Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

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

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D2.3

Report on Type Systems

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

Actual submission date: 2007-10-04

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

Organisation name of lead contractor for this deliverable: TUD

Revision of Deliverable 2.1

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 and Revision Table

Contributions

Site	Contributed to Chapter
INRIA	2, 3
CTH	2
IC	2, 4
TUD (previously RWTH)	1, 2, 5
LMU	3
UEDIN	3
UPM	3
ETH	4

Revision Table

Chapter	Difference to Deliverable 2.1
1	revision of chapter
2	revision of chapter including • general overview about the specific problem arising with unstructured programs • new results about types for programs in a concurrent (multithreaded) bytecode language • extension of the type system: new results for multithreading, and • new results for distributed programs in a bytecode language
3	revision of chapter including • extension of the heap space type system to loops and exceptions; integration with MOBIUS base logic • extension of the permission logic with loops; analysis of constraint resolution • new results about an external resource tracking and control library • extension of the execution time analysis to other costs; transfer to bytecode
4	revision of chapter including • refined results about Universe types formalization (UJ) • summary of refined results about UJ and concurrency • new results about generic Universe types • new results about multiple ownership
5	replacement

Executive Summary:
Report on Type Systems

This document summarises deliverable D2.3 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`.

We present final results that were obtained during the first two years of the MOBIUS project in WP2, Tasks 2.1 (Types for Information flow Security), 2.3 (Types for Basic Resource Policies), and 2.5 (Alias Control Types). Type systems are an enabling technology for the MOBIUS Proof-Carrying Code architecture. We developed type systems for properties in all three domains, both at the level of bytecode and at higher language levels. We improved property coverage and language coverage as well as flexibility and scalability of type systems. The aspects of the bytecode language that we address include, for example, objects, exceptions, methods, and concurrency. For information flow control, we enforce a noninterference-like property. The adaption to concurrent programs (multithreading as well as distribution) made it necessary to revise the definition of the security property, but we preserved the underlying intuition as much as possible. The results are published in [27, 29, 30, 31, 115, 103, 104, 150, 151, 179, 178] and additional details are given in [32, 105, 152]. For resource control, we statically enforce bounds on the resource consumption, generate generic cost relations between input and resource consumption, and restrict the resource access through the MIDP-API. We cover generic cost properties for program runs as well as properties depending on single instructions such as API-calls. The results are published in [7, 8, 9, 10, 21, 40, 44, 88, 119, 118, 120, 137]. For alias control we cover generic class definitions and improve flexibility by permitting multiple ownership. The results are published in [68, 69, 128, 60, 54, 132] and additional details are given in [67, 73, 131].

Deliverable D2.3, the Report on Type Systems, is based on and substantially extends deliverable D2.1, the Intermediate Report on Type Systems. For Task 2.1, D2.3 additionally contains a general overview about the specific problem arising with unstructured programs and the new results for concurrent programs in a bytecode language including results for multithreading and distribution. For Task 2.3, D2.3 additionally contains the new results for the MOBIUS base logic implementation of a heap space type system, the permission analysis, the external resource tracking and control library with the bulk messaging example, and the generic cost analysis for Java bytecode. For Task 2.5, D2.3 additionally contains more refined results about topological and encapsulation properties of Universe types in Java (UJ), the new results about extending Universe types in order to avoid races, generic Universe types, and multiple ownership.

Contents

Chapter 1

Introduction

This deliverable reports the results on type systems for Mechanisms for Safe Information Flow (Chapter 2), Basic Resource Policies (Chapter 3), and Alias Control (Chapter 4).

This deliverable includes contributions from all MOBIUS partners involved in the Tasks covered by Tasks 2.1 (Types for information flow security), 2.3 (Types for basic resource policies), and 2.5 (Alias control types), namely INRIA, CTH, IC, RWTH, TUD, LMU, UEDIN, UPM, and ETH. This deliverable supersedes deliverable D2.1, the Intermediate Report on Type Systems.

Types are syntactically defined, automatically decidable assertions about program behaviour. The type systems developed here guarantee adherence to security and resource-related properties of mobile code. The soundness of a type system must be proved independently and anew for each type system developed. Type systems are an enabling technology for the MOBIUS Proof-Carrying Code (PCC) architecture because they are intuitive, automatic and scalable. Hence improvements of program coverage and language coverage as well as of flexibility and scalability, as they are presented in this deliverable, are important steps to build the MOBIUS PCC-architecture.

Chapter 2 describes a type system for security policies that control the flow of information in sequential bytecode programs and extensions of this type system for concurrent programs. In the development of the type system, we exploited many ideas that were first developed for simpler languages. The security type system presented in this deliverable is the first that can be applied to a realistic, low-level language such as Java bytecode that includes features such as objects, exceptions, methods and concurrency. In this way, type-based information flow security becomes applicable within MOBIUS.

Chapter 3 describes various type systems and static analyses for controlling resources, notably execution time, heap space, and access to external resources. Again, we strived for maximising the re-use of pre-existing work from within the consortium, but considerable adaptions and extensions were carried out to meet the MOBIUS requirements regarding language coverage and threat models. The work presented here improves and substantially extends the possible control on Java bytecode and the MIDP-API.

Chapter 4 differs from the previous two chapters in that the property guaranteed here is less intuitive to grasp in this case and alias control is to be seen as an auxiliary device to help other analyses and methodologies to fulfil their more tangible goals. In a nutshell, alias control introduces the ownership relation between objects and guarantees that certain kinds of accesses to an object are performed only by its owners. While this kind of property is trivial in a purely functional or procedural setting, it becomes delicate in the presence of pointers and aliasing as we find them in the JVM, hence in MOBIUS. Once established, alias control can then be instantiated in a number of ways, e.g., it can contribute to maintenance of security levels by requiring that owners of any object have the security level required to access it.

Chapter 2

Types for Information Flow Security

This chapter presents the security type system for bytecode to enforce information flow security, developed in the context of the MOBIUS project. Section 2.1 presents the type system that enforces information flow security for sequential bytecode. The type system is defined for bytecode with objects, methods and exceptions. Section 2.2 makes this type system applicable to concurrent, multithreaded programs. Section 2.3 presents further extensions to the type system improving its applicability for concurrent systems.

2.1 Security Types for Sequential Bytecode

The purpose of this section is to present a type system to enforce confidentiality of object-oriented applications executing on a sequential Java-like virtual machine, and to show that the type system enforces non-interference.

Information-flow policy Any information flow policy must specify[1] a lattice of security levels. The choice of the lattice depends on the nature of the property to be enforced, i.e. confidentiality or integrity, and on the granularity of the policy. In addition, any information flow policy must state the observational capabilities of the attacker. Many different models have been considered in the literature; in our work, we focus on attackers that can only observe the input and output of programs. Since we are dealing with an object-oriented virtual machine, the input is the set of parameters of the method and the initial heap, and the output is the result value and the final heap.

Policies and modularity In order to ensure its scalability and its compatibility with dynamic class loading, the Java bytecode verifier performs modular verification, and verifies each method independently using method signatures to simulate method calls and returns at type level. Thence, an important requirement of our work is that our information-flow type system should also operate on a method per method basis, and thus we are led to attach security signatures to methods; the idea of considering security signatures to methods is not new, and can be found e.g. in [22].

