. This type imports, using the Coq *Require Export*, all the module types from the certificates of the direct superclass and interfaces. Note that this means that the attribute itself must declare the direct superclass and interfaces in the `imported_certs` table.

This module type contains additionally

- A definition of the `class` identifier which describes the structure of the class as required by Bicolano.
- All the definitions necessary to define the above `class`, in particular all the definitions of methods.
- All the definitions of preconditions and postconditions of methods.
- Module type parameters that for each method marked as public or spec_public expresses the property that if the method is called in a state in which the precondition holds then the postcondition holds after a return from the method.

This module type is used in the course of verification of other classes as a way to refer to the Coq representations of the specifications concerning the current class. The definitions of the methods are generated based on the classfile method structures. The definitions of preconditions and postconditions are composed using the BML preconditions and postconditions together with invariants — in the end we have only one pair precondition-postcondition for each method.

Class-level modules

The class-level certificate gives rise to two Coq modules, aside the mentioned above module type. The first module is an implementation of the module type while the second one contains definitions and proofs that are common for all the correctness proofs of methods. We generate separate modules for all the methods as we expect that verification of some of them may be resource consuming and thus worth separating in different units.

Implementation of the module type The name of the module has the form `List_Name` where `Name` is the fully qualified name of the class with dots replaced with underscores, e.g. the name of the module that corresponds to the java.lang.Object class is `List_java_lang_Object`. This module imports using the Coq *Require Export* all the modules imported in the module type `LIST_NAME` and all the method-level modules (not necessarily public ones). The latter modules contain all the definitions and proofs needed by the signature module type so nothing more is necessary.

Common definitions and proofs The name of the module has the form `List_Name_Common` where `Name` is as in the case before. This module imports using the Coq *Require Export* all the modules imported in the module type `LIST_NAME`. Except from that it contains:

- the definitions of all the preconditions and postconditions as they occur in `LIST_NAME` module type,
- common definitions and proofs which are directly the actual content of the class-level attribute.

Method-level modules

The name of the module has the form `List_Name_MName` where `Name` is the fully qualified name of the class with dots replaced with underscores, and `Name` is the name of the method. For instance, the name of the module that corresponds to the method toString in java.lang.Object class is `List_java_lang_Object_toString`. This module imports using the Coq *Require Export* all the modules imported in the module type `LIST_NAME`. Except from that it contains:

- the definition of the method in question,
- the theorem which states that if the method is called in a state in which the precondition holds then the postcondition holds after a return from the method,
- the proof of the theorem above,
- for each assert the theorem which states that if the method is called in a state in which the precondition holds then the assert holds in a related program position,
- the proofs of the theorems above.

The method-level certificate for a given method m contains the proofs of the theorems included in the module corresponding to m. The proofs are identified by names of theorems they prove.

2.8.5 Bico — from bytecode to Bicolano

A preliminary version of a tool that transforms bytecode files equipped with specifications and proof certificates to a format accepted by the Coq proof assistant is already developed. For the time being the tool can translate a classfile with no BML specifications and certificates to Coq files in the format described above with the additional restriction that no BML related definitions are added. We plan to extend the tool so that it handles the BML specifications and the related proof certificates.

Chapter 3

Flexible certificates

The use of Coq-based certificates is central to much to many aspects of the MOBIUS research and development work. Within this work package, another form of proof certificate has also been explored and developed. In particular, fixed points, as represented by sets of atomic formulas, are also a popular form of proof since these are used in many static analysis systems and can be use as certificates for model checkers. In this Chapter we explore in more detail our work on the structure and use of such fixed points as proof certificates.

3.1 Reduced Certificates in Abstraction Carrying Code

Abstraction-Carrying Code (ACC) has recently been proposed as a framework for mobile code safety in which the code supplier provides a program together with an *abstraction* (or abstract model of the program) whose validity entails compliance with a predefined safety policy. The abstraction plays thus the role of safety certificate and its generation is carried out automatically by a fixed-point analyzer. The advantage of providing a (fixed-point) abstraction to the code consumer is that its validity is checked in a *single pass* (i.e., one iteration) of an abstract interpretation-based checker. A main challenge to make ACC useful in practice is to reduce the size of certificates as much as possible while at the same time not increasing checking time. The intuitive idea is to only include in the certificate information that the checker is unable to reproduce without iterating. We introduce the notion of *reduced certificate* which characterizes the subset of the abstraction which a checker needs in order to validate (and re-construct) the *full certificate* in a single pass. Based on this notion, we can instrument a generic analysis algorithm with the necessary extensions in order to identify the information relevant to the checker. Interestingly, the fact that the reduced certificate omits (parts of) the abstraction has implications in the design of the checker. We provide the sufficient conditions which allow us to ensure that 1) if the checker succeeds in validating the certificate, then the certificate is valid for the program (correctness) and 2) the checker will succeed in validating any reduced certificate which is valid (completeness). We have developed our ideas on a generic analysis algorithm which has been recently used for the analysis and verification of Java bytecode programs [71]. The application to Java bytecode is achieved by direct adaptations of essentially the same parametric fixpoint-based analysis algorithm that we have used in this work on the producer side.

3.1.1 The Motivation

Abstraction-Carrying Code (ACC) [14] has been proposed as an enabling technology for PCC in which an *abstraction* (or abstract model of the program) plays the role of certificate. An important feature of ACC is that not only the checking, but also the generation of the abstraction is carried out automatically, by a fixed-point analyzer. In this part, we will consider analyzers which construct a program *analysis graph* which is interpreted as an abstraction of the (possibly infinite) set of states explored by the concrete execution. To capture the different graph traversal strategies used in different fixed-point algorithms, we use the *generic* description of [54], which generalizes the algorithms used in state-of-the-art analysis engines.

Essentially, the certification/analysis carried out by the supplier is an iterative process which repeatedly traverses the analysis graph until a fixpoint is reached. The analysis information inferred for each "call" which appears during the (multiple) graph traversals is stored in the *answer table.* The calls correspond to the abstract states which appear in the concrete execution of the program. After each iteration (or graph traversal), if the answer computed for a certain call is different from the one previously stored in the answer table, both answers are combined (by computing their lub) and the result is used 1) to *update* the table, and 2) to enforce recomputation of those calls whose answer depends on it. In the original ACC framework, the final *full* answer table constitutes the certificate. A main idea is that, since this certificate contains the fixpoint, a single pass over the analysis graph is sufficient to validate such certificate on the consumer side.

Both the ACC framework and our work here are defined for Constraint Logic Programs (CLP) at the *source* level. In contrast, in existing PCC frameworks, the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). Nevertheless, our choice of making our presentation at the source level is without loss of generality because both the original ideas in the ACC approach and those in our current proposal can also be applied directly to bytecode. Indeed, a good number of abstract interpretation-based analyses have been proposed in the literature for bytecode and machine code, most of which compute a fixpoint during analysis which can be reduced using the general principle of our proposal. For instance, in recent work, the concrete CLP verifier used in the original ACC implementation has itself been shown to also be applicable without modification to Java bytecode via a transformational approach [12, 13, 48, 10]. Furthermore, in [71, 72] a fixpoint-based analysis framework has been developed specifically for Java bytecode which is essentially equivalent to that used in the ACC proposal and to the one that we will apply in this work on the producer side to perform the analysis and verification, thus supporting the direct applicability of our approach to bytecode.

One of the main challenges for the practical uptake of ACC (and related methods) is to produce certificates which are reasonably small. This is important since the certificate is transmitted together with the untrusted code and, hence, reducing its size will presumably contribute to a smaller transmission time –very relevant for instance under limited bandwidth and/or expensive network connectivity conditions. Also, this reduces the storage cost for the certificate. Nevertheless, a main concern when reducing the size of the certificate is that checking time is not increased as a consequence (among other reasons because pervasive and embedded systems also suffer typically from limited computing –and power– resources). In principle, the consumer could use an analyzer for the purpose of generating the whole fixpoint from scratch, which is still feasible as analysis is automatic. However, this would defeat one of the main purposes of ACC, which is to reduce checking time. The objective of this part is to characterize the smallest subset of the abstraction which must be sent within a certificate –and which still guarantees a single pass checking process– and to design an ACC scheme which generates and validates such reduced certificates.

3.1.2 The Notion of Reduced Certificate

In this part, we will consider analyzers which construct a program *analysis graph* which is interpreted as an abstraction of the (possibly infinite) set of states explored by the concrete execution. The program analysis graph is implicitly represented in the algorithm by means of two global data structures, the *Answer Table* (AT) and the *Dependency Arc Table* (DAT), both initially empty.[1]

- The answer table contains entries of the form $A : CP \mapsto AP$ where A is always a base form[2] and CP and AP are abstract substitutions in D_α. Informally, its entries should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern) CP meets postcondition (or answer pattern), AP."
- The intended meaning of a dependency $A_k : CP \Rightarrow \; B_{k,i} : CP'$ associated to a program rule A_k :- $B_{k,1}, \ldots, B_{k,n}$ with $i \in \{1, ..n\}$, is that the answer for $A_k : CP$ depends on the answer for

[1] Given the information in these, it is straightforward to construct the graph and the associated program-point annotations.

[2] We require that each rule defining a predicate p has identical sequence of variables $x_{p_1}, \ldots x_{p_n}$ in the head atom, i.e., $p(x_{p_1}, \ldots x_{p_n})$. We call this the *base form* of p. This is not restrictive since programs can always be normalized.

$B_{k,i} : CP'$, say AP'. Thus, if AP' changes with the update of some rule for $B_{k,i}$ then, the *arc* $A_k : CP \Rightarrow B_{k,i} : CP'$ must be reprocessed in order to compute the new answer for $A_k : CP$. This is to say that the rule for A_k has to be processed again starting from atom $B_{k,i}$. Thus, dependency arcs are used for forcing recomputation until a fixed-point is reached.

In order to analyze a program, traditional (goal dependent) abstract interpreters for (C)LP programs receive as input, in addition to the program P and the abstract domain D_α, a set $S_\alpha \in AAtom$ of Abstract Atoms (or *call patterns*). Such call patterns are pairs of the form $A : CP$ where A is a procedure descriptor and CP is an abstract substitution (i.e., a condition of the run-time bindings) of A expressed as $CP \in D_\alpha$.

Intuitively, the analysis algorithm is a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, a *prioritized event queue* is used. We use $\Omega \in QHS$ to refer to a *Queue Handling Strategy* which a particular instance of the generic algorithm may use. The concrete analysis algorithm can be found in [54, 89, 9].

A basic idea in ACC is that the full answer table is used as certificate. The key observation in order to reduce the size of such certificates is that certain entries in a certificate may be *irrelevant*, in the sense that the checker is able to reproduce them by itself in a single pass. The notion of *relevance* is directly related to the idea of re-computation in the program analysis graph. Intuitively, given an entry in the answer table $A{:}CP \mapsto AP$, its fixpoint may have been computed in several iterations from $\bot$, AP_0, $AP_1, \ldots$ until AP. For each change in the answer, an event *updated*$(A{:}CP)$ is generated during the analysis. The above entry is relevant in a certificate (under some strategy) when its updates force the re-computation of other arcs in the graph which *depend* on $A{:}CP$ (i.e., there is a dependency from it in the table). Thus, unless $A{:}CP \mapsto AP$ is included in the (reduced) certificate, a single-pass checker which uses the same strategy as the code producer will not be able to validate the certificate.

According to the above intuition, we are interested in determining when an entry in the answer table has been "updated" during the analysis and such changes affect other entries. There is a special kind of updated events which can be directly considered "irrelevant" and corresponds to those updates which force a redundant computation. We write $DAT|_{A:CP}$ to denote the set of arcs of the form $H : CP_0 \Rightarrow B : CP_2$ in the current dependency arc table which depend on $A : CP$, i.e, such that $A : CP = (B : CP_2)\sigma$ for some renaming σ.

Definition 3.1.1 (redundant update) *Let $P \in Prog$, $S_\alpha \in AAtom$ and $\Omega \in QHS$. We say that an event updated$(A : CP)$ which appears in the event queue during the analysis of P for S_α is* redundant *w.r.t. Ω if, when it is generated, $DAT|_{A:CP} = \emptyset$.*

It should be noted that redundant updates can only be generated by updated events for call patterns which belong to S_α, i.e., to the initial set of call patterns. Otherwise, $DAT|_{A:CP}$ cannot be empty. Even if it is possible to fix the strategy and define an analysis algorithm which does not introduce redundant updates, we prefer to follow as much as possible the generic one.

We are interested on finding those entries $A{:}CP$ in the answer table, whose analysis have forced the re-processing of some arcs. Certainly, the re-processing of an arc only may be caused for a non redundant updated event for $A{:}CP$, which introduced all arcs in $DAT|_{A:CP}$ to the prioritized queue. However some updated events are not dangerous. For instance, if the processing of an arc $H : CP_0 \Rightarrow A{:}CP$ has been stopped because of the lack of answer for $A{:}CP$. This arc must be considered as "suspended", since its continuation has not been introduced in the queue. In particular, we do not take into account updated events for $A{:}CP$ which are generated when $DAT|_{A:CP}$ only contains suspended arcs. Note that this case still corresponds to the first traversal of any arc and should not be considered as a reprocessing. The following definition introduce the notion of *suspended arc* during the analysis.

Definition 3.1.2 (suspended arc) *In the conditions of Definition 3.1.1, we say that an arc $H{:}CP_0 \Rightarrow B{:}CP_2$ in the dependency arc table is* suspended *w.r.t. Ω during the analysis of P for S_α iff when it is generated, the answer table does not contain any entry for $B{:}CP_2$.*

For the rest of updated events, its relevance depends strongly on the strategy used to handle the prioritized event queue. For instance, suppose that the prioritized event queue contains an event $arc(H : CP_0 \Rightarrow A{:}CP)$, coming from a suspended arc in DAT. If all updated events for $A{:}CP$ are processed before this arc (i.e., the fixpoint of $A{:}CP$ is available before processing the arc), then these updated events does not force re-computation. Let us define now what it is understood by re-computation.

Definition 3.1.3 (multi-traversed arc) *In the conditions of Definition 3.1.1, we say that an arc $H : CP_0 \Rightarrow A{:}CP)$ in the dependency arc table has been* multi-traversed *w.r.t. Ω after the analysis of P for S_α iff it has been introduced in the dependency arc table at least twice as a non suspended arc.*

We define now the notion of *relevant entry*, which will be crucial to defined reduced certificates. The key observation is that those answer patterns whose computation has generated multi-traversed arcs should be available in the certificate.

Definition 3.1.4 (relevant entry) *In the conditions of Definition 3.1.1 we say that the entry $A{:}CP \mapsto AP$ in the answer table is* relevant *w.r.t. Ω after the analysis of P for S_α iff there exists a multi-traversed arc $\_ \Rightarrow A{:}CP$ in the dependency arc table.*

The notion of *reduced certificate* allows us to remove irrelevant entries from the answer table and produce a smaller certificate which can still be validated in one pass.

Definition 3.1.5 (reduced certificate) *In the conditions of Definition 3.1.1, let* FCert *be the answer table for P and S_α computed by the analysis. We define the* reduced certificate *as the set of relevant entries in* FCert *for Ω.*

3.1.3 Checking Reduced Certificates

In the ACC framework for full certificates the checking algorithm [14] uses a specific graph traversal strategy, say Ω_C. This checker has been shown to be very efficient but in turn its design is not generic with respect to this issue (in contrast to the analysis design). Note however that both the analysis and checking algorithms are always parametric on the abstract domain, with the resulting genericity, which allows proving a wide variety of properties by using the large set of available domains, this being one of the fundamental advantages of ACC. This is not problematic in the context of full certificates since, even if the certifier uses a strategy Ω_A which is different from Ω_C, it is ensured that all valid certificates get validated in one pass by that specific checker. This result does not hold any more in the case of reduced certificates. In particular, *completeness* of checking is not guaranteed if $\Omega_A \neq \Omega_C$. This occurs because though the answer table is identical for all strategies, the subset of redundant entries depends on the particular strategy used. The problem is that, if there is an entry $A{:}CP \mapsto AP$ in FCert such that it is relevant w.r.t. Ω_C but it is not w.r.t. Ω_A, then a single pass checker will fail to validate the RCert generated using Ω_A. Therefore, it is essential in this context to design generic checkers which are not tied to a particular graph traversal strategy. Upon agreeing on the appropriate parameters, the consumer uses the particular instance of the generic checker resulting from the application of such parameters. In a particular application of our framework, we expect that the graph traversal strategy is agreed a priori between consumer and producer. But, if necessary (e.g., the consumer does not implement this strategy), then it could be sent along with the transmitted package.

It should be noted that the design of generic checkers is also relevant in light of current trends in verified analyzers (e.g., [59, 33]), which could be transferred directly to the checking end. In particular, since the design of the checking process is generic, it becomes feasible in ACC to use automatic program transformers to specialize a certified (specific) analysis algorithm in order to obtain a certified checker with the same strategy while preserving correctness and completeness.

In addition to the genericity issue discussed above, an important difference with the checker for full certificates [14] is that there are certain entries which are not available in the certificate and that we want to reconstruct and output in checking. The reason for this is that the safety policy has to be tested w.r.t.

the full answer table. Therefore, the checker must reconstruct, from RCert, the answer table returned by ANALYZE_F, FCert, in order to test for adherence to the safety policy. Note that reconstructing the answer table does not add any additional cost compared to the checker in [14], since the full answer table also has to be created in [14].

The checker of reduced certificates has to detect (and issue) two sources of errors:

a) The answer in the certificate is more precise than the one obtained by the checker. This is the traditional error in ACC and means that the certificate and program at hand do not correspond to each other.

b) Recomputation is required. This should not occur during checking, i.e., no arcs must be multi-traversed by the checker. This second type of error corresponds to situations in which some non redundat update is needed in order to obtain an answer (it cannot be obtained in one pass).

The correctness of the checking process which amount to saying that if CHECKER_R does not issue an error when validating a certificate, then the re-construced answer table is a fixpoint. Regarding completeness, if $\Omega_C = \Omega_A$ then the checker is guaranteed to be complete. Additionally, a checker using a different strategy Ω_C is also guaranteed to be complete as long as the certificate reduced w.r.t Ω_C is equal to or smaller than the certificate reduced w.r.t Ω_A. Furthermore, if the certificate used is full, the checker is complete for any strategy. Note that if $\mathsf{RCert}_{\Omega_A} \not\supseteq \mathsf{RCert}_{\Omega_C}$, CHECKER_R with the strategy Ω_C may fail to validate $\mathsf{RCert}_{\Omega_A}$, which is indeed valid for the program under Ω_A.

3.1.4 Discussion and Related Work

As we have illustrated throughout this chapter, the reduction in the size of the certificates is directly related to the number of *updates* (or iterations) performed during analysis. Clearly, depending on the "quality" of the graph traversal strategy used different instances of the generic analyzer will generate reduced certificates of different sizes. Significant and successful efforts have been made during recent years towards improving the efficiency of analysis. The most optimized analyzers actually aim at reducing the number of updates necessary to reach the final fixpoint [89]. Interestingly, our framework greatly benefits from all these advances, since the more efficient analysis is, the smaller the corresponding reduced certificates are.