Virtual machine Our type system applies to a stack-based virtual machine that features Java-bytecode-like instructions for stack and heap manipulation, method invocation and exception handling. We follow closely the formal definition of the JVM semantics given in the Bicolano project [141]. The most significant

[1] Information flow security requirements relevant to global computing have been specified in MOBIUS deliverable 1.1 [123].

difference from the Java Virtual Machine is the assumption of an unbounded memory for the heap, but this is a standard assumption in formal verification. In this section we focus on sequential bytecode. Extensions to multithreading will be considered in the Sections 2.2 and 2.3.1. Moreover, we do not consider arrays, but an extension of this work for arrays can be found in [29]. Arrays require a different treatment than field accesses because of their dynamic nature. For example, we have to prevent an attacker to learn the value of a high index $\mathtt{i_H}$ by exploiting a low array reference a_L and looking at the result of an array access $\mathtt{a_L[i_H]}$. Finally, we do not consider subroutines, but this will be discussed when presenting future extensions of this work.

Our analysis is proven correct[2], and encompasses some major features of the JVM: objects, exceptions, and method calls. The work builds upon known techniques, especially from [22] and [25], but solves a number of non-trivial difficulties due to the complexity of the language (unstructured code, fine grain exception handling, ...).

Preliminaries We let $A^\star$ denote the set of A-stacks for every set A. We use hd and tl and :: and $+\!\!+$ to denote the head and tail and cons and concatenation operations on stacks.

Throughout the paper, we assume a given lattice $(\mathcal{S}, \leq, \sqcup, \sqcap)$ of security levels.

2.1.1 Security policy

A program in the JVM is composed of a set of classes. Each class includes a set of fields and a set of methods, including a distinguished method **main** that is the first one to be executed. Each method includes its code (set of labeled bytecode instructions), a table of exception handlers, and a signature that gives the type of its arguments and of its result[3]. We note $\mathsf{Handler}(i, C) = t$ when there is a handler at program point t for exception of class (or a subclass of) C thrown at program point i. We note $\mathsf{Handler}(i, C) \uparrow$ when there is no handler for exception of class (or a subclass of) C thrown at program point i.

A method takes a list of arguments, and may terminate normally by returning a value, or abnormally by returning an exception object if an uncaught exception occurred during execution, or may loop infinitely. We do not consider "wrong" executions that get stuck, as such executions are eliminated by bytecode verification. The semantics of methods is captured by judgments of the form $(h_i, lv) \Downarrow_m (r, h_f)$, meaning that executing the method m with initial heap h_i and parameters lv yields the final heap h_f and the result r, where r is either a return value, or an exception object. The definition of this judgment can be found in the Bicolano [141] formal semantics.

The security policy is based on the assumption that the attacker can only draw observations on the input/output behavior of methods. On the other hand, we adopt a termination insensitive policy which assumes that the attacker is unable to observe non-termination of programs. Formally, the policy is given by a lattice $(\mathcal{S}, \leq, \sqcup, \sqcap)$ of security levels, and:

- a security level $k_{\text{obs}} \in \mathcal{S}$ that determines the observational capabilities of the attacker. Essentially, the attacker can observe fields, local variables, and return values whose level is below k_{obs};
- a global policy $\mathsf{ft} : \mathcal{F} \to \mathcal{S}$ that attaches a security level to fields (we let $\mathcal{F}$ denote the set of fields). The global policy is used to determine a notion of equivalence $\sim$ between heaps. Intuitively, two heaps h_1 and h_2 are equivalent if $h_1(l).f = h_2(l).f$ for all locations l and fields f s.t. $\mathsf{ft}f \leq k_{\text{obs}}$; the formal definition of heap indistinguishability is rather involved and deferred to Section 2.1.4;

[2] Proofs can be consulted in a companion report [28]

[3] Methods may have a void return type, in which case they return no value. However, our description assumes for the sake of simplicity that all methods return a value upon normal termination.

- local policies for each method (we let $\mathcal{M}$ denote the set of methods). In a setting where exceptions are ignored, local policies are expressed using security signatures of the form $\mathbf{k_v} \xrightarrow{k_h} \mathbf{k_r}$ where $\mathbf{k_v}$ provides the security levels of the method's local variables (including method's arguments[4]), k_h is the effect of the method on the heap, and $\mathbf{k_r}$ (called *output level*) is a list of security levels of the form $\{n : k_{\mathrm{n}}, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, where k_{n} is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment (or due to an exception-throwing instruction) of level k_i. In the rest of the paper we will write $\mathbf{k_r}[\mathrm{n}]$ instead of k_{n} and $\mathbf{k_r}[e_i]$ instead of k_{e_i}. The vector $\mathbf{k_v}$ of security levels is used to determine a notion of indistinguishability $\sim_{\mathbf{k_v}}$ between arrays of parameters, whereas the output level is used to define a notion of indistinguishability $\sim_{\mathbf{k_r}}$ between execution outputs.

Essentially, a method is safe w.r.t. a signature $\mathbf{k_v} \xrightarrow{k_h} \mathbf{k_r}$ if:

1. two terminating runs of the method with $\sim_{\mathbf{k_v}}$-equivalent inputs, i.e. inputs that cannot be distinguished by an attacker, and equivalent heaps, yield $\sim_{\mathbf{k_r}}$-equivalent results, i.e. results that also cannot be distinguished by the attacker,
2. the method does not perform field updates on fields whose security level is below k_h—as a consequence, it cannot modify the heap in a way that is observable by an attacker that has access to fields whose security level is below k_h.

Formally, the security condition is expressed relative to the operational semantics of the JVM, which is captured by judgments of the form $(h_i, lv) \Downarrow_m (r, h_f)$, meaning that executing the method m with initial heap h_i and parameters lv yields the final heap h_f and the result r.

Then, we say that a method m is *safe* w.r.t. a signature $\mathbf{k_v} \xrightarrow{k_h} \mathbf{k_r}$ if its method body does not perform field updates on fields of level lower than k_h and if furthermore it satisfies the following non-interference property: for all heaps h_i, h_f, h'_i, h'_f, arrays of parameters $\mathbf{a}$ and $\mathbf{a}'$, and results r and r',

$$\left.\begin{array}{c}(h_i, \mathbf{a}) \Downarrow_m (r, h_f) \\ (h'_i, \mathbf{a}') \Downarrow_m (r', h'_f) \\ h_i \sim h'_i \\ \mathbf{a} \sim_{\mathbf{k_v}} \mathbf{a}'\end{array}\right\} \Rightarrow\ h_f \sim h'_f\ \wedge\ r \sim_{\mathbf{k_r}} r'$$

There are two important underlying choices in this security condition: first, the security condition focuses on input/output behaviors, and so does not consider the case of executions that loop infinitely; however, it also does not consider "wrong" executions that get stuck, as such executions are eliminated by bytecode verification. Second, the security condition is defined on methods, and not on programs, as we aim for a modular verification technique in the spirit of bytecode verification.