A detailed comparison of the technique of ACC with related methods can be found in [14]. In this section, we focus only on those works related to certificate size reduction in PCC. The common idea in order to compress a certificate in the PCC scheme is to store only the analysis information which the checker is not able to reproduce by itself [62]. In the field of abstract interpretation, this is known as *fixpoint compression* and it is being used in different contexts and tools. For instance, in the Astrée analyzer [39] designed to detect runtime errors in programs written in C, only one abstract element by head of loop is kept for memory usage purposes.

With our same purpose of reducing the size of certificates, Necula and Lee [81] designed a variant of the Edinburgh Logical Framework LF [53], called LF$_i$, in which certificates (or proofs) discard part of the information that is redundant or that can be easily synthesized. LF$_i$ inherits from LF the possibility of encoding several logics in a natural way but avoiding the high degree of redundancy proper of the LF representation of proofs. On the producer side, the original certificate is a LF proof is transformed via a representation algorithm into a LF$_i$ representation, which is then used as the final certificate. On the consumer side, LF$_i$ proofs are validated by using a one pass LF type checker which is able to reconstruct on the fly the missing parts of the proof in one pass. Experimental results for a concrete implementation reveal an important reduction on the size of certificates (w.r.t. LF representation proofs) and on the checking time. Although this work attacks the same problem as ours, the underlying techniques used are clearly different. Furthermore, our certificates may be considered minimal, whereas in [81], redundant information is still left in the certificates in order to guarantee a more efficient behaviour of the type checker.

A further step is taken in Oracle-based PCC [82]. This is a variation of the PCC idea that allows the size of proofs accompanying the code to adapt to the complexity of the property being checked such that when PCC is used to verify relatively simple properties such as type safety, the essential information contained in a proof is significantly smaller than the entire proof. The proof as an oracle is implemented as a stream of bits aimed at resolving the non-deterministic interpretation choices. Although the underlying representations and techniques are different from ours, we share with this work the purpose of reducing the size of certificates by providing the checker with the minimal information it requires to perform a proof and the genericity which allows both techniques to deal with different kinds of properties beyond types.

The general idea of certificate size reduction has also been deployed in lightweight bytecode verification (LBV) [92]. LBV is a practical PCC approach to Java Bytecode Verification [62] applied to the KVM (an embedded variant of the JVM). The idea is that the type-based bytecode verification is split in two phases, where the producer first computes the certificate by means of a type-based dataflow analyzer and then the consumer simply checks that the types provided in the code certificate are valid. As in our case, the second phase can be done in a single, linear pass over the bytecode. However, LBV is designed limited to types while ACC generalizes it to arbitrary domains. Also, ACC deals with multivariance with the associated accuracy gains (while LBV is monovariant). Regarding the reduction of certificate size, our work characterizes precisely the minimal information that can be sent for a generic algorithm not tied to any particular graph traversal strategy. [92] sends information for all "backward" jumps while our proposal carries the reduction further because it includes only the analysis information of those calls in the analysis graph whose answers have been *updated*, including both branching and non branching instructions (a more detailed comparison can be found in [9, 88].

The main ideas in ACC have been the basis to build a PCC architecture based on *certified* abstract interpretation in [28]. As in ACC, the code producer generates program certificates automatically by relying on an abstract interpreter. The main difference is that in this method code consumers users proof checkers derived from certified analysers to check certificates. When the consumer does not have the checker, the producer sends the checker together with its soundnesss proof. This soundess proof is then verified automatically by Coq type checker and if the verification succeeds, then the certified checker is installed. They have also provided Coq strategy commands which allow to reconstruct a certificate as a fixpoint which can encode the certificate reduction strategy of LBV.

3.2 Reducing the Width of Certificates

The goal of MOBIUS and in particular Task 4.5 is to produce certificates that fully fit to the memory of the devices that are going to execute programs that these certificates were based on, and still have room to spare for the actual verification of these certificates. Still, we should also be prepared for the case where we cannot sufficiently reduce the footprint of the certificates and their checker. In this case, the next best thing would be the consumption of a certificate in several chunks — the checker receives the first part of the certificate, verifies it wrt. the program, records some key lemmas that were proved, forgets the first part of the certificate, receives the second part of the certificate, verifies it wrt. the program, also using the lemmas that were saved from the first part, saves some new key lemmas, forgets the second part of the certificate and possibly also some lemmas that were saved from the first part, receives the third part of the certificate, etc. The question now is, how to formulate the certificate so that these saved "key lemmas" would consume as little space as possible.

3.2.1 Reducing the Memory Requirements of Type-Checking

For concreteness consider the type-checking of a simple imperative programming language (the WHILE-language) made up of primitive statements (assignments of expressions to variables, skip-statements, maybe something more, for example input/output), sequential composition of programs, conditional statements and while-loops. Assume that we have a set of types D and the type of a program has the form $d_1 \longrightarrow d_2$ for $d_1, d_2 \in D$. I.e. we intend the types in T to somehow describe sets of program states. The type system itself is then shaped as in Fig. 3.1. Here the predicates TF, LT, B_T, B_F and C encode the actual domain-specific claims that have to be true for the type system to be sound. The predicate TF states that $d \longrightarrow d'$

$$\frac{TF(d,p,d')}{\vdash p : d \longrightarrow d'}\ \text{(prim)}$$

$$\frac{\vdash P_1 : d_1 \longrightarrow d_1' \quad \vdash P_2 : d_2 \longrightarrow d_2' \quad LT(d_1', d_2)}{\vdash P_1; P_2 : d_1 \longrightarrow d_2'}\ \text{(seq)}$$

$$\frac{\vdash P_1 : d_1 \longrightarrow d_1' \quad \vdash P_2 : d_2 \longrightarrow d_2' \quad B_T(d,e,d_1) \quad B_F(d,e,d_2) \quad C(d_1', d_2', d')}{\vdash \mathit{if}\ e\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'}\ \text{(if)}$$

$$\frac{\vdash P_1 : d_1 \longrightarrow d_1' \quad B_T(d,e,d_1) \quad B_T(d_1',e,d_1) \quad B_F(d,e,d_2) \quad B_F(d_1',e,d_2) \quad C(d_1', d_2, d')}{\vdash \mathit{while}\ e\ \mathit{do}\ P_1 : d \longrightarrow d'}\ \text{(while)}$$

Figure 3.1: A generic type system for the WHILE-language

$$\mathsf{Cert}(p,T) = \{\{\, TF(d,p,d')\}\}$$

$$\mathsf{Cert}(P_1;P_2,T) = d_1'; \mathsf{Cert}(P_1,T_1); d_2; \{\{LT(d_1',d_2)\}\}; \mathsf{Cert}(P_2,T_2)$$

$$\begin{aligned}\mathsf{Cert}(\mathit{if}\ e\ \mathit{then}\ P_1\ \mathit{else}\ P_2,T) = d_1; \{\{B_T(d,e,d_1)\}\}; d_1'; \mathsf{Cert}(P_1,T_1);&\\ d_2; \{\{B_F(d,e,d_2)\}\}; d_2'; \mathsf{Cert}(P_2,T_2); \{\{(C(d_1',d_2',d')\}\}&\end{aligned}$$

$$\begin{aligned}\mathsf{Cert}(\mathit{while}\ e\ \mathit{do}\ P_1,T) = d_1; \{\{B_T(d,e,d_1)\}\}; d_1'; \mathsf{Cert}(P_1,T_1); \{\{B_T(d_1',e,d_1)\}\};&\\ d_2; \{\{B_F(d,e,d_2)\}\}; \{\{B_F(d_1',e,d_2)\}\}; \{\{C(d_1',d_2,d')\}\}&\end{aligned}$$

Figure 3.2: Certificate $\mathsf{Cert}(P,T)$ of typability according to Fig. 3.1

is a valid type for the primitive statement p. The predicate $LT(d_1', d_2)$ states that all situations that are described by the type d_1' are also describable by the type d_2 (in practice, this amounts to subtyping). The predicates $B_T(d,e,d')$ and $B_F(d,e,d')$ state how the type d' refines d by taking into account also the truth or falsity of e. The predicate $C(d_1,d_2,d)$ states that d is a valid combination of d_1 and d_2. The proofs of all these prdicates (which are domain-specific, too) are also included in the certificate of the whole program; we denote the proof of $P(\ldots)$ by $\{\{P(\ldots)\}\}$. We will describe our method of reducing the width of certificates on the basis of such type-systems. Still, that technique is adaptable to other kinds of verification conditions, too. Sec. 3.2.2 gives a more general view of our technique.

The certificate for a program P with a type-derivation tree T is a list containing proofs of predicates $\mathsf{Cert}(P,T)$ that is defined in Fig. 3.2. Here the meaning of the types $d, d', d_1, d_1', d_2, d_2'$ is the same as in the respective rule in Fig. 3.1. The trees T_1 and T_2 are the first and second subtree (if any) of the tree T. The certificate is checked by a procedure Verify whose arguments are P, the type $d \longrightarrow d'$ and the certificate C. The certificate checker works by parsing the program P, choosing the correct rule from Fig. 3.1, reading the necessary intermediate types and proofs from the certificate, and checking those proofs, recursively invoking itself if necessary. The certificate C in its entirety does not have to be loaded into the memory of the device doing the checking. On the contrary, it is meant to be consumed element-by-element; there are no references back to past types or proofs of predicates. We will not give a formal specification of the procedure Verify here.

The footprint of the procedure Verify is determined by the following:

- The memory requirements of checking the proof of the predicates TF, LT, B_T, B_F, C. We believe that these requirements are not particularly large.
- The number of types d that the verifier has to remember while it is checking the proof of some

$$\mathsf{Cert}^\star(p,T) = \{\{\,TF(d,p,d')\}\}$$

$$\mathsf{Cert}^\star(P_1;P_2,T) = d'_1; \mathsf{Cert}^\star(P_1,T_1); d_2; \{\{LT(d'_1,d_2)\}\}; d'; \mathsf{Cert}^\star(P_2,T_2)$$

$$\begin{aligned}\mathsf{Cert}^\star(\textit{if } e \textit{ then } P_1 \textit{ else } P_2,T) = {} & d_1; \{\{B_T(d,e,d_1)\}\}; d'_1; \mathsf{Cert}^\star(P_1,T_1);\\ & d; d_2; \{\{B_F(d,e,d_2)\}\}; d'_2; \mathsf{Cert}^\star(P_2,T_2); d'_1; d'; \{\{(C(d'_1,d'_2,d')\}\}\end{aligned}$$

$$\begin{aligned}\mathsf{Cert}^\star(\textit{while } e \textit{ do } P_1,T) = {} & d_1; \{\{B_T(d,e,d_1)\}\}; d'_1; \mathsf{Cert}^\star(P_1,T_1); d_1; \{\{B_T(d'_1,e,d_1)\}\};\\ & d; d_2; \{\{B_F(d,e,d_2)\}\}; d'_1; \{\{B_F(d'_1,e,d_2)\}\}; d'; d'_1; \{\{C(d'_1,d_2,d')\}\}\end{aligned}$$

Figure 3.3: Certificate $\mathsf{Cert}^\star(P,T)$ of typability according to Fig. 3.1

subprogram, as these types have to be used again when the proof of the subprogram is complete. For example, while checking the certificate of *if* e *then* P_1 *else* P_2 the type d'_1 has to be remembered while the certificate of the program P_2 is checked.

We aim to reduce the number of types d that have to be simultaneously remembered during the run of the verifier. Our idea [60] is very simple — if some type is used in several places of the certificate checking process then the verifier will still forget that type after the first use, but will expect to find it again in the certificate right before its next use. A certificate $\mathsf{Cert}^\star(P,T)$ that repeats the types in correct places is given in Fig. 3.3. The corresponding verifier will be such, that after verifying the certificate C that some program P has the type $d \longrightarrow d'$, it still has the type d', but not d available (but d' does not have to be available during the entire process of checking C).

The maximum number of simultaneously recorded types d for a verifier using the certificates $\mathsf{Cert}^\star(P,T)$ is 3. Obviously we cannot do better because the combining predicate C uses three types. However, the verifier has to be careful — a malicious code producer does not have to adhere to the certificate generating procedure given in Fig. 3.3. In particular, he may specify different types for different occasions of the same d in Fig. 3.3. This may cause the verifier to accept some programs that cannot be typed.

We overcome the issue of cheating prover by letting the verifier to not completely forget certain types, but still store a *cryptographic message digest* of the types that have been purged from memory. In this way the verifier retains the ability to check that certain types occurring several times in the certificate are indeed equal. The requirement to store the message digests increases the footprint of the verifier. However, a good cryptographic message digest (i.e. the algorithm is not yet considered broken) can take as few as 32 bytes [83]; this is much less than an entire type. Using certain search trees where the structure of the tree has been "cryptographically fixed" [31] may even reduce the memory requirement to a single identifier per saved type (and a single hash value for the entire certificate).

The verifier using the hash function H is given in Fig. 3.4. Its arguments are the program under certification and its claimed type. The certificate is considered to be a global list of types and proofs of predicates TF, LT, B_T, B_F, C. The next element in this list is given by the operation SHIFT. Another operation, $\mathsf{AUTHSHIFT}(h)$ extracts the next element of the certificate and compares its digest to the value of h. If the values match then the extracted element is returned, otherwise the verification fails. The operation $\mathsf{FREE}(d)$ frees the memory held by the type d. The operation $\textit{prim\_verify}(P(\ldots),C)$ will verify the proof C of the predicate $P(\ldots)$. This verification is supposed to free the "source" types of the predicate P, i.e. type d for predicates $TF(d,p,d')$, $LT(d,d')$, $B_T(d,e,d')$, $B_F(d,e,d')$, and the types d_1 and d_2 for the predicate $C(d_1,d_2,d')$.

We have the following theorem of correctness of the procedure Verify^H:

Theorem 3.2.1 *There exists a polynomial-time algorithm $\mathcal{A}^{(\cdot)}$, such that if $\mathsf{Verify}^H(P,d \longrightarrow d')$ (with the*

```
case $P$ of
        $p$ :
                $\mathit{prim\_verify}(\mathit{TF}(d, p, d'), \mathsf{SHIFT})$
        $P_1; P_2$ :
                $h' := H(d');\ \mathsf{FREE}(d');$
                $d'_1 := \mathsf{SHIFT};\ \mathsf{Verify}(P_1, d \longrightarrow d'_1);$
                $d_2 := \mathsf{SHIFT};\ \mathit{prim\_verify}(\mathit{LT}(d'_1, d_2), \mathsf{SHIFT});$
                $d' := \mathsf{AUTHSHIFT}(h');\ \mathsf{Verify}(P_2, d_2 \longrightarrow d')$
        $\mathit{if}\ e\ \mathit{then}\ P_1\ \mathit{else}\ P_2$ :
                $h' := H(d');\ \mathsf{FREE}(d');$
                $d_1 := \mathsf{SHIFT};\ h := H(d);\ \mathit{prim\_verify}(B_T(d, e, d_1), \mathsf{SHIFT});$
                $d'_1 := \mathsf{SHIFT};\ \mathsf{Verify}(P_1, d_1 \longrightarrow d'_1);$
                $h'_1 := H(d'_1);\ \mathsf{FREE}(d'_1);$
                $d := \mathsf{AUTHSHIFT}(h);\ d_2 := \mathsf{SHIFT};\ \mathit{prim\_verify}(B_F(d, e, d_2), \mathsf{SHIFT});$
                $d'_2 := \mathsf{SHIFT};\ \mathsf{Verify}(P_2, d_2 \longrightarrow d'_2);$
                $d'_1 := \mathsf{AUTHSHIFT}(h'_1);\ d' := \mathsf{AUTHSHIFT}(h');$
                $\mathit{prim\_verify}(C(d'_1, d'_2, d'), \mathsf{SHIFT})$
        $\mathit{while}\ e\ \mathit{do}\ P_1$ :
                $h' := H(d');\ \mathsf{FREE}(d');$
                $d_1 := \mathsf{SHIFT};\ h := H(d);\ \mathit{prim\_verify}(B_T(d, e, d_1), \mathsf{SHIFT});$
                $d'_1 := \mathsf{SHIFT};\ h_1 := H(d_1);\ \mathsf{Verify}(P_1, d_1 \longrightarrow d'_1);$
                $d_1 := \mathsf{AUTHSHIFT}(h_1);\ h'_1 := H(d'_1);\ \mathit{prim\_verify}(B_T(d'_1, e, d_1), \mathsf{SHIFT});\ \mathsf{FREE}(d_1);$
                $d := \mathsf{AUTHSHIFT}(h);\ d_2 := \mathsf{SHIFT};\ \mathit{prim\_verify}(B_F(d, e, d_2), \mathsf{SHIFT});$
                $d'_1 := \mathsf{AUTHSHIFT}(h'_1);\ \mathit{prim\_verify}(B_F(d'_1, e'd_2), \mathsf{SHIFT});$
                $d' := \mathsf{AUTHSHIFT}(h');\ d'_1 := \mathsf{AUTHSHIFT}(h'_1);\ \mathit{prim\_verify}(C(d'_1, d_2, d'), \mathsf{SHIFT})$
```

Figure 3.4: $\mathsf{Verify}^H(P, d \longrightarrow d')$

global certificate C) accepts for some H, P, d, d' and C, but $\not\vdash P : d \longrightarrow d'$, then $\mathcal{A}^H(P, d, d', C)$ outputs a collision of H.

3.2.2 Authenticated Paging

The previous technique can really be considered as the use of some external storage for paging purposes by the certificate-checking device. Whenever the footprint of the checker overextends the physical memory of the device, some inactive memory pages are sent to the external storage. When these pages are needed again, they are swapped back in. However, the device does not trust the external storage to not modify the swapped-out pages. Hence it stores the cryptographic message digest of those pages whenever they are swapped out; and compares that digest to the hash of the pages that are swapped back in. If the hash values do not match, the computation stops and the certificate is not deemed valid.

Where does the external storage come from? If the certificate generator knows all the inner workings of the certificate checker then the certificate itself can serve as the external storage. In this case the certificate generator knows precisely when the checker will swap the pages out or in, and what the contents of these pages will be. Hence the certificate generator is able to include the contents of swapped-in pages in the correct places of the certificate. Swapping out the pages becomes trivial — these pages are simply deleted from memory, after storing their cryptographic message digest.

3.3 Tables of Lemmas

A sequence of well chosen lemmas is often an important part of presenting a proof in, at least, informal mathematics. In some situations, one might feel that the sequence of lemmas itself could constitute an actual proof, particularly if the reader of the proof has significant mathematical means to fill in the gaps between the lemmas. Of course, as lemmas at the beginning of the list are proved, they can be used to help prove lemmas later in the list. (Most of the material in this section is taken from the paper [74]).

Although generating lemmas is a well known and critical activity in mathematical proof, producing and using such lemmas can be important in, say, logic programming, deductive databases, and model checking. In such settings, the underlying proofs that such systems attempt to build are usually *cut-free* (that is, they lack the use of lemmas). That does not mean, however, that lemmas (and, hence, the cut-inference rule) do not have a role in improving the search for or the presentation of proofs. As is well-known, using cut to present proofs can often greatly reduce the size of proofs.

Consider attempting to prove the conjunctive query $B \wedge C$ from a logic program Γ. This attempt can be reduced to first attempting to prove B from Γ and then C from Γ. It might well be the case that during the attempt to prove C, many subgoals might need to be proved that were previously established during the attempt to prove B. Of course, if proved subgoals can be remembered from the first conjunct to the second, then it might be possible to build smaller proofs and these might be easier to find and to check for correctness. Some implemented logic programming systems use tables in this fashion: for example, in XSB [91] and in Twelf [85], it is possible to specify that some predicates should be *tabled*: that is, whenever an atomic formula with such a predicate is successfully proved, that atomic formula is remembered, so that, any other time a proof of that atom is attempted, the proof process can be stopped with a success.

If we wish to consider a table as a sequence of lemmas, we are confronted immediately with the following problems.

(*i*) *How are tabled formulas entered into the proof context?* Proofs will be sequent calculus proofs, and tables will be partially ordered collections of formulas. In a straightforward fashion, the cut-inference rule is used to state the obligation to prove a tabled lemma as well as insert it into the main proof context.

(*ii*) *How do we avoid reproving tabled formulas.* It is easy to provide algorithmic means for making certain that formulas are not reproved (for example, prior to attempting a proof of a formula, check if that formula is in the table). More challenging is to find a purely proof theoretic solution in which the only proofs that can be built are those in which reproving cannot happen. We achieve this by first restricting tables to be literals (a typical assumption in implementations of tabling). Second, we exploit some recent developments in the understanding of *focused* proofs in intuitionistic logic that allow literals to be given different *polarity.* Polarity can be used to signal that a literal is in or out of the table. Focused proof search can then be organized so that a tabled literal is not reproved.

In its most general form, we consider a table as a partially ordered finite set of formulas.

Definition 3.3.1 *A* table *is a tuple $\mathcal{T} = \langle \mathcal{A}, \preceq \rangle$, where $\mathcal{A}$ is some finite set of formulas, and $\preceq$ is a partial order relation over the elements of $\mathcal{A}$.*

The intended meaning of a table is that it is a structured collection of provable formulas (from some assumed context, say, Γ). The order relationship $B \preceq C$ denotes the fact that the proof of the formula B is available for reuse during a proof attempt of C: that is, if an attempt to prove the formula C results in the subgoal B then proof search can stop immediately since B has a proof.

The following definition describes how a table can be translated to a collection of multicut inference rules: the multicut inference rules is a frequently used generalization of the cut inference rule [45, 99].

Definition 3.3.2 *Let $\mathcal{T} = \langle \mathcal{A}, \preceq \rangle$ be a table. The* multicut derivation *for $\mathcal{T}$ and the sequent $\mathcal{S} = \Gamma \longrightarrow G$, written as $mcd(\mathcal{T}, \mathcal{S})$, is defined inductively as follows: if $\mathcal{A}$ is empty, then $mcd(\mathcal{T}, \mathcal{S})$ is the derivation containing just the sequent $\Gamma \longrightarrow G$. Otherwise, if $\{A_1, \ldots, A_n\}$ is the collection of $\preceq$-minimal elements in $\mathcal{A}$ and if Π is the multicut derivation for the smaller table $\langle \mathcal{A} \setminus \{A_1, \ldots, A_n\}, \preceq \rangle$ and the sequent $\Gamma, A_1, \ldots, A_n \longrightarrow G$, then $mcd(\mathcal{T}, \mathcal{S})$ is the derivation*

$$\frac{\Gamma \longrightarrow A_1 \quad \cdots \quad \Gamma \longrightarrow A_n \qquad \begin{array}{c}\Pi\\ \Gamma, A_1, \ldots, A_n \longrightarrow G\end{array}}{\Gamma \longrightarrow G}\ mc$$

Multicut derivations are always open *derivations (that is, they contain leafs that are not proved). A* proof of a multicut derivation *is any (closed) proof that extends this open derivation.*

3.3.1 Focusing and polarities

In order to present a focused proof system, we first classify the connectives $\wedge$, $\exists$, *true* and $\perp$ as *synchronous* (their right introduction is not necessarily invertible) and the connectives $\supset$, and $\forall$ as *asynchronous* (their right introduction rules are invertible). This dichotomy must also be extended to atomic formulas: some atoms are considered asynchronous and the rest are considered synchronous. Since the terms "asynchronous" and "synchronous" do not apply well to atomic formulas, we shall instead use the slightly more general notions of *polarity* for a formula. In particular, a formula is positive if its main connective is synchronous or it is a *positive atom* and is negative if its main connective is asynchronous or it is a *negative atom.* The polarity of atoms is not necessarily fixed: we shall assign different polarities to atoms to achieve different purposes.

Although the notion of *focused proof* was originally given by Andreoli for linear logic [19], we shall use the recently designed *LJF* focused proof system for intuitionistic logic [63] displayed in Figure 3.5. This system has four types of sequents.

1. The sequent $[\Gamma] -_A\!\!\rightarrow$ is a *right-focusing* sequent (the focus is A);
2. The sequent $[\Gamma] \xrightarrow{A} [R]$: is a *left-focusing* sequent (with focus on A);
3. The sequent $[\Gamma], \Theta \longrightarrow \mathcal{R}$ is an *unfocused sequent.* Here, Γ contains negative formulas and positive atoms, and $\mathcal{R}$ is either in brackets, written as $[R]$, or without brackets;
4. The sequent $[\Gamma] \longrightarrow [R]$ is an instance of the previous sequent where Θ is empty.

$$\frac{[N,\Gamma] \xrightarrow{N} [R]}{[N,\Gamma] \longrightarrow [R]}\,D_l \qquad \frac{[\Gamma] -_{P}\!\!\rightarrow}{[\Gamma] \longrightarrow [P]}\,D_r \qquad \frac{[\Gamma], P \longrightarrow [R]}{[\Gamma] \xrightarrow{P} [R]}\,R_l \qquad \frac{[\Gamma] \longrightarrow N}{[\Gamma] -_{N}\!\!\rightarrow}\,R_r$$

$$\frac{[\Gamma, N_a], \Theta \longrightarrow \mathcal{R}}{[\Gamma], \Theta, N_a \longrightarrow \mathcal{R}}\,[]_l \qquad \frac{[\Gamma], \Theta \longrightarrow [P_a]}{[\Gamma], \Theta \longrightarrow P_a}\,[]_r$$

$$\frac{}{[\Gamma] \xrightarrow{A_n} [A_n]}\,I_l \qquad \frac{}{[\Gamma, A_p] -_{A_p}\!\!\rightarrow}\,I_r$$

$$\frac{}{[\Gamma], \Theta, \bot \longrightarrow \mathcal{R}}\,\mathit{false}_l \qquad \frac{[\Gamma], \Theta \longrightarrow \mathcal{R}}{[\Gamma], \Theta, \mathit{true} \longrightarrow \mathcal{R}}\,\mathit{true}_l \qquad \frac{}{[\Gamma] -_{\mathit{true}}\!\!\rightarrow}\,\mathit{true}_r$$

$$\frac{[\Gamma], \Theta, A, B \longrightarrow \mathcal{R}}{[\Gamma], \Theta, A \wedge B \longrightarrow \mathcal{R}}\,\wedge_l \qquad \frac{[\Gamma] -_{A}\!\!\rightarrow \quad [\Gamma] -_{B}\!\!\rightarrow}{[\Gamma] -_{A\wedge B}\!\!\rightarrow}\,\wedge_r$$

$$\frac{[\Gamma] -_{A}\!\!\rightarrow \quad [\Gamma] \xrightarrow{B} [R]}{[\Gamma] \xrightarrow{A\supset B} [R]}\,\supset_l \qquad \frac{[\Gamma], \Theta, A \longrightarrow B}{[\Gamma], \Theta \longrightarrow A \supset B}\,\supset_r$$

$$\frac{[\Gamma], \Theta, A \longrightarrow \mathcal{R}}{[\Gamma], \Theta, \exists y A \longrightarrow \mathcal{R}}\,\exists_l \qquad \frac{[\Gamma] -_{A[t/x]}\!\!\rightarrow}{[\Gamma] -_{\exists x A}\!\!\rightarrow}\,\exists_r \qquad \frac{[\Gamma] \xrightarrow{A[t/x]} [R]}{[\Gamma] \xrightarrow{\forall x A} [R]}\,\forall_l \qquad \frac{[\Gamma], \Theta \longrightarrow A}{[\Gamma], \Theta \longrightarrow \forall y A}\,\forall_r$$

Figure 3.5: The LJF system [63] originally has one disjunction and two conjunctions, $\wedge^+, \wedge^-$. In this chapter, we only need one conjunction: we will drop $\wedge^-$ and write $\wedge$ for $\wedge^+$. Here A_n denotes a negative atom, A_p a positive atom, P a positive formula, N a negative formula, N_a a negative formula or an atom, and P_a a positive formula or an atom. All other formulas are arbitrary and y is not free in Γ, Θ or R.

As an inspection of the inference rules of LJF reveals, the search for a *focused* proof is composed of two alternating phases, and these phases are governed by polarities. The *asynchronous phase* applies invertible (asynchronous) rules until exhaustion: no backtracking during this phase of search is needed. The asynchronous phase uses the third type of sequent above (the unfocused sequents): in that case, Θ contains positive or negative formulas. If Θ contains positive formulas, then an introduction rule (either $\wedge_l, \exists_l$, true_l, or *false*$_l$) is used to decompose it; if it is negative, then the formula is moved to the Γ context (by using the $[]_l$ rule). The end of the asynchronous phase is represented by the fourth type of sequent. Such a sequent is then established by using one of the decide rules, D_r or D_l. The application of one of these decide rules then selects a formula for focusing and switches proof search to the *synchronous phase* or *focused phase*. This focused phase then proceeds by applying sequences of inference rules on focused formulas: in general, backtracking may be necessary in this phase of search. The focusing phase ends with one of the *release rule* R_l or R_r.

As is pointed out in [63], if all atoms are given negative polarity, the resulting proof system models backward chaining proof search and uniform proofs [73]. If positive atoms are permitted as well, then forward chaining steps can also be accommodated.

We now present the LJF^t proof system that extends LJF by adding a multicut rule and by allowing atoms to have different polarity on the different branches of the multicut rule. In particular, occurrences of atoms in LJF^t proofs are assigned polarities in the following fashion: all atoms are initially given negative polarity: thus proof search with such atoms is the usual goal-directed search. When an atom is inserted into a proof context via a multicut inference rule, that atom's occurrences on the right-most branch will have positive polarity: in principle, a forward chaining discipline is used on that atom on that branch, and it is this discipline that is used to implement the reuse policy on that part of the multicut derivation.

The sequents in LJF^t are the same four kinds of sequents except that we add a polarity declaration, $\mathcal{P}$, to all of them: if an atom appears in the set of atoms $\mathcal{P}$, then it is considered positive; otherwise it is considered negative. Recall also that literals are either atomic formulas or negated atomic formulas (and that $\neg A$ is encoded as $A \supset \bot$). The multicut rule is the only rule that can change the declaration $\mathcal{P}$. In particular, the *polarized version* of the multicut rule is given as

$$\frac{\mathcal{P}; [\Gamma] \longrightarrow [L_1] \qquad \cdots \qquad \mathcal{P}; [\Gamma] \longrightarrow [L_n] \qquad \mathcal{P} \cup \Delta_P; [\Gamma \cup \Delta_L] \longrightarrow [R]}{\mathcal{P}; [\Gamma] \longrightarrow [R]}\ mc.$$

Here, $\Delta_L = \{L_1, \ldots, L_n\}$ is a set of literals and $\Delta_P = \{A \mid A \in \Delta_L \text{ or } \neg A \in \Delta_L\}$ is the set of all atoms in Δ_L. Notice that the literals in Δ_L are cut-formulas and that the atoms in Δ_P switch their polarity from negative in the conclusion of this rule to positive in the right-most premise. Whenever we use this multicut inference rule, we shall arrange things so that the sets Δ_P and $\mathcal{P}$ are disjoint.

As the notion of polarity of an atom is now declared via $\mathcal{P}$ instead of being globally fixed as in LJF, the inference rules in LJF^t must be adapted accordingly from LJF: for example, the LJF^t rule I_r^t will be derived from the LJF rule I_r as follows:

$$\frac{}{\mathcal{P}; [\Gamma, A_p] \overset{A_p}{-\!\!\longrightarrow}}\ I_r^t, \text{ where } A_p \in \mathcal{P}.$$

In general, if the name of a rule is R in LJF, the corresponding rule in LJF^t is R^t. The following proposition can be proved by a simple induction on the depth of the cut free proofs.

Proposition 3.3.3 *LJF^t is sound and complete with respect to LJF.*

3.3.2 Tables of finite successes

In this section, we restrict our attention in two directions. First, we shall only consider tables containing atomic formulas. Such a restriction is familiar from such implemented tabling systems as [91, 85] where the only items placed in a table are atomic formulas. Second, we shall only allow logic specifications to be Horn clauses, which are defined as D-formulas in the following grammar.

$$G := true \mid A \mid G_1 \wedge G_2 \mid \exists x\, G \qquad\qquad D := A \mid G \supset A \mid D_1 \wedge D_2 \mid \forall x\, D$$

As a consequence of these restrictions, we shall only be tabling atoms if they are proved by "finite success": this contrasts with the situation addressed in the next section where tables can contain negated atoms if "finite failure" is successful to prove them. The restriction to Horn clause formulas is critical for the results here since such clauses ensure that the goal-reduction phase can be seen as completely synchronous. Goals with implications and universal quantifiers causes goal-reduction to mix synchronous and asynchronous phases. Therefore, allowing them can cause the focus of proof search to be broken before positive atomic formulas are encountered.

The following proposition states that a multicut derivation of a provable sequent (using the polarized version of the multicut rule) can be extended to a valid focused proof.

Proposition 3.3.4 *Let Γ be a collection of Horn clauses, G be a G-formula, and let $\mathcal{T}$ be a table of atoms, all of which are provable from Γ (the partial order is not restricted). The sequent $\Gamma \longrightarrow G$ is intuitionistically provable if and only if the open derivation $mcd(\mathcal{T}, \Gamma \longrightarrow G)$ can be extended to a proof in LJF^t.*

The next proposition shows that polarities can be used to guarantee that any tabled atomic formula that has been proved once (and, hence, has positive polarity) will not be reproved. This proposition is proved by induction on the depth of the proof tree.

Proposition 3.3.5 *Let Γ be a set of Horn clauses, $A \in \mathcal{P} \cap \Gamma$, and Ξ be an arbitrary LJF^t proof tree for $\mathcal{P}; [\Gamma] \overset{G}{-\!\!\longrightarrow}$. Then every occurrence of a sequent with right-hand side the atom A is the conclusion of an I_r^t rule.*

Since all the lemmas of a table are included as positive atoms in the right branch of its multicut derivation, all the proofs of any lemma in this branch will be composed of a single rule I^t_r.

A simple example to illustrate this use of positive polarity, considered a goal directed proof of $A \wedge G$. Here, proof search would attempt to prove A and G separately. The proof of G might reduce further to many attempts to prove A and these attempts would be retried in a naive system. Focused proof search with polarities can be less naive: instead of doing the goal reduction illustrated on the left below, use a multicut as is illustrated on the right:

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow G}{\Gamma \longrightarrow A \wedge G} \quad \Longrightarrow \quad \frac{\mathcal{P}; [\Gamma] \longrightarrow [A] \qquad \mathcal{P} \cup \{A\}; [\Gamma, A] \longrightarrow [A \wedge G]}{\mathcal{P}; [\Gamma] \longrightarrow [A \wedge G]}\ mc.$$

In this way, all attempts to prove A on the right will be trivial applications of the initial rule.

When the asynchronous phase of proof search ends, that is, when all the invertible rules have been applied, the decide rules, namely D^t_l and D^t_r, chose a formula on which search should focus. Since logic programs generally contain many formulas, the choice made by these decide rules is a form of *don't know* non-determinism, which is a potential source of backtracking. For example, while the sequent $[A_1, A_1 \supset A_0, A_2 \supset A_0] \longrightarrow [A_0]$ has four formulas on which to focus, a valid LJF proof can be built on by focusing on the formula $A_1 \supset A_0$ (here, A_0, A_1, A_2 are atomic formulas).

To this point, we have been mainly interested in the use of tables and not with their discovery. We consider now an example of how a table can be built. In particular, a cut-free LJF proof Ξ of $\Gamma \longrightarrow G$ can be made into a table as follows. The table consists of all atoms that are on the right-hand side of some sequent in Ξ. The occurrences of proved atoms in Ξ can be ordered using post-order traversal (*i.e.*, process a node's premises before processing the node). The final order used for the table (which is on atomic formulas and not their occurrences) is then obtained from this post-order traversal by retaining only the first occurrence of any repeated atomic formula. The following proposition shows that it is trivial to extend a multicut derivation that is built in this way from a complete proof: the following definition helps to formalize what we mean as trivial here.

Definition 3.3.6 *The* decide-depth *of an LJF^t proof Ξ is the maximum number of occurrences of decide rules (*i.e.*, D_r and D_l) on any path from the root to a leaf in Ξ.*

Proposition 3.3.7 *Let Ξ be a LJF proof of $\Gamma \longrightarrow G$ and let $\mathcal{T}$ be a table obtained from Ξ using the post-order traversal described above. There exists a proof for $mcd(\mathcal{T}, [\cdot]\Gamma \longrightarrow G)$ such that all of its added subproofs have decide-depth of at most one.*

Given that it is simple to check if a table is derived from a cut-free proof, one might consider that the table is, in fact, a legitimate proof object. Within the proof carrying code framework [80], it might be more interesting to send an ordered collection of atoms to represent a proof than to send some more complex representation of a sequent calculus proof tree. This is what is done in the setting of *Abstraction Carrying Code* [?]: we will return to this aspect of tables in Section 3.3.3.

It is possible to generalize the previous section so that it is possible to table the negated atom, say $\neg A$, if the proof attempt for A lead to a finite failure. Tabling of such negated atoms is described in [74]. A critical step requires changing the status of logic programs from *theories* (which allows an *open-world assumption* and no proof theoretic notion of negation) with *definitions*, a.k.a, *fixed points* (which enforce the closed-world assumption and supports a natural approach to negation-as-finite-failure).

3.3.3 Table as proof objects

We have illustrated how tables can be incorporated into proofs. To what extent can we think of tables as proofs themselves? Of course, this question is best addressed when one knows what one will do with a proof.

In the *proof carrying code* setting, proof objects are transmitted together with mobile code to assure that some (safety) properties are satisfied by these programs. Before a client executes the transmitted code the client checks that the proof that that code is carrying proves the program's safety. Thus, proof objects must be engineered so that they are not too large (in order to reduce transmission costs) and not too complex to check (in order to reduce resource requirements on client proof checkers).

Tables might well be a good format for proofs in this setting for several reasons. First, tables represent declarative information and not procedural information: in particular, tables only describe what is provable and does not go into detail about how things are proved. Proof checking can then be organized around simple proof search engines that implement, for example, LJF. The trade-offs between proof size and proof checking time are fairly clear: if the producer of a proof tables all successfully proved atoms (as in Proposition 3.3.7) then tables can be large but proof checking can be simple (only proofs of decide-depth 1 must be considered in extending a multicut derivation). On the other hand, if some atomic formulas are not tabled, then the client may have to reprove them: clearly, reproving some atomic formulas might be rather straightforward and something that a client might be willing to do to help reduce the size of a transmitted proof.