2.1.2 Dealing with unstructured programs

Preventing direct flows with stack types. Any sound information flow type system must prevent direct information leakages that occur through assigning secret values to public variables. In a high level language, avoiding such indirect flows is ensured by setting appropriate rules for assignments; in a typical type system for a high-level language [173], the typing rule for assignments is of the form

$$\frac{\vdash e :\ k \qquad k \leq \mathsf{vt}(x)}{\vdash x := e :\ \mathsf{vt}(x)}$$

[4]JVM programs use a fragment of their local variables to store parameter values.

where $\mathsf{vt}(x)$ is the security given to variable x by the policy and k is an upper bound of the security level of the variables occurring in the expression e. The constraint $k \leq \mathsf{vt}(x)$ ensures that the value stored in x does not depend of any variable whose security level is greater than that of x, and thus that there is no illicit flow to x.

In a low level language where intermediate computations are performed with an operand stack, direct information flows are prevented by assigning a security level to each value in the operand stack, via a so-called *stack type*, and by rejecting programs that attempt storing a value in a low variable when the top of the stack type is high:

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow \mathsf{vt}(x) :: st} \qquad \frac{P[i] = \mathsf{store}\ x \quad k \leq \mathsf{vt}(x)}{i \vdash k :: st \Rightarrow st}$$

where st represents a stack type (a stack of security levels) and $\Rightarrow$ represents a relation between the stack type before execution and the stack type after execution of load. $P[i]$ represents here the current instruction at program point i.

For instance, $x_L = y_H$ is rejected by any sound information flow type system for a while language, because the constraint $H \leq L$ generated by the typing rule for assignment is violated. Likewise, the low level counterpart

$$\begin{array}{l} \mathsf{load}\ y_H \\ \mathsf{store}\ x_L \end{array}$$

cannot be typed as the typing rule for load forces the top of the stack type to high after executing the instruction, and the typing rule for store generates the constraint $H \leq L$.

Preventing indirect flows via security environments Any sound information flow type system must also prevent information leakages that occur through the control flow of programs. In a high level language, avoiding such indirect flows is ensured by setting appropriate rules for branching statements; in a typical type system for a high-level language [173], the typing rule for if statements is of the form

$$\frac{\vdash e :\ k \qquad \vdash c_1 :\ k_1 \qquad \vdash c_2 :\ k_2 \qquad k \leq k_1, k_2}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 :\ k}$$

and ensures that the write effects of c_1 and c_2 are greater than the guard of the branching statement.

To prevent illicit flows in a low-level language, one cannot simply enforce local constraints in the typing rules for branching instructions: one must also enforce global constraints that prevent low assignments and updates to occur under high guards. In order to express the global constraints that are necessary to enforce soundness, we rely on additional information about the program, namely control dependence regions (cdr) which approximate the scope of branching statements. The cdr information:

- is defined relative to a binary successor relation $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ between program points, and a set $\mathcal{PP}_r$ of return points. The successor relation and the set of return points are defined according to the semantics of instructions. Intuitively, j is a successor of i if performing one-step execution from a state whose program point is i may lead to a state whose program point is j. Likewise, j is a return point if it corresponds to a return instruction. In the sequel, we write $i \mapsto$ if $i \in \mathcal{PP}_r$;
- is captured by a function that maps a branching program point i (i.e. a program point with two or more successors) to a set of program points $\mathit{region}(i)$, called the region of i, and by a partial function that maps branching program points to a junction point $\mathsf{jun}(i)$.

The intuition behind regions and junction points is that $\mathit{region}(i)$ includes all program points executing under the guard of i and that $\mathsf{jun}(i)$, if it exists, is the sole exit from the region of i; in particular, whenever

$\mathsf{jun}(i)$ is defined there should be no return instruction in $region(i)$. The properties to be satisfied by control dependence regions, called SOAP properties (Safe Over Approximation Properties), are further discussed in the next sections.

In the type system, we use cdr information in conjunction with a security environment that attaches to each program point a security level, intuitively the upper bound of all the guards under which the program point executes. More precisely, programs are checked against a security environment se and global constraints arise in the type system as side conditions in the typing rules for branching statements. For instance, the rule for if bytecode is of the form:

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in region(i),\ k \leq se(j')}{i \vdash k :: st \Rightarrow \cdots}$$

In order to prevent indirect flows, the typing rules for instructions with write effects, e.g. store and putfield, must check that the security level of the variable or field to be written is at least as high as the current security environment. For instance, the rule for store becomes:

$$\frac{P[i] = \mathsf{store}\ x \qquad k \sqcup se(i) \leq \mathsf{vt}(x)}{i \vdash k :: st \Rightarrow st}$$

The combination of both rules allows to prevent indirect flows. For instance, the standard example of indirect flow `if` ($\mathtt{y_H}$) $\{\mathtt{x_L} = \mathtt{0};\}$ `else` $\{\mathtt{x_L} = \mathtt{1};\}$ is compiled in our low-level language as

```
      load y_H
      ifeq l_1
      push 0
      store x_L
      goto l_2
l_1 : push 1
      store x_L
l_2 : ...
```

By requiring that $se(i) \leq \mathsf{vt}(x)$ in the store rule and by requiring a global constraint on the security environment in the rule for ifeq, the type system ensures that the above program will be rejected: $se(i)$ must be H if the store instruction is under the influence of a high ifeq, and thus the transition for the store instruction cannot be typed.

2.1.3 Type system

In this section, we define an information flow type system that guarantees safety of all methods in a program.

Extra security annotations

Bytecode verification for secure information flow requires not only verification of direct flows such as assignments of high values to low memories, but also verification of implicit flows, such as assignments to low memories in branches of the program that depend on high values. Tracking information flow via control flow in a structured language without exceptions is easy since the analysis can exploit control structure [**?**, 22]. For unstructured low level code, such as Java bytecode, implicit flows can be tracked using extra security annotations.

$$\frac{P_m[i] \in \{\mathsf{binop}\ op, \mathsf{push}\ c, \mathsf{pop}, \mathsf{load}\ x, \mathsf{store}\ x, \mathsf{ifeq}\ j\}}{i \mapsto^{\emptyset} i+1} \qquad \frac{P_m[i] \in \{\mathsf{ifeq}\ j, \mathsf{goto}\ j\}}{i \mapsto^{\emptyset} j} \qquad \frac{P_m[i] = \mathsf{return}}{i \mapsto^{\emptyset}}$$

$$\frac{P_m[i] \in \{\mathsf{getfield}\ f, \mathsf{putfield}\ f, \mathsf{throw}, \mathsf{invokevirtual}\ m_{\mathrm{ID}}\} \qquad \mathsf{Handler}(i, \mathbf{NullPointer}) = t}{i \mapsto^{\mathbf{NullPointer}} t}$$

$$\frac{P_m[i] \in \{\mathsf{getfield}\ f, \mathsf{putfield}\ f, \mathsf{throw}, \mathsf{invokevirtual}\ m_{\mathrm{ID}}\} \qquad \mathsf{Handler}(i, \mathbf{NullPointer}) \uparrow}{i \mapsto^{\mathbf{NullPointer}}}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad C \in \mathsf{classanalysis}(m, i) \qquad \mathsf{Handler}(i, C) = t}{i \mapsto^{C} t}$$

$$\frac{P_m[i] = \mathsf{throw} \qquad C \in \mathsf{classanalysis}(m, i) \qquad \mathsf{Handler}(i, C) \uparrow}{i \mapsto^{C}}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad C \in \mathsf{excanalysis}(m_{\mathrm{ID}}) \qquad \mathsf{Handler}(i, C) = t}{i \mapsto^{C} t}$$

$$\frac{P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad C \in \mathsf{excanalysis}(m_{\mathrm{ID}}) \qquad \mathsf{Handler}(i, C) \uparrow}{i \mapsto^{C}}$$

Figure 2.1: Successor relation

Class and exception analysis The class analysis returns an over-approximation of classes of exceptions of a program point while the exception analysis gives a superset of the escaping exceptions of each method. For the soundness of the information flow type system, we assume that both the class-analysis and the exception-analysis are in the trusted computing base. The type system exploits the information of these analyses to restrict the static flow graph of a program and hence reject less non-interferent programs.