In [94], Roychoudhury *et.al.* propose using tables to build *justifications* that can be seen as a kind of proof. In their setting, these proof objects serve to explain why a logic program can or cannot prove a given atom. They argue that their justification can be used within model checkers and parsers. It seems likely that our use of tables as proofs can be used in these settings as well.

We now consider two examples where tables relate to more than just proofs: they can also be simulations (Example 6) and winning strategies (Example 7). These examples also illustrate that non-Horn examples can also be used in the framework that was described above.

Example 6 *Encode a noetherian abstract labeled transition system as a definition by writing the transition* $P \xrightarrow{A} P'$ *as the clause* $one(p,a,p') \triangleq true$. *McDowell* et.al. *showed in [70] that the additional definition clause*

$$\forall P, Q[sim(P,Q) \triangleq \forall A, P'.one(P,A,P') \supset \exists Q'.one(Q,A,Q') \wedge sim(P',Q')]$$

can be used to compute the simulation relation. In particular, processes P is simulated by Q if and only if the atomic formula $sim(P,Q)$ is provable. (Bisimulation can be encoded using a slightly more complex definition.) Moreover, if Ξ is a cut-free proof of that atomic formula and if $\mathcal{S}$ is the set of all pairs $\langle t,s\rangle$ such that Ξ contains a subproof of $sim(t,s)$, then $\mathcal{S}$ is a simulation. Furthermore, let $\preceq$ be the post-order relation on $\mathcal{S}$ derived from Ξ as described in Section 3.3.2. Notice that it is now a simple matter to check that $\mathcal{S}$ is, in fact, a simulation by treating it as a table and considering extending its induced multicut derivation to a complete proof. In particular, let $\langle p,q\rangle \in \mathcal{S}$ and let $\mathcal{P} = \{sim(t,s) \mid \langle t,s\rangle \in \mathcal{S},\ \text{and } sim(t,s) \prec sim(p,q)\}$. An attempt to prove the sequent $\mathcal{P};[\mathcal{P}] \longrightarrow sim(p,q)$ yields a proof of the form

$$\dfrac{\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{P};[\mathcal{P}]-_{true}\!\!\rightarrow}\ true_r^t}{\mathcal{P};[\mathcal{P}]-_{one(q,a,q')}\!\!\rightarrow}\ \mathrm{Def}_r \qquad \dfrac{}{\mathcal{P};[\mathcal{P}]-_{sim(p',q')}\!\!\rightarrow}\ I_r^t}{\mathcal{P};[\mathcal{P}]-_{one(q,a,q')\wedge sim(p',q')}\!\!\rightarrow}\ \wedge_r^t}{\mathcal{P};[\mathcal{P}]-_{\exists Q'.one(q,a,p')\wedge sim(p',Q')}\!\!\rightarrow}\ \exists_r^t}{\mathcal{P};[\mathcal{P}] \longrightarrow [\exists Q'.one(q,a,Q') \wedge sim(p',Q')]}\ D_r \qquad \cdots}{\mathcal{P};[\mathcal{P}], one(p,A,P') \longrightarrow [\exists Q'.one(q,A,Q') \wedge sim(P',Q')]}\ \mathrm{Def}_l}{\mathcal{P};[\mathcal{P}] \longrightarrow \forall A, P'.\ one(p,A,P') \supset \exists Q'.\ one(q,A,Q') \wedge sim(P',Q')}\ \forall_l^t, \supset_l^t, []_r^t$$

The ellipses represents that there are other premises generated by the Def_l *rule that introduces the atom* $one(p,A,P')$*: there is one premise for each pair* $\langle a',p'\rangle$ *such that* $p \xrightarrow{a'} p'$ *(if there is none, then the proof is completed at this point). Notice that the only* Def_r *rule in this proof is on the one-step transition and since these are given via a simple list of clauses, finding a* q' *such that* $q \xrightarrow{a'} q'$ *is a simple computation.*

Example 7 *Consider a game between two players, named 1 and 2, who alternate in playing (consider tic-tac-toe) and that one player wins when the other player cannot move. We assume that the state of the game* *is encoded as a term in the logic and that the binary predicate* $move(P, Q)$ *encodes the fact that there is move from position* P *to* Q*. Furthermore, assume that there are no infinite plays. Then there is a winning strategy from the position* P *if and only if the atom* $win(P)$ *is provable from a definition that includes the clause*

$$\forall P[win(P) \stackrel{\Delta}{=} \forall P'.\ move(P, P') \supset \exists Q.\ move(P', Q) \wedge win(Q)]$$

as well as the (Horn clause) definition of $move(P, Q)$*. As with the previous example, let* Ξ *be a proof of the atom* $win(p)$*, let* $\mathcal{W}$ *be the set of atoms of the form* $win(P)$ *that are proved in subproofs of* Ξ*, and let* $\preceq$ *be the post-order traversal ordering of* $\mathcal{W}$ *based on* Ξ*. It is now a simple matter to verify that* $\mathcal{W}$ *encodes a winning strategy: simply build the multicut derivation associated to the table* $\mathcal{W}$ *and extend it to a complete proof. This later step is essentially the same kind of restricted proof search that is presented for the previous example based on simulation.*

3.3.4 Discussion

We have argued above that tables can, at times, be viewed as proofs. We have connected tables with sequent calculus and with the notion of fixed points (models) for a collection of Horn clauses (inductively defined relational clauses). Tables are essentially fixed points that have been sorted in such a way that atomic formulas earlier in the table are (re)used to prove atomic formulas later in the table. As described in Section 3.1.1, tabling of proved atomic formulas plays a central role in the construction of proofs and certificates within *Abstraction Carrying Code* (ACC) framework [?]. In particular, the post-order traversal of a cut-free proof yields a linear ordered collection of atoms. If one communicates a fixed point by sending the smallest elements of the table first, proving the later elements involves simpler proofs since previously proved atoms can be reused. If all atoms from the cut-free proof are tabled (as described in Proposition 3.3.7) then checking the next element of a table involves finding a trivial proof (having decide-depth of at most one). Of course, there is a great deal of room here for compressing tables further. For example, not all atoms in a cut-free proof need to be retained: for example, if the checker is willing to do some bounded proof search, the decide-depth could be increase beyond one, in which case fewer atoms need to be explicitly stored in the table and communicated to the proof checker. There might also be atomic formulas that the client is willing to reprove at all times: for example, since abstract states of a static analyzer might involve lists and since operations such as concatenation of lists might need to be preformed, it might be sensible to have the proof checker recompute the result of concatenating two lists instead of transmitting that result.

As they are described here, tables can be more general than just least fixed points of Horn clauses. Negated atoms can also be tabled. Tables can also be used to store greatest fixed points and can be used to store simulations and bisimulations (in the process calculus sense) as well as winning strategies in the theory of games. In both of these situations, it not important to store all the transitions a process can make nor all the moves an opponent can make: such computations are part of the asynchronous phase of proof construction (since all such transitions and moves must be checked, there are no choices there): only the response to easy step or move needs to be retained (which is provided by the synchronous phase of proof search).

We expect that similar results will also allow us to relate sequent calculus proofs to other proof objects: in particular, the *oracles* of Necula and Rahul [82] should be closely related since oracles are used to resolve non-determinism in ways that seem closely related to the classification of proof search steps into asynchronous and synchronous phases.

3.4 Generation of Flexible Certificates

We develop our techniques to generate flexible certificates in the context of *Abstraction-Carrying Code* (ACC) and in particular in its adaptation to Java-like languages [79]. As alluded in Section 3.1, ACC is an approach to mobile code safety which uses abstract interpretation as enabling technology. This framework relies on an expressive class of safety policies which can be defined over different abstract domains. In the

ACC model, we use an *abstraction* (or abstract model) of the program computed by standard static analyzers as a certificate. The validity of the abstraction on the consumer side is checked in a single pass by a very efficient and specialized abstract-interpreter. By relying on this framework, we have developed the COSTA system (COSt and Termination Analyzer) as the enabling technology to generate flexible resource-related (i.e., *cost and termination) certificates* for Java bytecode.

COSTA is a fully automatic static analyzer which receives as input a bytecode program and a selection of *resources* of interest, and tries to bound the resource consumption of the program with respect to such a *cost model.* COSTA provides several non-trivial notions of resource which allow the different consumers to impose different safety policies, such as the consumption of the heap, the number of bytecode instructions executed, the number of calls to a specific method (e.g., the library method for sending text messages in mobile phones), etc. The system uses the same machinery to infer *upper bounds* on cost, and for proving *termination* (which also implies the boundedness of any resource consumption).

In this deliverable, we describe the techniques used to improve the efficiency and the accuracy of the analyzer with the aim of having more practical and more expressive resource-related safety policies and certificates:

- Improving the efficiency of the tool is essential in order to have a practical tool able to generate resource-related certificates from realistic programs. In this regard, our focus has been on identifying variables which are *useless* in the sense that they do not affect the resource consumption of the program and therefore can be ignored by cost analysis. In Section 3.4.1, we identify two classes of useless variables and propose automatic analysis techniques to detect them. The first class corresponds to *stack variables* that can be replaced by program variables or constant values. The second class corresponds to variables whose value is *cost-irrelevant*, i.e., does not affect the cost of the program. We propose an algorithm, inspired in *static slicing* which safely identifies cost-irrelevant variables. Eliminating useless variables clearly results in a more efficient analysis, which is crucial for the practical uptake of our technology.

- Improving the accuracy of the tool is crucial in order to have a static analyzer which is expressive enough to generate resource-related certificates and cover different safety policies for a wide range of programs. In this regard, one of such problems is to find a practical solution to infer cost and termination when the number of iterations of loops is affected by *numeric fields.* We have performed statistics on the Java libraries to see how often this happens in practice and we found that in 12.95% of cases, the number of iterations of loops (and therefore cost and termination) explicitly depends on values stored in fields and, in the vast majority of cases, such fields are numeric. In Section 3.4.2, we focus on the problem of termination with numeric fields, as studying the problem in the context of termination is strictly simpler than doing it in cost analysis. Inspired by the examples found in the libraries, we have identified a series of difficulties that need to be solved in order to deal with numeric fields in termination and propose some ideas towards a lightweight analysis which is able to prove termination of sequential Java-like programs in the presence of numeric fields.

Finally, in Section 3.4.4, we will show the architecture of COSTA and how these techniques are integrated in the system. We will briefly describe how to use the tool by means of the recently developed COSTA web interface. The web interface allows users to try out the system on a set of representative examples, and also to upload their own bytecode programs.

3.4.1 Improving the efficiency of the generation of certificates

Several cost analyses exist for a wide variety of programming languages, including logic [41, 42, 78], functional [26, 47, 90, 95, 93] and imperative languages [5, 57, 26]. In general, given an input program, cost analysis infers a Cost Relation System (CRS), which is a general form of describing the resource consumption of programs w.r.t. the cost model of interest. Different cost models can be used to capture different aspects of the computation, such as the number of bytecode instructions executed, [6], the memory (heap) consumption, [11], etc. Intuitively, CRSs are systems of *recursive* equations which express the cost of a part of the program in terms of the cost of the parts which may follow it, according to the program structure. All kinds of iterations in the program are expressed in form of recursion in the corresponding CRS. In addition, CRSs include information about how the size of variables changes when the control moves between different parts of the program (e.g., the increase of a loop index). CRSs are a generalization of recurrence equations in the sense that cost analysis often infers results which are CRSs but do not satisfy the syntactic requirements imposed by recurrence equations. CRSs can sometimes be *solved* by inferring a *closed form* solution (or an *upper bound* of it), i.e., an expression without recurrences, by using *Computer Algebra Systems* (CASs) such as Mathematica and Maple.

Ideally, we are interested in obtaining CRSs where only the program variables and arguments which affect the cost appear as arguments in the equations. The remaining ones can be ignored as far as cost analysis is concerned. The focus of this paper is on defining automatic techniques which can identify and remove from CRSs generated by cost analysis of Java bytecode [5], variables which do not affect the cost, and are therefore *useless*. Importantly, removing useless arguments from CRSs is crucial for the practical uptake of cost analysis for, at least, the following two reasons: (i) static (cost) analysis can be more efficient if we reduce the number of variables; and (ii) CRSs become simpler, and more likely solvable with standard CASs. Essentially, what we do in [8], is to classify useless variables into two categories:

1. *Redundant stack variables.* A *stack variable* represents an operand stack location at a given program point. Stack variables are a component of our rule-based representation of programs (Sec. 3.4.1, Step II). They are created from the original program by means of a transformation step which eliminates the stack and uses additional variables instead. It is often the case that a stack variable is created by loading a program variable (or constant) on the stack, and it is immediately eliminated after storing the result of some computation (as in the typical *load*, *load*, *add*, *store* sequence). In this case, in the rule-based representation, it is safe to replace stack variables by the local variables or constants whose value was loaded on the corresponding stack location in the program. This can be done by first applying a *single static assignment* transformation to the corresponding sequence of bytecode instructions, to avoid name clashes, and then *unifying* (i.e., making identical) the local variable or constants with the corresponding stack variable. E.g., in $\mathsf{iload}(\mathsf{v}, \mathsf{s_i})$ we unify $\mathsf{s_i}$ with v, and in $\mathsf{iconst}(1, \mathsf{s_i})$ we unify $\mathsf{s_i}$ with 1.

2. *Cost-Irrelevant program variables.* The program variables which may have an impact on the cost of a program are those that may affect directly or indirectly the conditional statements (i.e., they can affect the control flow of the program), and those that may affect the values which are used as input to operations which do not have a constant cost. Calls to methods are a typical example of non-constant operations w.r.t. cost. Also bytecode instructions can be non-constant, as in the case of array manipulation (see Sec. 3.4.1). We refer to both kinds of variables as *cost-relevant.* The rest of variables are called *cost-irrelevant*, and can be safely removed. For instance, typically, *accumulating* variables which merely keep the temporary value of some result do not affect the control flow, and, if they are not used in operations with a non-constant cost, they are cost-irrelevant. We formalize the problem of computing a safe approximation of the set of cost-relevant arguments as a *backward slicing* [104] problem, where variables in reachable conditional statements and operations with non-constant cost are the information we want to preserve, i.e., they are used to build the *slicing criterion.*

While redundant stack variables only occur in stack-based programming languages (such as Java bytecode), their removal is beneficial in principle for any static analysis, not only cost analysis. Also, though cost-irrelevant variable elimination is particular to cost analysis (and also termination) it is of interest for cost analysis of any language, not only Java bytecode. Therefore, we claim that our contributions are both useful for other analyses (category 1) and applicable to other programming languages and paradigms (category 2).

Cost Analysis of Java Bytecode by Example

We briefly illustrate the cost analysis we refer to [5] by using the example depicted in Fig. 3.6. The Java program (on the top, provided just for clarity since the analysis is directly performed on the Java bytecode

```
class Sum {
 static int sum(int m, int n) {
  int res=0;
  for (int i=1; i<=m; i++)
   for (int j=i; j<=n; j++)
    res += i*j;
  return res;
 }
}
```

```
 0:  iconst_0         18:  iload_2
 1:  istore_2         19:  iload_3
 2:  iconst_1         20:  iload 4
 3:  istore_3         22:  imul
 4:  iload_3          23:  iadd
 5:  iload_0          24:  istore_2
 6:  if_icmpgt 37     25:  iinc 4, 1
 9:  iload_3          28:  goto 12
10:  istore 4         31:  iinc 3, 1
12:  iload 4          34:  goto 4
14:  iload_1          37:  iload_2
15:  if_icmpgt 31     38:  ireturn
```

Figure 3.6: A Java Program and its corresponding Java bytecode

program) and its corresponding Java bytecode (on the bottom) define a method *sum* that, given the integer values n and m, computes the sum

$$res = \sum_{i=1}^{m} \sum_{j=i}^{n} i * j$$

Computing a *closed form* function which is an exact solution or an upper bound of the cost of *sum* (e.g., we may choose a cost model where the cost is the number of bytecode instructions which may be executed) in terms of its input arguments consists of several steps which are explained below: (1) recovering the structure of the program by means of a set of Control Flow Graphs (CFGs); (2) transforming the CFGs into a rule-based representation; (3) inferring *size relations* between the program variables and generating a CRS from which we can obtain a closed form solution or upper bound using standard CASs.

Step I: Control Flow Graph

In the first step, each sequence of Java bytecode instructions (which corresponds to a method) is transformed into a corresponding set of CFGs by using techniques from compiler theory [1, **?**]. This is done by splitting the instruction sequence into maximal sub-sequences of *non-branching* instructions, which form the basic blocks (nodes) of the initial graph. The basic blocks are connected by *guarded* edges which describe the possible transitions. Guards and edges are introduced by considering the last bytecode instruction of each block, and represent the condition (guard) for the control going from one block to another one. Finally, a *loop extraction* transformation is applied on the initial CFG in order to separate those sub-graphs corresponding to loops. This transformation has been well studied in the area of program decompilation [16]. It is crucial when the program contains *nested loops*, since it allows analyses which are *compositional*, in the sense that they can reason on just one loop at a time.

The CFGs of the *sum* method are depicted in Fig. 3.7. In block 0, the variables i and *res* are initialized (first four instructions), and the control is transferred to the middle CFG (using the instruction call_loop), which corresponds to the outer-loop. Upon return from that loop, the method returns the value *res* (last two instructions). In block 1 (the entry of the outer-loop), the values of i and m are compared. If $i \leq m$

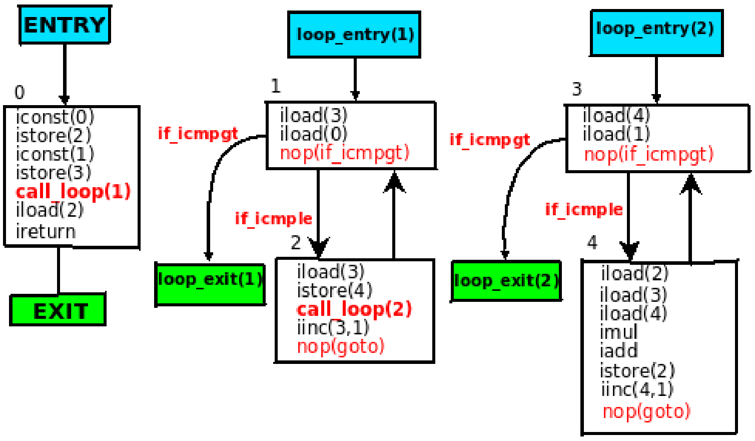


Figure 3.7: CFGs for the Java bytecode program in Fig. 3.6

then the control is transferred to block 2 which corresponds to the loop's body, otherwise the control is transferred to a block which indicates that the loop has terminated (and control goes back to the caller). Note that the corresponding edges are annotated by conditions (guards) corresponding to $i{\leq}m$ and $i{>}m$. Block 2 corresponds to the body of the outer-loop: it initializes j, transfers the control to block 3 which corresponds to the inner-loop, and upon return it increases i by one. The inner-loop is defined similarly by the CFG on the right.