Control dependence regions An analysis of *control dependence regions* (cdr) gives information about dependencies between blocks in the program due to conditional or exceptional instructions. This analysis can be statically approximated [30, 145]. These regions are used by the type system to prevent implicit flows.

A control dependence *region* for a branching instruction at program point i must include at least those program points that will not be reachable in all executions, or more precisely those program points that will be reachable in executions depending on instruction found in i. A *junction point* for a program point i is a program point that is not included in its control dependence region, but that is reachable from program points in the control dependence region and that will always be executed if program point i is executed first (however the junction point of i does not depend on the result of the execution of instruction at program point i).

In order to obtain a more accurate cdr analysis in presence of multiple exceptions, the analysis of regions is computed on top of the class analysis to refine the set of exceptions that can be thrown by the throw instruction.

The correctness of the cdr analysis is expressed using the successor relation $\mapsto^m$ on program points. The relation is decorated by an element (called *tag*) in $\{\emptyset\} + \mathcal{C}$ in order to reflect the nature of the underlying semantics step: $\emptyset$ for a normal step and $E \in \mathcal{C}$ for a step where an exception of class E has been thrown.

The definition of this new relation is given in Figure 2.1. This relation can be statically computed thanks to the handler function of each method.

Intuitively, $i \mapsto^{\tau} j$ means that there is an instruction at program point i whose execution is of kind τ and may lead to the program point j in the same method. $i \mapsto^{\tau}$ means that the execution of method m may end at program point i (normal return or uncaught exception). The formal definition of $\mapsto$ is given in Figure 2.1 . Successors of a throw instruction are approximated thanks to the class analysis result and successors of a invokevirtual thanks to the exception analysis result of the called method.

Formally, cdr results are associated not only to program points but also to tags:

$$region_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \to \wp(\mathcal{PP}) \qquad \mathsf{jun}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \rightharpoonup \mathcal{PP}$$

We call *return point* a point i such that there exists $\tau \in \{\emptyset\} + \mathcal{C}$ with $i \mapsto^\tau$. When necessary will write $i \mapsto j$ for $\exists \tau,\ i \mapsto^\tau j$. The following definition captures the expected properties of the cdr structure.

SOAP1: for all program points i, j, k and tags τ such that $i \mapsto j$, $i \mapsto^\tau k$ and $j \neq k$ (i is hence a branching point), $k \in region(i, \tau)$ or $k = \mathsf{jun}(i, \tau)$;

SOAP2: for all program points i, j, k and tags τ, if $j \in region(i, \tau)$ and $j \mapsto k$, then either $k \in region(i, \tau)$ or $k = \mathsf{jun}(i, \tau)$;

SOAP3: for all program points i, j and tags τ, if $j \in region(i, \tau)$ (or $i = j$) and j is a return point then $\mathsf{jun}(i, \tau)$ is undefined;

SOAP4: for all program points i and tags τ_1, τ_2, if $\mathsf{jun}(i, \tau_1)$ and $\mathsf{jun}(i, \tau_2)$ are defined and $\mathsf{jun}(i, \tau_1) \neq \mathsf{jun}(i, \tau_2)$ then $\mathsf{jun}(i, \tau_1) \in region(i, \tau_2)$ or $\mathsf{jun}(i, \tau_2) \in region(i, \tau_1)$;

SOAP5: for all program points i, j and tags τ, if $j \in region(i, \tau)$ (or $i = j$) and j is a return point then for all tags τ' such that $\mathsf{jun}(i, \tau')$ is defined, $\mathsf{jun}(i, \tau') \in region(i, \tau)$.

Junction points uniquely delimit ends of regions. SOAP1 expresses that successors of branching points belong to (or end) the region associated with the same kind as their successor relation. SOAP2 says that a successor of a point in a region is either still in the same region, or it is the junction point at its end. SOAP3 forbids junction points for a region which contains (or starts with) a return point. SOAP4 and SOAP5 express properties between region of a same program point but with different tags. SOAP4 says that if two differently tagged regions end in distinct points, the junction point of one must belong to the region of the other. SOAP5 imposes that the junction point of a region must be within every region which contains (or starts with) a return point and possesses a different tag.

These conditions allow to program a straightforward checker in order to verify that a given cdr result verifies them. Figure 2.2 presents an example of safe cdr for an abstract transition system.

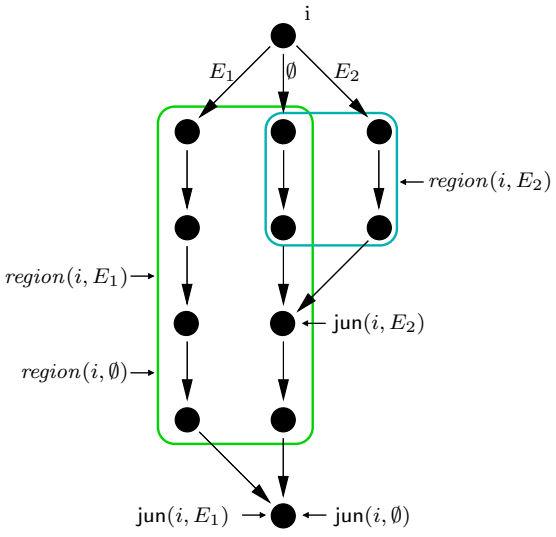

Figure 2.2: Example of cdr. Only relevant tags are presented here.

Typing judgment and typing rules

Typing rules impose constraints on stack types (stack of security levels) and security environments (mapping from program points to security levels). Stack types are used to track the security level of an expression whose evaluation has been compiled into a sequence of stack manipulations. Security environments give for each program point the security level of branching conditions on which its accessibility depends and are used to prevent implicit flow leaks.

The typing judgment considered is of the form

$$\Gamma, region, se, sgn, i \vdash^{\tau} st_1 \Rightarrow st_2 \qquad \Gamma, region, se, sgn, i \vdash^{\tau} st_1 \Rightarrow$$

where Γ is a table of method signatures, *region* a cdr result for the method under verification, *se* a security environment, *sgn* the security signature of the current method, i the current program point, τ the tag of the current transition, and st_1, st_2 two stack types.

The table Γ of method signatures is necessary for typing rules involving method calls — as in bytecode verification, we use the signature of other methods to perform the analysis in a modular way. This table associate to each method identifier[5] m_{ID} and security level $k \in \mathcal{S}$, a security signature $\Gamma_m[k]$. This signature gives the security policy of the method m called on object of level k (as in the type system [22] for source program). This allows a more flexible type system than having only one signature per method.

Figure 2.3 presents some selected typing rules. The full set of rules is available in a companion report [28].

Below we comment this selection of rules:

- The typing rule for ifeq requires that the result stack type is lifted (lift_k is the point-wise extension to stack types of $\lambda l.\ k \sqcup l$) with the level of the guard, i.e. the top of the input stack type. It is necessary to perform this lifting operation to avoid illicit flows through operand stack leakages.

  The following example illustrates why we need to lift the operand stack. This is a contrived example because it does not correspond to any simple source code, but it is nevertheless accepted by a standard bytecode verifier.

  $$\begin{array}{ll} & \mathsf{push}\ 0 \\ & \mathsf{push}\ 1 \\ & \mathsf{load}\ y_H \\ l_1: & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{swap}\\ \mathsf{pop}\\ \mathsf{goto}\ l_3\end{array}\right\} region(l_1) \\ l_2: & \mathsf{pop} \\ l_3: & \mathsf{store}\ x_L \end{array}$$

  In this example, the final value of variable x_L is equal to the value of y_H. So the program is interferent. It is nevertheless rejected by our type system, thanks to the lift of the operand stack at point l_1 that constrain the top of the stack at point l_3 to be a high value (store rule then prevents the assignment from high to low).

- The transfer rule for return requires $se(i) \leq k_r$ that avoids return instructions under the guard of expressions with a security level greater than $\mathbf{k_r}[\mathrm{n}]$. In addition, the rule requires that the value on top of the operand stack has a security level below $\mathbf{k_r}[\mathrm{n}]$, since it will be observed by the attacker.

[5] Associating signatures with a method identifier (*i.e.* a method name, a class name and a type signature) instead of a method allows to enforce that overriding of a method preserve its declared security signatures.

$$\frac{P_m[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathit{region}(i,\emptyset),\ k \leq \mathit{se}(j')}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\emptyset} k :: \mathit{st} \Rightarrow \mathrm{lift}_k(\mathit{st})}$$

$$\frac{P_m[i] = \mathsf{return} \qquad k \sqcup \mathit{se}(i) \leq \mathbf{k_r}[\mathrm{n}]}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\emptyset} k :: \mathit{st} \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \mathbf{k'_a} \xrightarrow{k'_h} \mathbf{k'_r} \\ k \sqcup k_h \sqcup \mathit{se}(i) \leq k'_h \qquad \mathsf{length}(\mathit{st}_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \mathbf{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(\mathit{st}_1) - 1],\ \mathit{st}_1[i] \leq \mathbf{k'_a}[i+1] \\ k_e = \bigsqcup \{\ \mathbf{k'_r}[e] \mid \exists e \in \mathsf{excanalysis}(m_{\mathrm{ID}}), \exists t \in \mathcal{PP}, \mathsf{Handler}(i,e) = t\ \} \\ \forall j \in \mathit{region}(i,\emptyset),\ k \sqcup k_e \leq \mathit{se}(j) \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\emptyset} \mathit{st}_1 :: k :: \mathit{st}_2 \Rightarrow \mathrm{lift}_{k \sqcup k_e}\ ((\mathbf{k'_r}[\mathrm{n}] \sqcup \mathit{se}(i)) :: \mathit{st}_2)}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \mathbf{k'_a} \xrightarrow{k'_h} \mathbf{k'_r} \\ k \sqcup k_h \sqcup \mathit{se}(i) \leq k'_h \qquad \mathsf{length}(\mathit{st}_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \mathbf{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(\mathit{st}_1) - 1],\ \mathit{st}_1[i] \leq \mathbf{k'_a}[i+1] \\ e \in \mathsf{excanalysis}(m_{\mathrm{ID}}) \qquad \forall j \in \mathit{region}(i,e),\ k \sqcup \mathbf{k'_r}[e] \leq \mathit{se}(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{e} \mathit{st}_1 :: k :: \mathit{st}_2 \Rightarrow (k \sqcup \mathbf{k'_r}[e]) :: \varepsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \mathbf{k'_a} \xrightarrow{k'_h} \mathbf{k'_r} \\ k \sqcup k_h \sqcup \mathit{se}(i) \leq k'_h \qquad \mathsf{length}(\mathit{st}_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \mathbf{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(\mathit{st}_1) - 1],\ \mathit{st}_1[i] \leq \mathbf{k'_a}[i+1] \\ e \in \mathsf{excanalysis}(m_{\mathrm{ID}}) \qquad k \sqcup \mathbf{k'_r}[e] \leq \mathbf{k_r}[e] \qquad \forall j \in \mathit{region}(i,e),\ k \sqcup \mathbf{k'_r}[e] \leq \mathit{se}(j) \qquad \mathsf{Handler}(i,e) \uparrow \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{e} \mathit{st}_1 :: k :: \mathit{st}_2 \Rightarrow}$$

$$\frac{\begin{array}{c} P[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup \mathit{se}(i) \sqcup k_2 \leq \mathsf{ft}(f) \qquad k_h \leq \mathsf{ft}(f) \\ \forall j \in \mathit{region}(i,\emptyset),\ k_2 \leq \mathit{se}(j) \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: \mathit{st} \Rightarrow \mathsf{lift}_{k_2} \mathit{st}}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup \mathit{se}(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ \forall j \in \mathit{region}(i, \mathbf{NullPointer}),\ k_2 \leq \mathit{se}(j) \qquad \mathsf{Handler}(i, \mathbf{NullPointer}) = t \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\mathbf{NullPointer}} k_1 :: k_2 :: \mathit{st} \Rightarrow k_2 \sqcup \mathit{se}(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup \mathit{se}(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ k_2 \leq \mathbf{k_r}[\mathbf{NullPointer}] \qquad \forall j \in \mathit{region}(i, \mathbf{NullPointer}),\ k_2 \leq \mathit{se}(j) \qquad \mathsf{Handler}(i, \mathbf{NullPointer}) \uparrow \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\mathbf{NullPointer}} k_1 :: k_2 :: \mathit{st} \Rightarrow}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathit{region}(i,\emptyset),\ k \leq \mathit{se}(j)}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\emptyset} k :: \mathit{st} \Rightarrow \mathrm{lift}_k((\mathsf{ft}(f) \sqcup \mathit{se}(i)) :: \mathit{st})}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathit{region}(i, \mathbf{NullPointer}),\ k \leq \mathit{se}(j) \qquad \mathsf{Handler}(i, \mathbf{NullPointer}) = t}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\mathbf{NullPointer}} k :: \mathit{st} \Rightarrow k \sqcup \mathit{se}(i) :: \epsilon}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \mathsf{Handler}(i, \mathbf{NullPointer}) \uparrow \qquad k \leq \mathbf{k_r}[\mathbf{NullPointer}]}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\mathbf{NullPointer}} k :: \mathit{st} \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classanalysis}(i) \cup \{\mathbf{NullPointer}\} \\ \forall j \in \mathit{region}(i,e),\ k \leq \mathit{se}(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{e} k :: \mathit{st} \Rightarrow k \sqcup \mathit{se}(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classanalysis}(i) \cup \{\mathbf{NullPointer}\} \\ k \leq \mathbf{k_r}[\mathbf{NullPointer}] \qquad \forall j \in \mathit{region}(i,e),\ k \leq \mathit{se}(j) \qquad \mathsf{Handler}(i,e) \uparrow \end{array}}{\Gamma, \mathit{region}, \mathit{se}, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{e} k :: \mathit{st} \Rightarrow}$$