Step II: Rule-Based Representation

In the second step, CFGs are represented procedurally by means of rule-based programs. A *rule-based program* defines a set of *procedures*, each of them defined by one or more rules. We use $\bar{x}$ to denote a sequence of variables $\langle x_1, \ldots, x_n\rangle$. Each rule takes the form $head(\bar{x}, \bar{y})$:=$[guard]$, $instr$,$[cont]$ where (1) *head* is a unique identifier for the procedure the rule belongs to; (2) $\bar{x}$ and $\bar{y}$ respectively indicate the sequences of input and output arguments; (3) *guard* takes the form $\mathsf{guard}(\phi)$, where ϕ is a Boolean condition on the variables in $\bar{x}$; (4) *instr* is a sequence of bytecode instructions (where all input and output arguments to the instructions, including the local variables and stack elements they work on, are made explicit) and calls to other rules; and (5) *cont* indicates a call to another procedure which represents the continuation of this procedure, if it exists. We use [_] to denote that an element is optional.

The most relevant issue in our study is related to the variables which become the arguments of the rules. Regarding the input arguments, $\bar{x}$ should include the local variables for the method, and the stack elements at the beginning of the block. As for the output arguments, $\bar{y}$ usually contains only the return value of the method, denoted by r. Moreover, in the case of rules which correspond to loops, output arguments also include the variables which are modified during the execution of the loop.

The rule-based program(s) depicted in Fig. 3.8 correspond respectively to the CFGs in Fig. 3.7. The rule *sum* corresponds to the method entry. It takes the input local variables m and n, and returns the output variable r. It first calls $\mathsf{init\_local\_vars}(\langle \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle)$ which initializes the local variables to the default value of their type as stipulated by Java, and then calls the rule sum_0 (corresponding to block 0). The rule sum_0 takes all local variables (including the formal parameters) as input and returns r as output. The instructions $\mathsf{iconst}(0, \mathsf{s}_0)$ and $\mathsf{istore}(\mathsf{s}_0, \mathsf{res})$ initialize res to zero (note that s_0 corresponds to a stack position which is explicit in the rule-based representation). Similarly, $\mathsf{iconst}(1, \mathsf{s}_0)$ and $\mathsf{istore}(\mathsf{s}_0, \mathsf{i})$ initialize i to one. Afterwards, the outer-loop is called using $\mathsf{sum}_1(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{res}, \ \mathsf{i}, \mathsf{j}\rangle)$, and, upon return, the last two instructions $\mathsf{iload}(\mathsf{res}, \mathsf{s}_0)$ and $\mathsf{ireturn}(\mathsf{s}_0, \mathsf{r})$ bind the output r to the return value of *sum*. Note that, when calling the outer-loop sum_1, the list of output arguments only includes those that might be modified during the execution of sum_1. The rule sum_1 (which corresponds to block 1) is the entry rule to the outer-loop. The fact that block 1 has two successors (block 2 and the loop-exit block) is expressed by a call to a

$$\boxed{\begin{array}{l}
\mathsf{sum}(\langle \mathsf{m},\mathsf{n}\rangle,\langle \mathsf{r}\rangle) := \mathsf{init\_local\_vars}(\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle),\mathsf{sum_0}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{r}\rangle).\\
\mathsf{sum_0}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{r}\rangle) := \mathsf{iconst}(0,\mathsf{s_0}),\mathsf{istore}(\mathsf{s_0},\mathsf{res}),\mathsf{iconst}(1,\mathsf{s_0}),\\
\quad \mathsf{istore}(\mathsf{s_0},\mathsf{i}),\mathsf{sum_1}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle),\mathsf{iload}(\mathsf{res},\mathsf{s_0}),\mathsf{ireturn}(\mathsf{s_0},\mathsf{r}).
\end{array}}$$

$$\boxed{\begin{array}{l}
\mathsf{sum_1}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{i},\mathsf{s_0}),\mathsf{iload}(\mathsf{m},\mathsf{s_1}),\\
\quad \mathsf{nop}(\mathsf{if\_icmpgt}(\mathsf{s_0},\mathsf{s_1})),\mathsf{sum_1^c}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s_0},\mathsf{s_1}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle).\\
\mathsf{sum_1^c}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s_0},\mathsf{s_1}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmple}(\mathsf{s_0},\mathsf{s_1})),\\
\quad \mathsf{sum_2}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle).\\
\mathsf{sum_1^c}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s_0},\mathsf{s_1}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmpgt}(\mathsf{s_0},\mathsf{s_1})).\\
\mathsf{sum_2}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{i},\mathsf{s_0}),\mathsf{istore}(\mathsf{s_0},\mathsf{j}),\\
\quad \mathsf{sum_3}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle),\mathsf{iinc}(\mathsf{i},1),\mathsf{nop}(\mathsf{goto}),\\
\quad \mathsf{sum_1}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle).
\end{array}}$$

$$\boxed{\begin{array}{l}
\mathsf{sum_3}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{j},\mathsf{s_0}),\mathsf{iload}(\mathsf{n},\mathsf{s_1}),\\
\quad \mathsf{nop}(\mathsf{if\_icmpgt}(\mathsf{s_0},\mathsf{s_1})),\mathsf{sum_3^c}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s_0},\mathsf{s_1}\rangle,\langle \mathsf{res},\mathsf{j}\rangle).\\
\mathsf{sum_3^c}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s_0},\mathsf{s_1}\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmple}(\mathsf{s_0},\mathsf{s_1})),\\
\quad \mathsf{sum_4}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle).\\
\mathsf{sum_3^c}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s_0},\mathsf{s_1}\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmpgt}(\mathsf{s_0},\mathsf{s_1})).\\
\mathsf{sum_4}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{res},\mathsf{s_0}),\mathsf{iload}(\mathsf{i},\mathsf{s_1}),\mathsf{iload}(\mathsf{j},\mathsf{s_2}),\\
\quad \mathsf{imul}(\mathsf{s_1},\mathsf{s_2},\mathsf{s_1}),\mathsf{iadd}(\mathsf{s_1},\mathsf{s_0},\mathsf{s_0}),\mathsf{istore}(\mathsf{s_0},\mathsf{res}),\mathsf{iinc}(\mathsf{j},1),\mathsf{nop}(\mathsf{goto}),\\
\quad \mathsf{sum_3}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle).
\end{array}}$$

Figure 3.8: The Intermediate Representation for the CFGs of Fig. 3.7

$$\begin{array}{rll}
(1) & \mathsf{sum}(\mathsf{m},\mathsf{n}) = \mathsf{sum_0}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\underline{\mathbf{j}}) & \{res=0,i=0,j=0\}\\
(2) & \mathsf{sum_0}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\underline{\mathbf{j}}) = 6+\mathsf{sum_1}(\mathsf{m},\mathsf{n},\underline{\mathbf{res'}},\mathsf{i'},\underline{\mathbf{j}}) & \{res'=0,i'=1\}\\
(3) & \mathsf{sum_1}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}}) = 3+\mathsf{sum_1^c}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & \{s_0=i,s_1=m\}\\
(4) & \mathsf{sum_1^c}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) = \mathsf{sum_2}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}}) & \{s_0 \leq s_1\}\\
(5) & \mathsf{sum_1^c}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) = 0 & \{s_0 > s_1\}\\
(6) & \mathsf{sum_2}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}}) = 4+\mathsf{sum_3}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j'})+\mathsf{sum_1}(\mathsf{m},\mathsf{n},\underline{\mathbf{res'}},\mathsf{i'},\underline{\mathbf{j''}}) & \{j'=i,i'=i+1\}\\
(7) & \mathsf{sum_3}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}) = 3+\mathsf{sum_3^c}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & \{s_0=j,s_1=n\}\\
(8) & \mathsf{sum_3^c}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) = \mathsf{sum_4}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}) & \{s_0 \leq s_1\}\\
(9) & \mathsf{sum_3^c}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) = 0 & \{s_0 > s_1\}\\
(10) & \mathsf{sum_4}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}) = 9+\mathsf{sum_3}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j'}) & \{j'=j+1\}
\end{array}$$

Figure 3.9: CRS for the rule-based program of Fig. 3.8

continuation rule sum_1^c (at the end of sum_1), which in turn is defined by two rules. The first one accounts for to the case where $i \leq m$ ($\mathsf{guard}(\mathsf{if\_icmple}(\mathsf{s_0},\mathsf{s_1}))$), which continues to sum_2. The second one accounts for $i > m$ ($\mathsf{guard}(\mathsf{if\_icmpgt}(\mathsf{s_0},\mathsf{s_1}))$), which terminates the loop.

Instructions labeled with nop are not considered when computing the size relations in the following step, since they have been replaced either by guards or by calls to some other rule. For instance, nop(if_icmpgt) in *sum₁* is replaced by the corresponding guards in sum_1^c. Similarly, nop(goto) in sum_2 is replaced by the call to sum_1.

Step III: Generating a Cost Equations System

In the last step, size relations analysis is applied to the rule-based program and a CRS, which defines the cost of each rule as a function of its input arguments, is generated. The aim of the size analysis is to infer (linear) relations between the values (or sizes of data structures) of the different variables at different program points. For example, it infers that the value of i when calling sum_1 (in the rule sum_2) is greater than the input value of i by one. Using these size relations, for each rule in the corresponding rule-based program we generate an equation of the form

```
$\mathsf{sum_0}(\langle m, n, res, i, j\rangle, \langle r\rangle)$ :=
  $\mathsf{iconst}(0, s'_0)$,                                     $\%\rho_2{=}\rho_1[s_0{\mapsto}s'_0]$
  $\mathsf{istore}(s'_0, res')$,                                  $\%\rho_3{=}\rho_2[res{\mapsto}res']$
  $\mathsf{iconst}(1, s''_0)$,                                    $\%\rho_4{=}\rho_3[s_0{\mapsto}s''_0]$
  $\mathsf{istore}(s''_0, i')$,                                   $\%\rho_5{=}\rho_4[i{\mapsto}i']$
  $\mathsf{sum_1}(\langle m, n, res', i', j\rangle, \langle res'', i'', j'\rangle)$,  $\%\rho_6{=}\rho_5[\langle res, i, j\rangle{\mapsto}\langle res'', i'', j'\rangle]$
  $\mathsf{iload}(res'', s'''_0)$,                                $\%\rho_7{=}\rho_6[s_0{\mapsto}s'''_0]$
  $\mathsf{ireturn}(s'''_0, r)$.                                  $\%\rho_8{=}\rho_7$
```

```
$\mathsf{sum_1}(\langle m, n, res, i, j\rangle, \langle res', i', j'\rangle)$ :=
  $\mathsf{iload}(i, s'_0)$,                                      $\%\rho_2{=}\rho_1[s_0{\mapsto}s'_0]$
  $\mathsf{iload}(m, s'_1)$,                                      $\%\rho_3{=}\rho_2[s_1{\mapsto}s'_1]$
  $\mathsf{nop}(\mathsf{if\_icmpgt}(s'_0, s'_1))$,                $\%\rho_4{=}\rho_3$
  $\mathsf{sum^c_1}(\langle m, n, res, i, j, s'_0, s'_1\rangle, \langle res', i', j'\rangle)$.  $\%\rho_5{=}\rho_4[\langle res, i, j\rangle$
                                                                  ${\mapsto}\langle res', i', j'\rangle]$
```

Figure 3.10: The SSA transformations for some rules in Fig. 3.8

$$p(\bar{x}) = c + \sum_{i=1}^{k} p_i(\bar{x}_i), \quad \varphi$$

which defines the cost of the rule p in terms of its input arguments $\bar{x}$ to be: (1) c is the direct cost of the bytecode instructions which appear in the rule; plus (2) the cost of all calls to other rules, namely $p_1(\bar{x}_1)\ldots, p_k(\bar{x}_k)$. The linear constraints φ (which are inferred by the size analysis) describe the size relations between the variables $\bar{x} \cup \bar{x}_1 \cdots \cup \bar{x}_k$. We refer to the set of all generated equations as CRS.

Assuming that we are interested in knowing the number of bytecode instructions executed, from the rule-based program in Fig. 3.8 we obtain the CRS depicted in Fig. 3.9. Equation 1 corresponds to the entry cost equation and its size relations reflect the initialization of the local variables (init_local_vars in the recursive representation). Let us consider now, for example, the equations 3-6 which correspond to the outer-loop. Equation 3 defines the cost of $\mathsf{sum_1}(m, n, res, i, j)$ to be the number of its bytecode instructions, namely 3, plus the cost of executing $\mathsf{sum^c_1}(m, n, res, i, j, s_0, s_1)$ where $s_0{=}i$ and $s_1{=}m$. Equations 4 and 5 define the cost of $\mathsf{sum^c_1}(m, n, res, i, j, s_0, s_1)$ to be equal to the cost of $\mathsf{sum_2}(m, n, res, i, j)$ if $s_0{\le}s_1$, and 0 if $s_0{>}s_1$. Equation 6 defines the cost of $\mathsf{sum_2}(m, n, res, i, j)$ to be the number of its bytecode instructions, namely 4, plus the cost of $\mathsf{sum_3}(m, n, res, i, j')$ (the inner-loop) and $\mathsf{sum_1}(m, n, res', i', j'')$ where $j'{=}i$ (the initial value for j in the inner-loop) and $i'{=}i{+}1$ (the outer-loop counter is increased).

Automatic cost analysis usually aims at providing a closed form upper bound from the CRS, e.g., $sum(m, n){=}O(m * n)$. Sometimes, this can be done by using standard CASs, such as Mathematica and Maple. A problem that most automatic cost analyzers face is that, without a further processing, the generated CRS are not even considered as a valid input for such systems. This is often the case when equations in the CRS contain *irrelevant* variables which do not affect the cost of the corresponding rules. As a consequence, human interaction is usually required to remove them before giving the CRS to the CAS. The problem can be often alleviated by further simplifying the equations by automatically removing such irrelevant variables. As we will explain in the next sections, in the CRS of Fig. 3.9, the underlined variables are irrelevant to the cost and therefore can be safely eliminated. Furthermore, all *stack* variables, which appear in frames, are redundant and can be replaced by the corresponding local variables. In the next two sections, we provide automatic techniques to identify and eliminate both classes of irrelevant variables in the context of cost analysis of Java bytecode.

Removing Redundant Stack Variables

Stack variables can often be removed from the rule-base representation of a program by replacing their occurrences with local variables or constants. Our method to remove redundant stack variables is based on transforming the rule to be in *static single assignment* form and then *unifying* stack variables to local variables (or constants) in the statements that relate them. This is a simple process that is done locally to the rules. We sketch the three steps which are performed by our system in each corresponding subsection.

Static Single Assignment

In the first step, we perform a Static Single Assignment (SSA) transformation on the bytecode instructions of the rule-based representation. This is necessary since we want to apply unification to its variables (and they cannot be assigned to more than one value). It should be noted that this step is also needed –regardless whether we perform useless variable elimination– for the size analysis because it infers input-output denotations for each program point. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ will be transformed to $\mathsf{iadd}(s_0, s_1, s'_0)$ where s'_0 refers to the value of s_0 after executing the instruction. To implement the SSA transformation, we maintain a mapping ρ, for each rule, 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 write $\rho[\bar{x} \mapsto \bar{y}]$ to denote the mapping of each element in $\bar{x}$ to a corresponding one in $\bar{y}$.

For each rule $p(\bar{x}, \bar{y}){:=}b_1, \ldots, b_n$, the SSA transformation generates a new rule $p(\bar{x}, \rho_{n+1}(\bar{y})){:=}b'_1, \ldots, b'_n$ by translating each b_i to b'_i as illustrated in the following table for a few bytecode instructions.

i−*th element*	*SSA*	ρ_{i+1}
$\mathsf{iload}(v, s_j)$	$\mathsf{iload}(\rho_i(v), s')$	$\rho_i[s_j \mapsto s']$
$\mathsf{istore}(s_j, v)$	$\mathsf{istore}(\rho_i(s_j), v')$	$\rho_i[v \mapsto v']$
$\mathsf{iadd}(s_j, s_{j+1}, s_j)$	$\mathsf{iadd}(\rho_i(s_j), \rho_i(s_{j+1}), s')$	$\rho_i[s_j \mapsto s']$
$q(\overline{x}, \overline{y})$	$q(\rho_i(\overline{x}), \overline{y}')$	$\rho_i[\overline{y} \mapsto \overline{y}']$
$\mathsf{guard}(\phi)$	$\mathsf{guard}(\rho_i(\phi))$	ρ_i

where (1) ρ_1 is the identity renaming; (2) ρ_i $(2{\le}i{\le}n+1)$ is the mapping available before traversing b_i; (3) ρ_{i+1} is the result of updating ρ_i; and (4) $\overline{y}'$, s' and v' are fresh variables. As an example, we show in Fig. 3.10 the SSA transformation for rules *sum₀* and *sum₁* together with the associated renaming for each bytecode.

Propagating Dependencies by Unification

After applying the SSA transformation, we can *unify* the stack elements, local variables and constants that occur as arguments of instructions like iload, iconst, istore and ireturn. These unifications will automatically reduce the number of (distinct) variables which occur in the rule. In addition, the corresponding instruction can be removed from the program as its effect is already accounted for in the unifications. This can be implemented using standard unification as, for instance, done in logic programming [67]. In our example, the rules sum_0 and sum_1 in Fig. 3.10 allow us to apply respectively the following sets of unifications:

$$\begin{array}{ll} sum_0: & \{0{=}s'_0, s'_0{=}res', 1{=}s''_0, s''_0{=}i, res''{=}s'''_0, s'''_0{=}r\} \\ sum_1: & \{i{=}s'_0, m{=}s'_1\} \end{array}$$

Removing the corresponding instructions[3] (after applying the unifications) results in the following simplified rule:

$$\begin{array}{lcl} \mathsf{sum_0}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{r}\rangle) & := & \mathsf{sum_1}(\langle \mathsf{m}, \mathsf{n}, 0, 1, \mathsf{j}\rangle, \langle \mathsf{r}, \mathsf{i}'', \mathsf{j}'\rangle). \\ \mathsf{sum_1}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}'\rangle) & := & \mathsf{sum_1^c}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}, \mathsf{i}, \mathsf{m}\rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}'\rangle). \end{array}$$

A main advantage of implementing the dependency tracking by relying on unification is that we implicitly have *backwards* propagation of dependencies.

Argument Filtering

In addition to removing the bytecode instructions that manipulate stack elements, after unifying stack variables to local variables and constants, we can often filter out some arguments from rules. In our example for the rule sum_1, after the unification of s'_0 and s'_1 to the local variables, respectively, i and m as described

[3] Even if these instructions have no effect in size analysis, depending on the cost model they have to be taken into account to generate the cost relation system.

in Section 3.4.1, calls to the rule sum_1^c become of the form $\mathsf{sum}_1^{\mathsf{c}}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}, \underline{\mathsf{i}}, \underline{\mathsf{m}} \rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}' \rangle)$. We can observe that the same local variables i and m are unnecessarily passed twice in all calls to sum_1^c. Therefore, in the last step, we remove repeated arguments and constant arguments from rules as long as *all* calls to such rules are performed with the same repeated variables. Therefore, in our example, without the irrelevant stack variables, the head of the rule becomes $\mathsf{sum}_1^{\mathsf{c}}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j} \rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}' \rangle)$. Similarly, we can reason that the head of the rule sum_3^c can be $\mathsf{sum}_3^{\mathsf{c}}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j} \rangle, \langle \mathsf{res}', \mathsf{j}' \rangle)$.