Figure 2.3: Selected typing rules

The following example illustrates the need for preventing return instructions in high regions. It corresponds to a source program like `if` ($\mathtt{y_H}$) {`return 0;`} `else` {`return 1;`}.

$$\begin{array}{ll} & \mathsf{load}\ y_H \\ l_1: & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{push}\ 0 \\ \mathsf{return}\end{array}\right\} region(l_1) \\ l_2: & \left.\begin{array}{l}\mathsf{push}\ 1 \\ \mathsf{return}\end{array}\right\} \end{array}$$

This program is interferent because there is a return in a high ifeq. This program is rejected by the type system thanks to the ifeq rule which lifts the security environment, and the return rule which prevents the program from returning in a high security environment.

- The transfer rule for putfield requires that $k_1 \leq \mathsf{ft}(f)$, where k_1 is the security type of the object of the field, in order to prevent an explicit flow from a high value to a low field. The constraint $se(i) \leq \mathsf{ft}(f)$ prevents an implicit flow caused by an assignment to a low field in a high security environment. The constraint $k_2 \leq \mathsf{ft}(f)$ prevents modifying low fields of high objects that are alias to a low object. Finally, the constraint $k_h \leq \mathsf{ft}(f)$ prevents modification of field with a level not greater than the heap effect of the current method.

  The following example illustrates this last point. It corresponds to a source program like

  ```
  C xL = new C();
  zH = yH ? new C() : xL;
  zH.fL = 1;
  ```

  We assume that C is a class that has a low field named f_L. Let x_L be a low variable and y_H, z_H high variables.

$$\begin{array}{ll} & \mathsf{new}\ C \\ & \mathsf{store}\ x_L \\ & \mathsf{load}\ y_H \\ l_1: & \mathsf{ifeq}\ l_2 \\ & \left.\begin{array}{l}\mathsf{new}\ C \\ \mathsf{goto}\ l_3\end{array}\right\} region(l_1) \\ l_2: & \mathsf{load}\ x_L \\ l_3: & \mathsf{store}\ z_H \\ & \mathsf{load}\ z_H \\ & \mathsf{push}\ 1 \\ & \mathsf{putfield}\ f_L \end{array}$$

  In this program, depending on the test on y_H, variable x_L and z_H might be aliases to the same object (of class C). Hence, the assignment to field f_L might have side effect on the object in x_L. This program is rejected thanks to the putfield rule which avoids this type of leaks due to alias (with the constraint $k_2 \leq \mathsf{ft}(f)$ preventing assignments to low fields from high target objects).

- In the rule for getfield f the value pushed on the operand stack has a security level at least greater than $\mathsf{ft}(f)$ and the level k of the location (to prevent explicit flows) and at least greater than $se(i)$ for implicit flows.

- The typing rule for virtual call contains several constraints. The heap effect level of the called method is constrained in several manners. The goal of the constraint $k \leq k'_h$ is to avoid invocation of methods with low effect on the heap with high target objects. Two different target objects (in two executions) may mean that the body of the method to be executed is different in each execution. If the effect of the method is low ($k_h \leq k_{\text{obs}}$), then low memory is differently modified in both executions, leading to

leak of information. The constraint $se(i) \leq k'_h$ prevents implicit flows (low assignment in high regions) during execution of the called method. The constraint $k_h \leq k'_h$ prevents the called method to update fields with a level lower that k_h. It allows to avoid invocation of methods with low effect on the heap by a method with high effect.

Constraints $k \leq \mathbf{k'_a}[0]$ and $\forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \mathbf{k'_a}[i+1]$ link argument levels with formal parameter levels.

In the first typing rule, the next stack type is lifted with level $k \sqcup k_e$. Lifting with level k avoids indirect flows because of null pointer exception on the current object. k_e is greater that all levels of the exceptions that may escape from the called method. If abnormal termination of the called method reveals secret information then k_e is high and the next stack type must be high too. The security level of the return value is $(\mathbf{k'_r}[\mathrm{n}] \sqcup se(i))$. $\mathbf{k'_r}[\mathrm{n}]$ corresponds to the level of the return value in the context of the called method. $se(i)$ prevents implicit flow on the result after the virtual call.

The second and the third typing rule are parameterised by an exception e that may be caught by the called method. In the second rule, this exception is caught in the current method while in the third it is not. In both rule $k \sqcup \mathbf{k'_r}[e]$ gives an upper-bound on the information that can be gained by observing if the called method reached the point $i+1$. This level is hence used to constrain $region(i, e)$, the top of the stack when e is caught and the security level $\mathbf{k_r}[e]$ when it is not.

Typable programs

A program P is typable with respect to a table Γ and a family of safe cdr results $(region_m)_m$ (one by method), written $region \vdash P : \Gamma$, if for each declared method m and for each security signature sgn of m (w.r.t. Γ) there exist $S \in \mathcal{PP} \to \mathcal{S}^\star$ and a security environment se such that $\Gamma, region \vdash m : sgn, S, se$. We define $\Gamma, region \vdash m : sgn, S, se$ as follows:

- S_0 contains the empty stack type;
- for all program points i in m and j s.t $i \mapsto^\tau j$ there is st such that $\Gamma, region, se, sgn, i \vdash^\tau S_i \Rightarrow st$ and $st \sqsubseteq S_j$.
- if $i \mapsto^\tau$ then $\Gamma, region, se, sgn, i \vdash^\tau S_i \Rightarrow$.

The definition of a typable program is stated to ensure that runs of typable programs verify at each step the constraints imposed by the typing rules, provided they are called with parameters that respect the signature of their main method.

Typability of a method against its signature can be performed via a dataflow analysis based on Kildall's algorithm [85]. The analysis takes as inputs the local and global policies, the method table, the cdr structure, the security environment, the current signature, and either returns a type $S : \mathcal{PP} \to \mathcal{S}^\star$, or a tag indicating that type-checking has failed.

Assuming that the lattice of security levels satisfy the ascending chain property, i.e. that there is no infinite sequence of security levels

$$k_1 \sqsubset k_2 \sqsubset k_3 \ldots$$

it follows from the monotonicity of the typing rules that the analysis terminates.

We conclude this section by mentioning that there are alternatives to the definition of typable methods, and to verifying typability. One dimension of choice lies in the precision in the analysis: whereas our analysis is monovariant, our earlier work [25] adopted a polyvariant analysis in which types assign to each program

point a set of stack types. Polyvariant analyses rely on the finiteness of the set of stack types to guarantee termination. They type more programs, but yield less compact types.