It could happen that there exists one (or several) call which does not have the same repeated arguments. In such case, the argument filtering process described above cannot be performed. It would be possible, though, to filter those arguments which are actually repeated in all possible call patterns to the corresponding rule. In practice, by using this simple analysis described in the three phases above, all stack variables can often be eliminated, except some of them that correspond to the return value of methods. Clearly, eliminating the stack variables from the rule-based representation can improve the performance of any analysis of Java bytecode, as the number of total variables is significantly reduced. In addition, it can also be beneficial for obtaining a clearer source code in decompilation of bytecode.

Eliminating Cost-Irrelevant Program Variables

This section describes a technique, based on program slicing and dependence calculus, whose purpose is to remove information which is not needed by cost analysis. Program slicing [106, 104] has been usually applied to source code, since its main interest is to help humans in debugging, maintaining and understanding software. This is also the case of Java [17], where little or no attention has been paid to slicing bytecode presumably for this reason. In this sense, our slicing algorithm is the first one developed at the level of bytecode. However, our approach does not properly belong to standard slicing, since the focus is on removing variables instead of statements. Also, the executability of the slice is not an issue here.

The process of eliminating variables which are irrelevant for computing the cost is based on the basic observation that the cost is affected by (1) variables appearing in *guards*, since those are the variables corresponding to loop conditions, recursion base-case conditions, etc.; and (2) arguments of bytecode instructions which are required by the cost model in order to compute its corresponding cost, e.g., when creating an array the size of the array is important if we count memory consumption. Therefore, variables which are guaranteed not to affect, directly or indirectly, any variable of the above two categories can be safely removed from the analysis. This can be done by computing a superset (i.e., a safe approximation) of the set of the variables which might affect variables of the above two categories. Thus, the problem can be formalized as a (static) *backward slicing* problem, where the slicing criterion is all variables of the above two categories. Due to this choice of criterion, the slicing algorithm does not need to track *implicit dependencies* (e.g., an assignment in the branches of a conditional, where an implicit dependence exists between the guard variables and the modified variables), since all variables in guards will be included in the relevant set of variables. Standard backward slicing algorithms [104] can be adapted and applied directly to the rule-based representation. The algorithm in Figure 3.11 is a slicing algorithm that computes a set of relevant input variables for each rule p. The algorithm returns for each rule p a set $\mathsf{slice}(p)$ of relevant input arguments. It has a fixpoint nature with an abstract interpretation flavor [38], where at each iteration the set $\mathsf{slice}(p)$ is refined to include more variables. As we discuss below, our algorithm assumes information flow [44] and sharing information [97] is precomputed and available at slicing time. The procedure slicing is the fixpoint driver, and the procedure slice_procedure is the one applied iteratively by the driver until a fixpoint is reached. We start by explaining slice_procedure.

Assume, for a given rule $p(\bar{x}, \bar{y}) := G, b_1, \ldots, b_n$, that we already know that a subset of the output variables $\mathcal{V} \subseteq \bar{y}$ is required for the cost of some other rules, for example, it can be because these output variables affect guards of other rules. The aim is to compute a subset of the input variables $\bar{x}$ that might affect (1) the given subset of output variables $\mathcal{V}$; (2) the guard of p; (3) variables that are relevant to the cost of other rules that are called from p; and (4) arguments of bytecodes that are required by the cost model. We do this by going backwards from b_n to b_1 (lines **??**-**??**) such that at each step the set $\mathcal{V}$ is refined by using some dependency information that we obtain from the corresponding b_i. We distinguish two cases

```
1 :  Procedure slicing
2 :    for all procedure $p$ defined in $Program$
3 :      slice($p$) = $\emptyset$      // initialization
4 :      last_invoke($p$) = $\emptyset$
5 :      add_to_queue($\langle p, \emptyset \rangle$, $\mathcal{Q}$)      // initial calls
6 :    while $\langle p, \mathcal{P} \rangle$ = get_from_queue($\mathcal{Q}$)   // non-empty queue
7 :      slice_procedure(p,$\mathcal{P}$)

8 :  Procedure slice_procedure($p$,$\mathcal{P}$)
9 :    for all  $p(\bar{x}, \bar{y}) := G, b_1, \ldots, b_n \in Program$
10 :     $\mathcal{V} =_{sh}$ pos_to_vars($\mathcal{P}, \bar{y}$)   // converting from positions
11 :     for i=n to 1
12 :       if $b_i$ takes the form $q(\bar{x}_i, \bar{y}_i)$ then   // proc. call
13 :         $\mathcal{V}' = \{w \mid w \in \bar{x}_i, z \in \mathcal{V} \cap \bar{y}_i, w\text{might affect}z\}$
14 :         $\mathcal{V} =_{sh} (\mathcal{V} \backslash \bar{y}_i) \cup$pos_to_vars(slice($q$)$, \bar{x}_i$)$\cup \mathcal{V}'$
15 :         $\mathcal{P}_0$ = vars_to_pos($\mathcal{V} \cap \bar{y}_i, \bar{y}_i$)
16 :         if $\mathcal{P}_0 \not\subseteq$ last_invoke($q$) then   // re-analyze q
17 :           last_invoke($q$) = last_invoke($q$) $\cup \mathcal{P}_0$
18 :           add_to_queue($\langle q$, last_invoke($q$)$\rangle$, $\mathcal{Q}$)
19 :       else    // $b_i$ is a bytecode
20 :         if  $\mathcal{V} \cap$output_vars($b_i$) $\neq \emptyset$ then
21 :           $\mathcal{V}'$ = input_vars($b_i$)
22 :         else
23 :           $\mathcal{V}' = \emptyset$
24 :         $\mathcal{V} =_{sh} (\mathcal{V} \backslash$output_vars($b_i$)$) \cup \mathcal{V}' \cup$cm_vars($b_i$)
25 :     $\mathcal{V} =_{sh} \mathcal{V} \cup vars(G)$
26 :     $\mathcal{P}$ = vars_to_pos($\mathcal{V} \cap \bar{x}, \bar{x}$)
27 :     if $\mathcal{P} \not\subseteq$ slice($p$) then
28 :       slice($p$) = slice($p$) $\cup \mathcal{P}$
29 :       forall $q$ which call $p$
30 :         add_to_queue($\langle q$, last_invoke($q$)$\rangle$, $\mathcal{Q}$)
```

Figure 3.11: A naïve algorithm for backward slicing of the rule-based representation

depending on whether b_i is a bytecode or a call to another rule:

- If $b_i = q(\bar{x}_i, \bar{y}_i)$ (line 12) then we first compute a set of variables $\mathcal{V}' \subseteq \bar{x}_i$ (line 13) that might affect the value of any variable in $\mathcal{V} \cap \bar{y}_i$ (we assume this information is available using information flow analysis [44]). Then (line 14) we refine $\mathcal{V}$ by removing all variables that are in $\mathcal{V} \cap \bar{y}_i$ and adding the relevant variables for q (namely slice(q)) and $\mathcal{V}'$. Afterwards (lines 15-18), q is scheduled for re-analyses if the set of output variables of interest $\mathcal{V} \cap \bar{y}_i$ is not included in the one with respect to which it was analyzed the last time. Note that the role of pos_to_vars and vars_to_pos is to convert from variable positions to their names and vice versa since it is sometimes convenient to use the names and sometimes the positions.

- If b_i is a bytecode (line 19) then if any of the output variables of b_i is included in $\mathcal{V}$ we will include its set of input variables in $\mathcal{V}$, this is reflected by computing $\mathcal{V}'$ in lines 20-23. Then $\mathcal{V}$ is refined (line 24) by removing the output variables of b_i and adding $\mathcal{V}'$ and those which are required by the cost model, namely cm_vars(b_i).

Once the backwards propagation is done, we add the guard variables to $\mathcal{V}$ (line 25) and project $\mathcal{V}$ on the head variables (line 26). At the end, if the computed set of relevant arguments is not included in the previous

$$
\begin{array}{rlcll}
(1) & \mathsf{sum(m,n)} & = & \mathsf{sum_0(m,n)} & \{\} \\
(2) & \mathsf{sum_0(m,n)} & = & 6+\mathsf{sum_1(m,n,i')} & \{i'{=}1\} \\
(3) & \mathsf{sum_1(m,n,i)} & = & 3+\mathsf{sum_1^c(m,n,i)} & \{\} \\
(4) & \mathsf{sum_1^c(m,n,i)} & = & \mathsf{sum_2(m,n,i)} & \{i{\leq}m\} \\
(5) & \mathsf{sum_1^c(m,n,i)} & = & 0 & \{i{>}m\} \\
(6) & \mathsf{sum_2(m,n,i)} & = & 4+\mathsf{sum_3(n,j')}+\mathsf{sum_1(m,n,i')} & \\
 & & & \{j'=i, i'=i+1\} & \\
(7) & \mathsf{sum_3(n,j)} & = & 3+\mathsf{sum_3^c(n,j)} & \{\} \\
(8) & \mathsf{sum_3^c(n,j)} & = & \mathsf{sum_4(n,j)} & \{j{\leq}n\} \\
(9) & \mathsf{sum_3^c(n,j)} & = & 0 & \{j{>}n\} \\
(10) & \mathsf{sum_4(n,j)} & = & 9+\mathsf{sum_3(n,j')} & \{j'{=}j{+}1\}
\end{array}
$$

Figure 3.12: A Simplified version of the CRS of Fig. 3.9

one (line 27), then the new set is stored (line 28), and each rule q that calls p is scheduled for re-analyses (lines 29-30). Note in particular the use of $=_{sh}$ which means that the computed set is closed under sharing information, i.e., if a variable v is in the set, then all variables that might share with v a data structure on the heap are also included in the set. This information can be computed using sharing analysis [97]. The main procedure slicing performs the fixpoint by means of a queue $\mathcal{Q}$. It initializes the values of each slice(p) to empty set, and schedules each p to be analyzed with respect to an empty set of output variables.

As an example, applying backward slicing on the rule-based program of Fig.3.8 (after eliminating the stack variables) results in the following set of relevant variables for the different rules:

sum $=\{m,n\}$	$\mathsf{sum_1}=\{m,n,i\}$	$\mathsf{sum_3}=\{n,j\}$
$\mathsf{sum_0}=\{m,n\}$	$\mathsf{sum_1^c}=\{m,n,i\}$	$\mathsf{sum_3^c}=\{n,j\}$
	$\mathsf{sum_2}=\{m,n,i\}$	$\mathsf{sum_4}=\{n,j\}$

which can be used to simplify the CRS in Fig. 3.9 to the one in Fig. 3.12.

Note that an interesting feature of our slicing problem, is that any set of relevant variables can be safely used to refine the CRS, even if it is not a superset (i.e., safe approximation) of the actual relevant variables. This is due to the fact that removing arguments from the CRS can only increase the cost, and therefore we are approximating the CRS in the safe direction since we are interested in computing upper bounds. Indeed, in our current implementation we ignore the information flow (line 13) and the sharing information in $=_{sh}$ which in turn might result in unsafe approximation of the relevant variables, but in practice we still get very useful sets of relevant variables.

Experiments

We have incorporated the analysis techniques for the elimination of useless stack and program variables as described in the paper in COSTA.

Table 3.1 shows the effect of eliminating redundant stack and irrelevant local variables on a series of benchmarks for which our system can infer automatically CRSs. We include classical recursive programs such as *Factorial*, *Hanoi*, *Fibonacci*, *MergeSort* or *QuickSort*. 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* or *Incr*, which contain object creation, virtual invocation, etc. The remaining benchmarks are implemented using data structures: *ArrayReverse*, *MatMultVector*, *Search* use arrays and *BubbleSort* and *DoSum* traverse linked lists.

Column #**R** shows the number of rules in the rule-based representation of each program. Note that, for the case of our running example *Sum*, column #**V** contains 15 instead of 10, as shown in Fig. 3.8. This is because in Fig. 3.8 we have omitted some calls to continuations in order to simplify the presentation.

Benchmark	#R	#V	SVE	Slic	Bth	Rt
Polynomial	19	89	61	64	41	2.17
BinarySearch	15	115	78	83	52	2.21
DivByTwo	12	43	31	22	15	2.87
Indexes	29	232	169	155	105	2.21
EvenDigits	19	93	65	52	34	2.74
Factorial	11	31	21	20	13	2.38
ArrayReverse	12	65	46	31	17	3.82
Concat	15	108	71	64	34	3.18
Incr	40	177	136	104	75	2.36
ListReverse	12	63	46	32	20	3.15
Power	11	38	27	20	13	2.92
Search	26	137	105	73	52	2.63
MergeSort	26	212	156	134	87	2.44
QuickSort	26	226	166	159	106	2.13
Sum	15	101	75	55	37	2.73
ListInter	38	247	182	164	113	2.19
SelectSort	18	135	101	90	60	2.25
OrdSort	18	109	79	69	46	2.37
DoSum	29	155	111	87	57	2.72
BubbleSort	18	143	108	95	64	2.23
MatMult	18	191	144	94	56	3.41
Hanoi	12	49	35	13	7	7.00
Fibonacci	11	37	23	26	15	2.47

Table 3.1: Effect of removing useless variables

Column **#V** shows the total number of different variables (including both stack and local variables) in the rule-based representation after applying SSA. We show this figure because, for efficiency, our prototype always generates the rule-based representation with SSA, since SSA is required in order to perform size analysis. The next three columns, namely **SVE**, **Slic**, **Bth**, show the number of remaining variables after performing different removal techniques. More precisely, in column **SVE**, we have performed stack variable elimination but not removal of cost-irrelevant variables (slicing). In the column **Slic**, slicing is done but not stack variable elimination. It should be noted that slicing can sometimes remove redundant stack variables, which explains the significantly high variable removal in this case as well. Finally, **Bth** combines the result of applying simultaneously both techniques. Clearly, the effect of applying both techniques is not, in terms of the number of removed variables, the sum of the two analyses, since some variables can be removed by both **SVE** and **Slic**. The last column **Rt** shows **#V / Bth**. It can be observed that the number of variables is importantly reduced. The overall reduction ranges from 2.13 in the case of *Quicksort*, to 7 in the case of *Hanoi*. This is explained in part by the fact that local variables are always pushed on the stack by means of a load instruction such that the corresponding stack variable can be removed by unifying it with the local variable, as explained in Sec. 3.4.1.

3.4.2 Improving the accuracy of the analyzer

Termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Termination analysis is about the study of *loops*, which are the program constructs which may introduce non-termination. Loops may correspond to iterative constructs or to recursion. The boolean conditions which determine whether the loop should be executed again or not are called *guards*. Automated techniques for proving termination 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. In particular, analysis should keep track of how the (size of the) data involved in loop guards changes when the loop goes through its iterations. This information is used for specifying a *ranking function* for the loop [86], which is a function which strictly decreases on a well-founded domain at each iteration of the loop, thus guaranteeing that the loop will be executed a finite number of times.

In the last two decades, a variety of sophisticated termination analysis tools have been developed. Several analyses and tools exist, primarily for less-widely used programming languages, including term rewrite systems [46], and logic and functional languages [64, 36, 61]. Termination-proving techniques are also emerging in the imperative paradigm [30, 37, 46], even for dealing with large industrial code [37].

Termination analysis of realistic object-oriented programming languages faces new difficulties due to the existence of advanced features such as exceptions, virtual method invocation, references, heap-allocated data-structures, objects, fields. Focusing on Java, termination analyzers for Java bytecode programs [15] and for Java source [55] are being developed which are able to accurately handle a good number of the features mentioned above. However, interesting open problems still remain. In particular, it is well known that the *heap* poses important difficulties to static analysis. Some reasons for this are that the heap is a global data structure whose contents are not accessed using named variables, but rather using (possibly chained) references. Therefore, the same location in the heap may be modified using different aliased references and, furthermore, references may be reassigned several times, and thus they may point to different locations during execution. When loop guards involve information stored in the heap, such as object *fields*, tracking size information becomes rather complex and accurate aliasing information is required in order to track all possible updates of the corresponding fields (see e.g. [69]).

A partial solution to this problem is already solved by the *path-length* domain [55] which allows proving termination of loops which traverse acyclic heap-allocated data structures (i.e., linked lists, trees, etc.). Path-length is an abstract domain which, for reference values, provides a safe approximation of the length of the longest reference chain reachable from it. Unfortunately, though the path-length domain is a useful abstraction for fields which contain references, it does not capture any information about fields which contain numbers. In [3] we look into the Sun implementation of the Java libraries for J2SE 1.4.2 in order to estimate how often loop termination depends on *numeric* values stored in fields and to try to come up with sufficient conditions for termination which are able to cover a large fraction of those loops whose termination is not provable using current techniques, such as those in [55, 15].

Motivating Examples from the Java Libraries

Since termination is an undecidable problem, all techniques for proving termination provide sufficient (but not necessary) conditions for termination. Therefore, for any termination proving technique it is possible to find terminating programs where the given technique fails to prove termination. Thus, usually the practicality of termination analyses is measured by applying the analyses to a representative set of real programs. In this work, the design of the analysis is driven by common programming patterns for loops that we have found in the Java libraries. By looking at Sun's implementation of the J2SE (version 1.4.2_13) libraries, which contain 71432 methods, we have found 7886 loops (for, while, and do) from which 1021 (12.95%) explicitly involve fields in their guards. By inspecting these 1021 loops, we have observed, among others, the following three kinds of common patterns in the Java libraries.

Pattern #1: Loops in this category use numeric fields as bounds for loop counters and, moreover, the value of those fields is not updated within the loop. This is demonstrated in the following loop of the method public void or(BitSet set) of library java.util.BitSet, where unitsInUse is a field of type int:

```
for(; i<set.unitsInUse; i++) bits[i]=set.bits[i];
```

Pattern #2: Loops in this category are similar to those in the previous category. The difference is that, rather than corresponding to the value of a numeric field, the bound of the loop counter corresponds to the length of an array which is stored in a field. In this case, even if the elements of the array may be updated within the loop, if the field itself does not, the length of the array remains constant. This is demonstrated in the following example, corresponding to method public void fixupVariables(java.util.Vector vars, int globalsSize) of library org. apache.xpath.functions.FunctionMultiArgs where m_args is a field of type Expression[ ]:

```
for(int i=0; i<m_args.length; i++) m_args[i].fixupVariables(vars,globalsSize);
```

Pattern #3: Loops in this category use numeric fields as loop counters, which means that the field value is updated within the loop, but none of the references in the *path* to the field (in this example, the chain just consists of the reference this) are re-assigned within the loop, i.e., all updates correspond to the same object on the heap. This is demonstrated in the following loop of the method public synchronized void setLength(int newLength) in the library java.lang.StringBuffer, in which count is a field of type int:

```
for(; count<newLength; count++) value[count] = '\0';
```

In this paper we concentrate on proving termination of loops that fall in the above categories by providing (uniform) conditions under which proving termination of such loops becomes possible. The Java libraries include also other patterns such as loops that: (1) increase/decrease an integer variable until it reaches a given upper/lower bound; (2) traverse a non-cyclical data structure or an array; (3) look for an element in an input stream, which is common in classes that manipulate structured text such as parsing XML documents; and (4) look for a non-null element in a given array in a circular way, which is very common in the multi-threading classes. The first two patterns are the major part of the loops, and they are already handled in [15]. The other patterns are planned for future research and are not addressed in this paper.