Typable example

The following method may throw two kinds of exceptions: an exception of class `C` if the parameter `x` is true and of class **NullPointer** in the other case. The first exception depends on `x` while the second depends both on `x` and `y`. Normal return depends on `y` and `x` because execution terminates normally only if `y` is not *null* and `x` is `false`.

```
int m(boolean x,C y) throws C {
  if (x) {throw new C();}
  else {y.f = 3;};
  return 1;
}
```

At the bytecode level we obtain the following method:

$0:$ load x
$1:$ ifeq 4
$2:$ new C
$3:$ throw
$4:$ load y
$5:$ push 3
$6:$ putfield f
$7:$ const 1
$8:$ return

Such a method is typable with the signature

$$m : (x : L,\ y : H) \xrightarrow{H} \{n : H,\ C : L,\ \mathbf{NullPointer} : H\}$$

thanks to the cdr[6], the stack types and the security environment given below:

i	$region(i,\cdot)$	$\mathsf{jun}(i,\cdot)$	S_i	$se(i)$
0	$\emptyset$	1	ε	L
1	$\{2,3,4,5,6,7,8\}$	undef	$L :: \varepsilon$	L
2	$\emptyset$	3	ε	L
3	$\emptyset$	undef	$L :: \varepsilon$	L
4	$\emptyset$	5	ε	L
5	$\emptyset$	6	$H :: \varepsilon$	L
6	$\{7,8\}$	undef	$L :: H :: \varepsilon$	L
7	$\emptyset$	8	ε	H
8	$\emptyset$	undef	$H :: \varepsilon$	H

[6]In this example, it is safe to take the same cdr for all tags, so we do not distinguish them here.

The next method gives an example of code with method invocation where fine grain exception handling is necessary. To keep the example short, we here give a compressed version of a compiled code.

foo :
0 load x_L
1 load y_H
2 invokevirtual m
3 store z_H
4 load o_L
5 push 1
6 putfield f_L
handler : $[0, 2]$, NullPointer $\rightarrow 3$

i	$region(i, \emptyset)$	jun$(i, \emptyset)$	$region(i, NP)$	jun(i, NP)	$region(i, C)$	jun(i, C)	S_i	$se(i)$
0	$\emptyset$	1	$\emptyset$	1	$\emptyset$	1	ε	L
1	$\emptyset$	2	$\emptyset$	2	$\emptyset$	2	$L :: \varepsilon$	L
2	$\emptyset$	3	$\emptyset$	3	$\{3, 4, 5, 6, \ldots\}$	$\ldots$	$H :: L :: \varepsilon$	L
3	$\emptyset$	4	$\emptyset$	4	$\emptyset$	4	$H :: \varepsilon$	L
4	$\emptyset$	5	$\emptyset$	5	$\emptyset$	5	ε	L
5	$\emptyset$	6	$\emptyset$	6	$\emptyset$	6	$H :: \varepsilon$	L
6	$\{\ldots\}$	$\ldots$	$\{\ldots\}$	$\ldots$	$\{\ldots\}$	$\ldots$	$L :: L :: \varepsilon$	L

Update $o_L.f_L = 1$ at point 6 is accepted if and only if $se(5)$ and $se(6)$ are low. Thanks to the fine grain regions, typing rule for virtual call only propagate exception levels of m in distinct regions:

$$\forall j \in region(i, \mathbf{NullPointer}) = \emptyset,\ \mathbf{k_r}[\mathbf{NullPointer}] = H \leq se(j)$$
$$\forall j \in region(i, C) = \{3, 4, 5, 6, \ldots\},\ \mathbf{k_r}[C] = L \leq se(j)$$

It follows that $se(5)$ and $se(6)$ are low and the update is accepted by our type system.

2.1.4 Non interference theorem

Memory model

The memory model is summarised in Figure 2.4. During the execution of a method values manipulated by the JVM are either numerical values (taken in a set $\mathcal{N}$), locations (taken in an infinite set $\mathcal{L}$), or simply the *null* constant. Method computation is done on states of the form $\langle h, pc, \rho, s\rangle$ where h is the heap of objects, pc is the current program point, ρ is the set of local variables and s the operand stack. Heaps are modelled as a partial function $h : \mathcal{L} \rightharpoonup \mathcal{O}$, where the set $\mathcal{O}$ of objects is modelled as $\mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V})$, i.e. a class name and a partial function from fields to values. A set of local variables is a mapping $\rho \in \mathcal{X} \rightarrow \mathcal{V}$ from local variables to values. Operand stacks are lists of values. A method execution terminates on *final states*. A final state is either a pair $(\langle v\rangle_{\mathrm{v}}, h) \in \mathcal{V} \times$ Heap (normal termination), or a pair $(\langle l\rangle_{\mathrm{e}}, h) \in \mathcal{L} \times$ Heap (the method execution terminates because of an exception thrown on an object pointed by a location l, but not caught in this method).

$$
\begin{array}{rcll}
\mathcal{V} & = & \mathcal{N} + \mathcal{L} + \{null\} & \text{values} \\
\text{LocalVar} & = & \mathcal{X} \to \mathcal{V} & \text{local variables} \\
\text{OpStack} & = & \mathcal{V}^* & \text{operand stacks} \\
\mathcal{O} & = & \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}) & \text{objects} \\
\mathsf{Heap} & = & \mathcal{L} \rightharpoonup \mathcal{O} & \text{heap} \\
\text{State} & = & \mathsf{Heap} \times \mathcal{PP} \times \text{LocalVar} \times \text{OpStack} & \text{states} \\
\text{FinalState} & = & (\mathcal{V} + \mathcal{L}) \times \mathsf{Heap} & \text{final states}
\end{array}
$$

with

$$
\begin{array}{rcl}
\mathcal{N} & : & \text{the set of numerical values} \\
\mathcal{L} & : & \text{the set of locations} \\
\mathcal{X} & : & \text{the set of variable names} \\
\mathcal{C} & : & \text{the set of class names} \\
\mathcal{F} & : & \text{the set of field names}
\end{array}
$$

Figure 2.4: Memory model of the JVM

Indistinguishability

The observational power of the attacker is formally defined by various *indistinguishability* relations $\sim^D$ on each different semantic sub-domains D of the JVM memory.

The manipulation of dynamically allocated values requires to parameterise the indistinguishability relations: two locations/values can be considered indistinguishable at a low level, even if they are different. In this case, we require these values to be in correspondence with respect to a permutation between locations, following the approach proposed by Banerjee and Naumann [22]. Such a permutation models the difference of allocation history between two states. This permutation is defined with the help of a partial bijection β on locations. The partial bijection maps low objects allocated in the heap of the first state to low objects allocated in the heap of the second state. Each indistinguishability relation is hence parameterised by a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$.

Definition 2.1.1 (Value indistinguishability)**.** *Given two values $v_1, v_2 \in \mathcal{V}$, and a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ value indistinguishability $v_1 \sim_\beta^{\mathcal{V}} v_2$ is defined by the clauses:*

$$
null \sim_\beta^{\mathcal{V}} null \qquad \frac{v \in \mathcal{N}}{v \sim_\beta^{\mathcal{V}} v}
$$

$$
\frac{v_1, v_2 \in \mathcal{L} \qquad \beta(v_1) = v_2}{v_1 \quad \sim_\beta^{\mathcal{V}} \quad v_2}
$$

Value indistinguishability is extended point wise to local variable maps (for low variables, *i.e.* local with low security levels according to $\mathbf{k_v}$).