Dealing with Fields in Termination

In a Java-like language, objects are stored in the *heap* and they are accessed by means of references (or pointers). References can take the value null or *point* to an object in the heap. Given a reference l which points to an object o, $l.f$ denotes the value of the field f in the object o. We say that a syntactic construction of the form $l.f$ is a *field access*. Each field f has a unique signature, which consists of the class where it is declared, its type, and its name.

Objects are global in that they survive the execution of methods. Typically, when a method starts execution, a large number of objects may exist in the heap. One approach to analyzing programs with objects is to compute an abstraction of the heap (see [77]) which approximates the execution context of each method. This usually requires computing abstractions of all possible objects in the program, which might turn out to be too expensive in practice if one wants to deal with real programs. However, in most cases, only a small fraction of such objects affects the execution of the method. We seek for a more lightweight approach which tries to approximate the contents of only a subset of the objects in the heap. The approach must remain correct by making safe assumptions about the objects (and fields) whose contents are not taken into consideration.

Another disadvantage of computing an abstraction of the heap, in addition to its computational complexity, is that we end up obtaining termination information which is *context-dependent*. Though context dependent analysis is in principle more precise, the results obtained are not extrapolable to other execution contexts. In particular, in the case of libraries, ideally we would like to prove termination in a *context-independent* way, i.e., regardless of what the contents of the heap are when the method is executed.

We now introduce the concept of *local* field access. In particular, we are interested in finding field accesses which are local to a *loop*. Though termination analysis in our context aims at proving termination of methods, in the rest of the paper we will concentrate on *loops* since they are the main subject of termination analysis.

Definition 3.4.1 (local field access) *We say that a field access $l.r_1.\ldots.r_n.f$, where f is a numeric field, is* local *to a loop L if*

(i) No prefix of $l.r_1.\ldots.r_n$ changes its value within L, i.e., they remain constant.

(ii) If the value of $l.r_1.\ldots.r_n.f$ changes within L, then all write accesses have to be done explicitly through the field access $l.r_1.\ldots.r_n.f$.

Condition (i) guarantees that all occurrences of the field access within the loop refer to the same memory location in the heap. Note that the prefixes of $l.r_1.\ldots.r_n$, i.e., l, $l.r_1$, $l.r_1.r_2$, $\ldots$ are references which altogether form a chain to an object where the numeric field f is stored. Condition (ii) guarantees that all write accesses to the field can be syntactically identified. Note that this condition can be violated due to aliasing, since we can have different field access which update the same memory location.

Given a loop L, we denote by *g-fields(L)* the set of field accesses $l.r_1 \ldots r_n.f$, where f is a numeric field, which explicitly appear inside the guard of L. For instance, for the three loops in Section 3.4.2, the sets *g-fields(L)* are, respectively, {this.unitsInUse}, {m_args.length} and {this.count}. These three fields are locally accessed within their corresponding loops. The practical implication is: if we ensure that a field in *g-fields*(L) is local, then we are able to treat this field in the same way as if it were a local variable, as regards the analysis of L. Essentially, given a loop L, the analysis proceeds as follows:

1. Compute the set *g-fields*(L).
2. Compute the set *l-g-fields*(L), which is the subset of *g-fields*(L) which contains the field accesses whose locality condition has been proved.
3. Analyze the termination of L by considering those field accesses in *l-g-fields*(L) as if they were local variables.

The method is applied locally to all nested loops in L. Note that the termination of a method is ensured if all loops *involved* in its body are terminating. By involved we mean not only those loops occurring explicitly in the body but also those coming from possible calls to some other methods.

Syntactic Inference of the Locality Condition on Field Accesses

The above approach is practical only if we provide effective mechanisms to prove the locality condition on field accesses. In this section, we consider only loops that do not contain method invocations. Later, in Section 3.4.3, we take method invocations into account. Now, we present sufficient *syntactic* conditions for ensuring that a field access is local. The following conditions ensure that a numeric field access $l.r_1.\ldots.r_n.f$ is local to a loop L:

1. The reference variable l remains constant in L. This can be ensured by checking that there is no assignment to l within L.
2. All reference fields $l.r_1, \ldots, l.r_1...r_n$ are constant in L. This can be ensured by checking that there is no assignment within L to a field with the same signature as any of r_i.
3. All assignments to a field with the same signature as f in L are done through the field access $l.r_1.\ldots.r_n.f$.

Let us briefly explain each of the above conditions. Conditions 1 and 2 ensure point (i) of Definition 3.4.1. The reason why we separate it into two conditions is due to the way in which it is syntactically checked in each case. For the reference variable l, we check that there is no assignment to it. These conditions guarantee that we do not incorrectly consider a loop of the form *while (l.size < 10)* {*l.size++; l=new C();* } as terminating. Note that this loop is not guaranteed to terminate since l potentially changes the location of *size* and hence its value.

Condition 2 guarantees that we do not change any of the intermediary reference fields $l.r_1, \ldots, l.r_1...r_n$. Note that if we modify a reference field $l.r_1...r_i$ then we fail to ensure constancy of the local field access. For instance, we would fail to prove termination of this loop *while (l.*r_1*.size < 10)* {*l.*r_1*.size++; l'.*r_1*=z;* }. This is a safe assumption, as without knowledge about the aliasing of l and l', we might be changing the reference to *size*.

Condition 3 is a sufficient condition to ensure that the field is not updated due to possible aliasing with another object (point (ii) in Definition 3.4.1). This condition is not satisfied in a loop of the form *while*

```
class A {
  int f,g;
  int m1(){return 1;}
};
abstract class B extends A {
  int m1(){return 2;}
  void m2(){
    f = f + 1;
  }
  abstract void m3();
};
class C extends B {
  void m3() { g=g-1; }
};
```

```
void test1(A a,int k) {
  while (a.f < k) a.f = a.f + a.m1();
}
void test2(B b,int k) {
  while (b.f < k) b.m2();
}
void test3(B a,int k) {
  while (a.f < k){
     a.m3();
     a.f = a.f + a.m1();}
}
```

Figure 3.13: Termination with fields and method invocations

(l.size < 10) {*l.size++; l'.size--;* } and therefore we do not prove termination for it. This is reasonable, as l and l' might be aliased during the execution.

Example 8 *Reconsider the third loop in Section 3.4.2. For clarity, we replace the access to the field* count *to explicitly include the* this *path variable:*

```
for(; this.count<newLength; this.count++) value[this.count] = '\0';
```

We can prove that this.count *is local to the loop by checking the syntactic conditions stated above: the reference this does not change; and all updates to this.count are done through the field access this.count. The key point is that, since* this.count *is local, we can safely treat it as local variable. Consequently, existing termination analysers [7] are able to infer that this.count is increasing at each iteration. Besides, as newLength remains constant in the loop, the analyzer finds out that newLength-this.count is a decreasing well-founded measure and thus termination is guaranteed.* □

3.4.3 Termination with (Virtual) Method Invocations

In this section, we address the more challenging problem of proving the termination of loops which contain method invocations. As notation, we denote by $M(L)$ the set of methods transitively invoked within the scope of a loop L. We now study what are the conditions that the methods in $M(L)$ must satisfy in order to preserve the locality condition on $g\text{-}fields(L)$.

Consider a method m invoked within L, we distinguish three possible scenarios. In the first two ones, the implementation of m is available at analysis time and thus we can apply the techniques to detect local field accesses to the code in m. As our method is purely syntactic, in order to check the conditions on m, first we must do a *renaming* between the variables in the call and the formal parameters in m, as parameter passing does. Note that, when a method m is invoked from a reference l, the *this* reference in m is renamed to l in order to check the conditions. In the first scenario, method m does not modify the value of the (numeric) field, whereas in the second one it does. In the third one, the implementation of m either it is not available (i.e., it is an abstract or native method) or it has been redefined by means of subclassing. We aim at proving *modular* termination of the loop by making assumptions on m. We study these scenarios in more detail below.

Scenario 1. Consider method test$_1$ at the top of the right-hand column in Fig. 3.13. Due to dynamic dispatching, the execution of a.m$_1$() can correspond to method m$_1$ in class A or to method m$_1$ in class B. Since, in both cases, the reference variable a remains constant and the field a.f is not updated within either implementation of m_1, we can guarantee that the field access a.f is local to (the loop in) test_1. Proving termination now is straightforward since both implementations of m_1 return a positive number.

Scenario 2. Now, we consider the case that, even if the field access is local to the loop, the field is updated during the execution of the invoked method. This happens, for example, in method test_2 where the call $\mathsf{b.m}_2()$ increments the value of b.f. Indeed, method m_2 is responsible for the termination of test_2. In this case, we need to track the variations in the field b.f in an inter-procedural manner. One way to do it is by *inlining* the invoked method. However, this cannot always be done, as it is problematic for recursive methods. Another approach is to transform the methods in such a way that they carry as additional parameters the fields that must be tracked. When we have virtual invocations and several instances of the same method can be executed at runtime, we need to do such transformation to all the possible instances. Doing it at the level of Java would require a more sophisticated transformation, since parameters are passed by value. It could, however, be easily integrated in a termination analyzer like [15], as it works on an intermediate representation with permits multiple output parameters. We plan to develop this part in an extended version of this work.

Scenario 3. If the code of a method m in $M(L)$ is not available or the implementation of the method has been redefined, unfortunately we can say very little about the termination of L. For instance, if m is an abstract method, it is customary that the user defines a new class which implements m and it is always possible that it modifies the fields which affect the termination of the loop. Also, the new implementation might introduce callbacks which endanger termination. Clearly, one possibility is, once the implementation is available, to re-analyze the loop with the new method. More interestingly, we can try to prove *modular* termination of the loop by assuming that (1) the method terminates, (2) it does not update any field access in *g-fields* and (3) it does not have callbacks. Once the new implementation is available, we actually have to ensure that the method m does not introduce a termination problem in L by checking the first two syntactic conditions in Sect. 3.4.2 as well as proving termination of m by applying our method to m again. For instance, consider method test_3, which is similar to test_1, but where a call to the (abstract) method m_3 has been added in the body of the loop. Assume that the class C is not available, then we make the assumption that m_3 is terminating and does not update a.f. Under these assumptions, we can prove modular termination of the loop. Consider now that the user defines class C at the bottom. Trivially, this method terminates and besides we can ensure that a.f is never updated from it. Note that, if the update inside m_3 was on f instead of on g, we would fail to ensure that that m_3 does not interfere with the guard. Indeed, the loop does not terminate in this case.

Method Invocations in the Java Libraries

It is common to find loops for scenarios 1 and 3 in the Java libraries. For instance, the loop of *Pattern #2* of Section 3.4.2 is an example of scenario 3. The method fixupVariables invoked by m_args[i] is an abstract method of the library org.apache.xpath.Expression. The code is not available, thus we can only aim at proving termination modularly. We first make the assumption that fixupVariables will not introduce a termination problem in the loop. Under this assumption, we can prove termination of the loop. Note that, for actual implementation of fixupVariables, we will have to check that the local access condition holds and that it terminates.

We found many loops for scenario 1. For instance, the following loop appears in method public int indexOf(Object elem) of the library java.util.ArrayList:

```
for (int i = 0;i<size;i++)
    if (elem.equals(elementData[i])) return i;
```

where size is a field of type int. Its termination depends on the termination of the calls to elem.equals(elementData[i]), where elem and elementData[i] are objects of class java.lang.Object. The implementation of equals is available and contains as unique instruction return (this==obj), which ensures the local field access of size. Thus the loop is definitely terminating. It is rare in the libraries to find loops for scenario 2, indeed we have not found any. Though we believe it is necessary to provide solutions for them in order to handle the termination of user-defined programs which rely on the libraries and define methods which actually update the fields.

It is important to note that the solution we have proposed for this scenario is valid as long as the implementation of the missing methods does not use static fields. The reason for this is that static fields can be, similarly to global variables, used in the code without being passed as arguments to the method. Therefore, the set of classes reachable from a method signature, as obtained by the procedure above, is not guaranteed to be a safe approximation of the actual classes reached by execution in the presence of static fields.

Perspectives for Future Work

The state of the practice in termination analysis is moving beyond less-widely used programming languages to realistic *object-oriented* languages. This paper draws attention to some difficulties that need to be solved if object *fields* are to be supported by termination analyzers. In particular, tracking size information becomes rather complex, and accurate *aliasing* information is required in order to track all possible updates of the corresponding fields. Motivated by examples found in the Java libraries, we have proposed some ideas towards dealing with numeric fields in a practical manner. The perspectives on the application of our technique include to infer *termination annotations* for as many methods in the Java libraries as possible. Applying termination tools on realistic programs which use libraries is a challenging problem, as there are many dependencies between the library classes and, in our experience, even small applications require analyzing a high number of library methods. By using precomputed annotations, the analyzer can safely assume the termination of those annotated methods in the Java libraries (and those that they depend upon).[4]

Although our ideas have not been experimentally evaluated yet, we believe that most of the patterns found in the libraries match those presented in Sec. 3.4.2. We, nevertheless, plan to improve the accuracy of the analysis in order to cover a broader range of patterns. For instance, as a starting point, we have proposed to check the local field access condition on those fields which appear explicitly in the guards, denoted *g-fields*. There are, of course, other possibilities and enhancements:

- Ideally, we should try to prove the locality condition not only on *g-fields*, but also on those fields which may interact with *g-fields*. For instance, in a loop of the form *while (l.size < 10)* {*l.size+= l'.size;* }, unless we track some information about $l'.size$ (in this case, its sign would suffice), we will fail to prove termination. Unfortunately, it is not always trivial to determine the minimal set of fields which may interact with *g-fields*. In particular, a simple syntactic inspection is not enough.

- To simplify the above point, another idea would be to try and prove the locality condition on *all* fields which appear inside the scope of the loop. This approach would be in general more accurate (e.g., would solve the above problem) but more expensive. Importantly, even if not all fields are local to the loop, the termination analysis proceeds (step 3 in Sect. 3.4.2). As long as the non-local field accesses do not affect the termination behaviour, the analysis can still succeed to prove termination.

- Another interesting refinement is to consider not only the fields which appear explicit in the guards but also those which are accessed through *getter* methods like *while (l.getSize() < 10)* {*...*}. For this, we should go through the code of the methods invoked in the loop guards and identify those fields. A simple solution to this problem is inlining the method. Afterwards, the same basic techniques explained in the paper could be applied.

It can be seen that in some cases there is an accuracy vs efficiency tradeoff and also that, what it is optimal for one example might not be good for others. We need to perform experimental evaluation to assess the different options.

[4] Note that precomputed assertions are valid as long as the user does not redefine methods which have been used (and analysed) to infer the assertions.

From an implementation perspective, we plan to enhance the COSTA system [7] with the ideas presented in this paper. COSTA is a cost and termination analyzer which works directly on the bytecode (and has no knowledge about the source Java). The termination module is based on the techniques proposed in [15] and the cost module on the method described in [5]. To carry out the implementation, the first issue is to incorporate the syntactic conditions to prove whether fields are accessed locally. Condition 3 can be easily checked on the bytecode by seeing that there is no `putfield` to the corresponding field signatures. Checking that the object does not change (conditions 1 and 2) requires to track dependencies between stack variables and local variables. This happens because, in the bytecode, the access to a field is done by first pushing the variable (on which the condition is to be checked) to the stack and then the field is accessed from the stack variable. This check can be done syntactically in most cases due to the elimination of stack variables [8]. Once the syntactic conditions are checked, we will implement the extensions to treat fields as local variables during analysis. This is straightforward to do in COSTA, as the tool converts the bytecode into a rule-based representation where the local variables (and the stack positions) appear as arguments of these rules. We can just add the required fields as additional arguments to them. Size analysis will directly treat them as it does with local variables in order to infer how they increase/decrease over the program.

3.4.4 The COSTA System

The system COSTA is 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 a specific method. 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 [15] for details). The system COSTA can deal with programs of realistic size and complexity, including programs involving Java libraries.

The Architecture of the COSTA System

COSTA [7] is implemented in Prolog and uses the Parma Polyhedra Library (PPL) [21] for manipulating *linear constraints*. It deals with the *Java bytecode* (JBC) language [65], which is slightly more complicated than the bytecode language used so far in the paper. In particular, (1) the instruction set includes different variants of the same instruction, parametrically on 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, so that there exists a *Return* instruction; (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.

Figure 3.14 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 like CFG build indicate the *main steps* of the process, while ellipses like CFG represent *what* the system produces at each stage. 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). Several optimizations make the analysis more efficient and accurate: in particular, *class analysis*, *static single assignment*, *loop extraction*, *constant propagation* and *stack variables elimination*.

Class analysis [100] 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 us to exclude

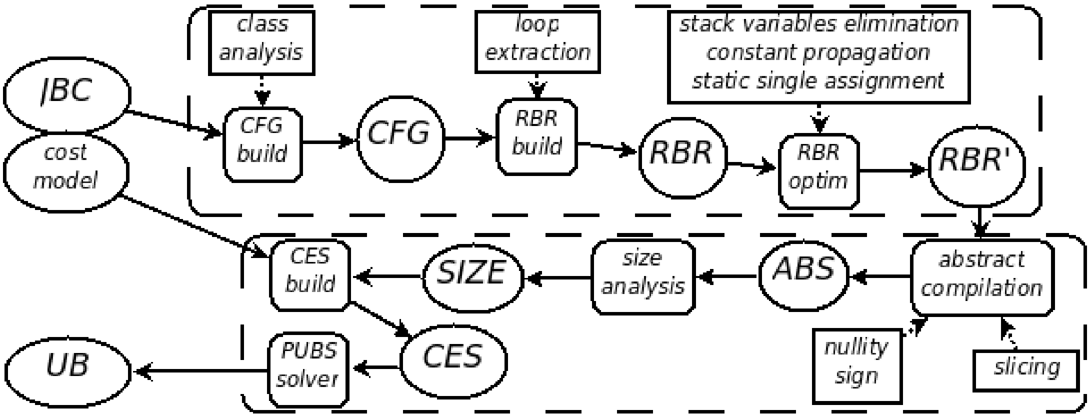


Figure 3.14: Architecture of COSTA

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) trasformation (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. Unification can also remove, in many cases, stack variables which are only used to perform simple operations.

As another important optimization, COSTA is able to *detect* and *extract loops* from CFGs. Indeed, when analyzing low-level languages, 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 us to reason 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 [16], but, to the best of our knowledge, its use in static analysis of Java bytecode is new. COSTA implements an efficient algorithm [102], modified to extract loops which, in addition to have 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.

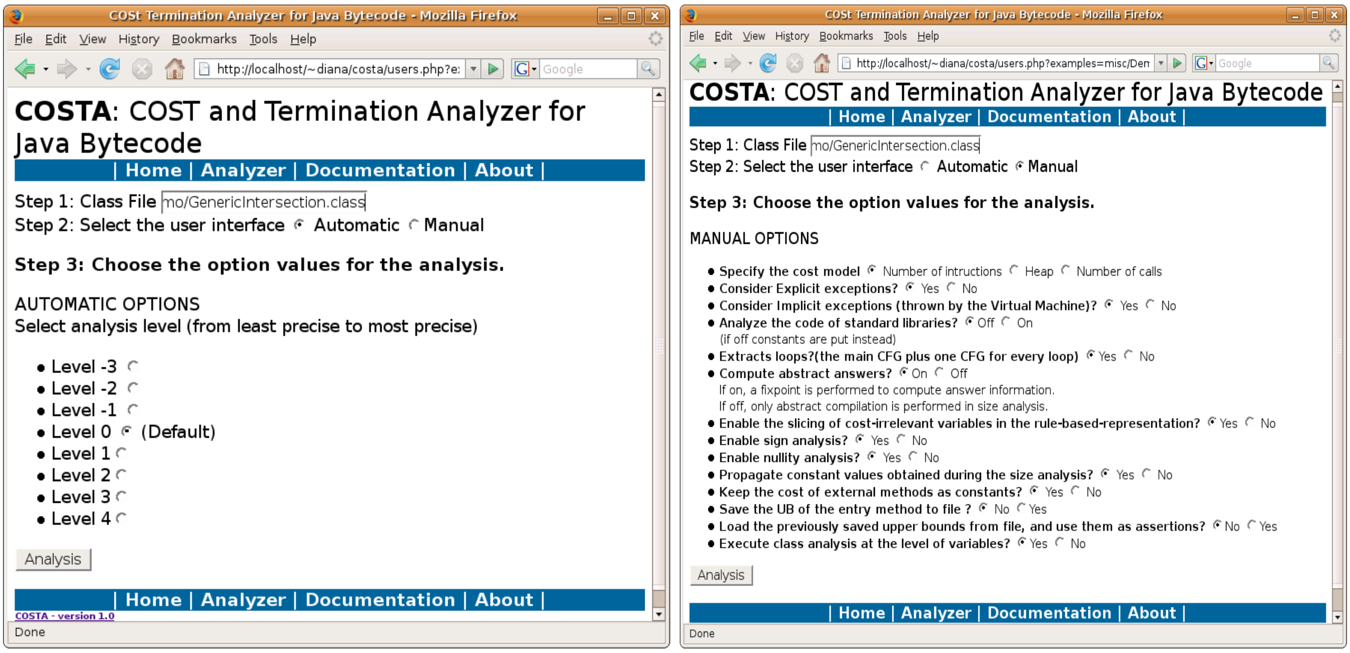
In phase (2), depicted in the lower half of the figure, cost analysis on the RBR is performed. *Abstract compilation*, which is helped by auxiliary static analyses, prepares the input to *size analysis* (Section **??**). COSTA relies on a series of static analysis techniques, such as *sign* and *nullity* analysis. Such analyses help in statically excluding some specific behaviors of the program: 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 $o.m$. When possible, such techniques are implemented in an optimized way to suitably account for the specific problems of cost analysis. Afterwards, COSTA sets up a *cost relation system* (CRS) for the selected *cost model*. 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. *Slicing* of the RBR removes variables which are *useless* in cost analysis. Finally, COSTA integrates the dedicated *upper bound solver* PUBS [2], which finds *closed-form solutions* for CRSs.

In some sense, the system is still a prototype, but many things have been considerably improved from the first versions. Currently, it can deal with quite a large class of JBC programs, and gives reasonable results in terms of precision and efficiency. COSTA allows to choose between several *options*, setting, for example,

1. if the code of *external libraries* (i.e., library methods 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 **??**);
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 currently supported. As for *native code*, a call to a native method is replaced by a symbolic constant, instead of an upper bound for the cost. Indeed, this is the only sensible, general solution, since (for example, when counting the number of instructions) it simply does not make sense to analyze programs which are not written in Java bytecode.

We have recently developed a COSTA web interface [4]. It allows users to try out the system on a set of representative examples, and also to upload their own bytecode programs. As the behaviour of COSTA can be customized using a relatively large set of options, the web interface allows two different alternatives for choosing the values for such options.


COSt Termination Analyzer for Java Bytecode - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/~diana/costa/users.php?e:
Google
| Home | Analyzer | Documentation | About |
Analysis
COSTA - version 1.0
Done
COSt Termination Analyzer for Java Bytecode - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/~diana/costa/users.php?examples=misc/Dem
Google
| Home | Analyzer | Documentation | About |
Analysis
Done

COSTA: COST and Termination Analyzer for Java Bytecode

Step 1: Class File mo/GenericIntersection.class
Step 2: Select the user interface ● Automatic ○ Manual

Step 3: Choose the option values for the analysis.

AUTOMATIC OPTIONS
Select analysis level (from least precise to most precise)

- Level -3 ○
- Level -2 ○
- Level -1 ○
- Level 0 ● (Default)
- Level 1 ○
- Level 2 ○
- Level 3 ○
- Level 4 ○

COSTA: COST and Termination Analyzer for Java Bytecode

Step 1: Class File mo/GenericIntersection.class
Step 2: Select the user interface ○ Automatic ● Manual

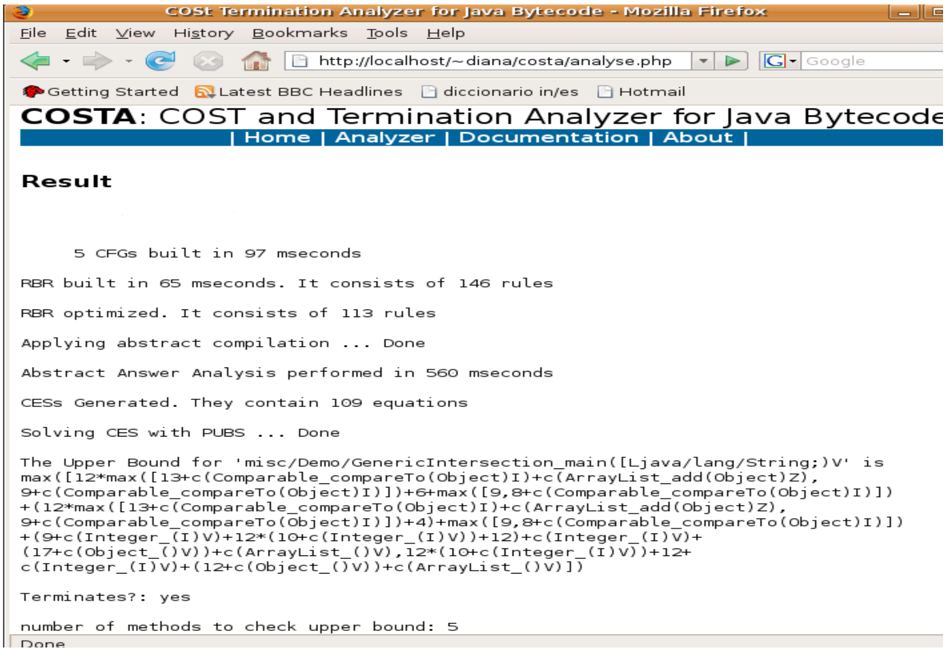
Step 3: Choose the option values for the analysis.

MANUAL OPTIONS

- **Specify the cost model** ● Number of intructions ○ Heap ○ Number of calls
- **Consider Explicit exceptions?** ● Yes ○ No
- **Consider Implicit exceptions (thrown by the Virtual Machine)?** ● Yes ○ No
- **Analyze the code of standard libraries?** ● Off ○ On
  (if off constants are put instead)
- **Extracts loops?(the main CFG plus one CFG for every loop)** ● Yes ○ No
- **Compute abstract answers?** ● On ○ Off
  If on, a fixpoint is performed to compute answer information.
  If off, only abstract compilation is performed in size analysis.
- **Enable the slicing of cost-irrelevant variables in the rule-based-representation?** ● Yes ○ No
- **Enable sign analysis?** ● Yes ○ No
- **Enable nullity analysis?** ● Yes ○ No
- **Propagate constant values obtained during the size analysis?** ● Yes ○ No
- **Keep the cost of external methods as constants?** ● Yes ○ No
- **Save the UB of the entry method to file ?** ● No ○ Yes
- **Load the previously saved upper bounds from file, and use them as assertions?** ● No ○ Yes
- **Execute class analysis at the level of variables?** ● Yes ○ No

Figure 3.15: Two ways of setting values for analysis options

The first alternative, which we call *automatic* (see Figure 3.15, left-hand side), 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. All this, without requiring 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 3.15, right-hand side) and it is meant for the expert user. There, the user has access to all of the analysis options available, allowing a fine-grained control over the behaviour of the analyzer. Some of these options include whether to analyze the Java standard libraries, 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. In the demo, we will show analyses using different cost models and also analyze applications for both Standard Edition Java and Micro Edition Java (in particular, for the MIDP profile for mobile phones).


COSt Termination Analyzer for Java Bytecode - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/~diana/costa/analyse.php
Google
Getting Started
Latest BBC Headlines
diccionario in/es
Hotmail
COSTA: COST and Termination Analyzer for Java Bytecode
| Home | Analyzer | Documentation | About |
Result

```
    5 CFGs built in 97 mseconds

RBR built in 65 mseconds. It consists of 146 rules

RBR optimized. It consists of 113 rules

Applying abstract compilation ... Done

Abstract Answer Analysis performed in 560 mseconds

CESs Generated. They contain 109 equations

Solving CES with PUBS ... Done

The Upper Bound for 'misc/Demo/GenericIntersection_main([Ljava/lang/String;)V' is
max([12*max([13+c(Comparable_compareTo(Object)I)+c(ArrayList_add(Object)Z),
9+c(Comparable_compareTo(Object)I)])+6+max([9,8+c(Comparable_compareTo(Object)I)])
+(12*max([13+c(Comparable_compareTo(Object)I)+c(ArrayList_add(Object)Z),
9+c(Comparable_compareTo(Object)I)])+4)+max([9,8+c(Comparable_compareTo(Object)I)])
+(9+c(Integer_(I)V)+12*(10+c(Integer_(I)V))+12)+c(Integer_(I)V)+
(17+c(Object_()V))+c(ArrayList_()V),12*(10+c(Integer_(I)V))+12+
c(Integer_(I)V)+(12+c(Object_()V))+c(ArrayList_()V)])

Terminates?: yes

number of methods to check upper bound: 5
Done
```

Figure 3.16: Results

Figure 3.16 shows the output of COSTA on an example program. In addition to showing the result of termination analysis and an upper bound on the execution cost, some data is displayed about the intermediate steps performed by the analyzer. In this case, the program is proved to terminate and an upper bound is shown which includes the cost of calls to several Java library methods.

Chapter 4

Conclusion

This deliverable has presented essentially three notions of certificates of varying degree of flexibility and specific adaptation to Java Bytecode. Foundational Certificates, Reduced Certificates in Abstraction-Carrying Code, and Tables of Lemmas. Other deliverables present two further notions of certificates: proof scripts in the base logic and verification conditions emitted by a proof-transforming compiler.

The process of identifying appropriate formats of certificates has herewith been completed. The type systems and analyses to be developed in WP2 will show to what extent the foundational notions of certificate remain viable. We expect that this will also depend on whether on-device checking will be successfully implemented within MOBIUS and if so under which platform. Efforts to run OCAML and hence COQ on small devices are currently underway; depending on the outcome on-device checking of foundational certificates based on Bicolano may become possible.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* PUB-AW, 1974.

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

[3] 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.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Ramírez, and D. Zanardini. The COSTA cost and termination analyzer for java bytecode and its web interface (tool demo). In Anna Philippou, editor, *22nd European Conference on Object-Oriented Programming*, July 2008.

[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *ESOP 2007* [43], pages 157–172.

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in cost analysis of Java bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2007.

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: A cost and termination analyzer for Java bytecode. In *BYTECODE*, ENTCS. Elsevier, 2008. To appear.

[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *SAC - Software Verification Track (SV08)*, pages 368–375, Fortaleza, Brasil, March 2008. ACM Press, New York.

[9] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced certificates for abstraction-carrying code. In *International Conference on Logic Programming*, number 4079 in Lecture Notes in Computer Science, pages 163–178. Springer-Verlag, August 2006.

[10] E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Typed-based homeomorphic embedding for online termination. In *Logic-based Program Synthesis and Transformation*, August 2007.

[11] 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.

[12] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Towards Java bytecode verification using tools for logic programming. In *Workshop on Software Verification and Validation*, August 2006.

[13] E. Albert, M. G'omez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in Lecture Notes in Computer Science, pages 124–139. Springer-Verlag, January 2007.

[14] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code: A model for mobile code safety. *New Generation Computing*, 2007.

[15] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano 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*, number 5051 in Lecture Notes in Computer Science, pages 2–18. Springer-Verlag, 2008.

[16] F. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler optimization*, pages 1–19, 1970.

[17] M. Allen and S. Horowitz. Slicing java programs that throw and catch exceptions. In *ACM PEPM*, 2003.

[18] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105, 4-7 June 1990, Philadelphia, Pennsylvania, USA, 1990. IEEE Computer Society.

[19] J-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[20] A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.

[21] 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. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.

[22] B. Barras and B. Grégoire. On the role of type decorations in the calculus of inductive constructions. In C.-H. Luke Ong, editor, *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Proceedings*, volume 3634 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, August 2005.

[23] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of Proof Obligations from Java to the Java Virtual Machine. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.

[24] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *ESOP 2007* [43], pages 125–140.

[25] R. Bellman. On a routing problem. 16:87–90, 1958.

[26] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2), 2004.

[27] F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, 2006.

[28] 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. Extended version.

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

[30] A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.

[31] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *Journal of Computer Security*, 10(3):273–296, 2002.

[32] L. Burdy, M. Huisman, and M. Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering*, volume 4422 of *LNCS*, pages 215–229. Springer-Verlag, 2007.

[33] D. Cachera, Thomas P. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Formal Methods Europe*, pages 91–106, 2005.

[34] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247, 2007.

[35] 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.

[36] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *JLP*, 41(1):103–123, 1999.

[37] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[38] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[39] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *European Symposium on Programming*, pages 21–30. Springer-Verlag, 2005.

[40] 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. ACM Press, 1978.

[41] S. K. Debray. Profiling prolog programs. *Software Practice and Experience*, 18(9):821–839, 1983.

[42] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower bound cost estimation for logic programs. In *Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[43] *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24—April 1, 2007*, number 4421 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[44] S. Genaim and F. Spoto. Information flow analysis for Java bytecode. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, January 2005.

[45] G. Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[46] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.

[47] G. Gomez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation.* ACM Press, 2002.

[48] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Improving the decompilation of Java bytecode to Prolog by partial evaluation. In *Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2007.

[49] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming*, pages 235–246. ACM Press, 2002.

[50] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics: Proceedings of the 18th International Conference TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Oxford, UK, August 2005. Springer-Verlag.

[51] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to pocklington certificates in type theory. *Symposium on Functional and Logic Programming*, 2006.

[52] C. Hankin, F. Nielson, and H. R. Nielson. *Principles of Program Analysis.* Springer-Verlag, 2005. Second Ed.

[53] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association of Computing Machinery*, 40(1):143–184, 1993.

[54] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[55] Patricia M. Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, ENTCS. Elsevier, 2006.

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

[57] J. Jouannaud and W. Xu. Automatic complexity analysis for programs extracted from Coq proof. *ENTCS*, 2006.

[58] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proc. of the 16th ACM Symp. on Theory of Computing*, pages 302–311. ACM Press, 1984.

[59] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 3(298):583–626, 2003.

[60] P. Laud, T. Uustalu, and V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292–310, 2006.

[61] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, New York, NY, USA, 2001. ACM Press.

[62] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.

[63] C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. In *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 451–465. Springer-Verlag, 2007.

[64] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, pages 63–77, Leuven, Belgium, 1997.

[65] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* PUB-AW, 1996.

[66] T. Lindholm and F. Yellin. *The Java$^{\text{TM}}$ Virtual Machine Specification. Second Edition.* Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[67] J. W. Lloyd. *Logic Programming.* PUB-SV, 1987.

[68] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In Laurie Hendren, editor, *International Conference on Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008.

[69] Claude Marché and Christine Paulin-Mohring. Reasoning about java programs with aliasing and frame conditions. In *TPHOL*, pages 179–194, 2005.

[70] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.

[71] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. An efficient, parametric fixpoint algorithm for analysis of java bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science. Elsevier, March 2007.

[72] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A flexible (c)lp-based approach to the analysis of object-oriented programs. In *Logic-based Program Synthesis and Transformation*, August 2007.

[73] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[74] D. Miller and V. Nigam. Incorporating tables into proofs. In J. Duparc and T. A. Henzinger, editors, *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 466–480. Springer-Verlag, 2007.

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

[76] Antoine Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[77] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.

[78] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, September 2007.

[79] Jorge Navas, Mario Méndez-Lojo, and Manuel Hermenegildo. A generic, context sensitive analysis framework for object oriented programs. In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, Technical Report, Nanjing University, pages 109–120, July 2007.

[80] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[81] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Logic in Computer Science*, pages 93–104, Los Alamitos, CA, 1998. IEEE Press.

[82] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. *ACM SIGPLAN Notices*, 36(3):142–154, 2001.

[83] NIST. Secure hash standard. Federal Information Processing Standards Publication 180-2 (FIPS PUB 180-2), August 2002.

[84] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Math. Program.*, 96(2):293–320, 2003.

[85] B. Pientka. Tabling for higher-order logic programming. In *Conference on Automated Deduction*, pages 54–69. Springer-Verlag, 2005.

[86] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, LNCS, 2004.

[87] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

[88] G. Puebla, E. Albert, P. Arenas, and M. Hermenegildo. On abstraction-carrying code and certificate-size reduction. In *Emerging Applications of Abstract Interpretation*, March 2006.

[89] G. Puebla and M. Hermenegildo. Optimized algorithms for the incremental analysis of logic programs. In *Static Analysis Symposium*, pages 270–284. Springer-Verlag, 1996.

[90] F. A. Rabhi and G. A. Manson. Using complexity functions to control parallelism in functional programs. TR. CS-90-1, Department of Computer Science, University of Sheffield, UK, 1990.

[91] Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 143–154. Springer-Verlag, 1997.

[92] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3–4):303–334, 2003.

[93] M. Rosendahl. Automatic complexity analysis. In *Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. Association of Computing Machinery, 1989.

[94] A. Roychoudhury, C. Ramakrishnan, and I. Ramakrishnan. Justifying proofs using memo tables. In *PPDP*, pages 178–189, 2000.

[95] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

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

[97] S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, pages 320–335, 2005.

[98] R. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, 1981.

[99] J. Slaney. Solution to a problem of Ono and Komori. *Journal of Philosophic Logic*, 18:103–111, 1989.

[100] F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *TOPLAS*, 25(5):578–630, 2003.

[101] G. Stengle. A nullstellensatz and a positivstellensatz in semialgebraic geometry. *Mathematische Annalen*, 207(2):87–97, 1973.

[102] W. Zou T. Wei, J. Mao and Y. Chen. A new algorithm for identifying loops in decompilation. In *SAS*, LNCS 4634, pages 170–183, 2007.

[103] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 2ed edition, 1951.

[104] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[105] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Rev.*, 38(1):49–95, 1996.

[106] M. Weiser. Program slicing. In *Proc. Int. Conf. on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[107] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[108] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher-Order Logics*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer-Verlag, 2004.

[109] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In M. Sagiv, editor, *European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2005.

[110] 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, August 2004.

[111] H. Xi and S. Xia. Towards array bound check elimination in java tm virtual machine language. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 1999.