Definition 2.1.2 (Local variables indistinguishability)**.** *Two local variable maps $\rho_1, \rho_2 \in$ LocalVar are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and a type annotation for local variables $\mathbf{k_v}$ if and only if for all $x \in \mathcal{X}$, $\mathbf{k_v}(x) \leq k_{\text{obs}} \Rightarrow \rho_1(x) \sim_\beta^{\mathcal{V}} \rho_2(x)$. We denote this fact: $\rho_1 \sim_{\beta,\mathbf{k_v}}^{\text{LocalVar}} \rho_2$*

The definition of object indistinguishability is similar.

Definition 2.1.3 (Object indistinguishability)**.** *Two objects $o_1, o_2 \in \mathcal{O}$ are indistinguishable with respect to a partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ and a type annotation for fields* ft *if and only if*

- o_1 *and* o_2 *are objects of the same class;*
- *for all fields* $f \in \mathsf{dom}(o_1)$, $\mathsf{ft}(f) \leq k_{\text{obs}} \Rightarrow o_1(f) \sim^{\mathcal{V}}_{\beta} o_2(f)$.

We note this fact: $o_1 \sim^{\mathcal{O}}_{\beta,\mathsf{ft}} o_2$

Note that because o_1 and o_2 are objects of the same class we have $\mathsf{dom}(o_1) = \mathsf{dom}(o_2)$ and $o_2(f)$ is well defined.

Heap indistinguishability requires β to be a bijection between the *low domains* (*i.e.* locations reachable from low local variables) of the considered heaps.

Definition 2.1.4 (Heap indistinguishability)**.** *Two heaps* h_1 *and* h_2 *are indistinguishable with respect to a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, *written* $h_1 \sim^{\mathsf{Heap}}_{\beta} h_2$, *if and only if:*

- β *is a bijection between* $\mathsf{dom}(\beta)$ *and* $\mathsf{rng}(\beta)$*;*
- $\mathsf{dom}(\beta) \subseteq \mathsf{dom}(h_1)$ *and* $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$*;*
- *for every* $l \in \mathsf{dom}(\beta)$, $h_1(l) \sim^{\mathcal{O}}_{\beta,\mathsf{ft}} h_2(\beta(l))$.

Definition 2.1.5 (Final state indistinguishability)**.** *Given a partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$, *a level* $l \in \mathcal{S}$, *and a type annotation* ft *for fields, an output level* $\mathbf{k_r}$ *(for normal termination and termination by an uncaught exception), indistinguishability of two final states is defined by the clauses:*

$$\frac{h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2 \qquad \mathbf{k_r}[\mathrm{n}] \leq k_{\text{obs}} \Rightarrow v_1 \sim^{\mathcal{V}}_{\beta} v_2}{(\langle v_1 \rangle v, h_1) \sim^{\text{FinalState}}_{\beta,\mathsf{ft},\mathbf{k_r}} (\langle v_2 \rangle v, h_2)}$$

$$\frac{h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2 \qquad \mathsf{class}(h_1, l_1) : k \in \mathbf{k_r} \qquad k \leq k_{\text{obs}} \qquad l_1 \sim^{\mathcal{V}}_{\beta} l_2}{(\langle l_1 \rangle e, h_1) \sim^{\text{FinalState}}_{\beta,\mathsf{ft},\mathbf{k_r}} (\langle l_2 \rangle e, h_2)}$$

$$\frac{h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2 \qquad \mathsf{class}(h_1, l_1) : k \in \mathbf{k_r} \qquad k \not\leq k_{\text{obs}}}{(\langle l_1 \rangle e, h_1) \sim^{\text{FinalState}}_{\beta,\mathsf{ft},\mathbf{k_r}} (\langle v_2 \rangle e, h_2)} \qquad \frac{h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2 \qquad \mathsf{class}(h_2, l_2) : k \in \mathbf{k_r} \qquad k \not\leq k_{\text{obs}}}{(\langle v_1 \rangle e, h_1) \sim^{\text{FinalState}}_{\beta,\mathsf{ft},\mathbf{k_r}} (\langle l_2 \rangle e, h_2)}$$

$$\frac{h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2 \qquad \mathsf{class}(h_1, l_1) : k_1 \in \mathbf{k_r} \qquad \mathsf{class}(h_2, l_2) : k_2 \in \mathbf{k_r} \qquad k_1 \not\leq k_{\text{obs}} \qquad k_2 \not\leq k_{\text{obs}}}{(\langle l_1 \rangle e, h_1) \sim^{\text{FinalState}}_{\beta,\mathsf{ft},\mathbf{k_r}} (\langle l_2 \rangle e, h_2)}$$

Formal definition of non-interference

Here we give the semantic definition of non-interfering JVM programs. We rely on the following semantic judgement: $p : s \Downarrow_m fs$ with p a program, m a method of p, s a state and fs a final state. This means that if an execution of method m (taken in a program p) is run on a state s, it terminates on a final state fs. The exact definition of this bigstep semantics is formally defined in the Bicolano project [141].

Definition 2.1.6 (Secure method)**.** *A method* m *(in a program* p*) is said* secure with respect to a signature $\mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}$ *(and a type annotation* ft *for fields) if and only if for all arrays of local variables* $\rho_1, \rho_2 \in \text{LocalVar}$, *for all heaps* $h_1, h_2 \in \mathsf{Heap}$, *for all final states* $(ret_1, h_1'), (ret_2, h_2') \in \text{FinalState}$, *for all partial function* $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that* $\rho_1 \sim^{\text{LocalVar}}_{\beta,\mathbf{k_r}} \rho_2$, $h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2$, $\langle h_1, 0, \rho_1, \varepsilon \rangle \Downarrow_m (ret_1, h_1')$ *and* $\langle h_2, 0, \rho_2, \varepsilon \rangle \Downarrow_m (ret_2, h_2')$ *the following properties hold:*

- *there exists a partial function* $\beta' \in \mathcal{L} \rightharpoonup \mathcal{L}$ *such that* $\beta \subseteq \beta'$ *and final states* (ret_1, h_1') *and* (ret_2, h_2') *are indistinguishable with respect to* β', ft *and output level* $\mathbf{k_r}$*:* $(ret_1, h_1') \sim^{\text{FinalState}}_{\beta',\mathsf{ft},\mathbf{k_r}} (ret_2, h_2')$*;*

- *no modification under level k_h is done on h_1 and h_2: for all field $f \in \mathcal{F}$ such that $k_h \not\leq \mathsf{ft}(f)$,*
  - *for all locations $l \in \mathcal{L}$ such that $h_1(loc).f$ is defined then $h'_1(loc).f$ is defined and equal to $h_1(loc).f$;*
  - *for all locations $l \in \mathcal{L}$ such that $h_2(loc).f$ is defined then $h'_2(loc).f$ is defined and equal to $h_2(loc).f$.*

A method is said secure *if it is secure with respect to all its signatures.*

The set of security signatures of a method m is defined as $\text{Policies}_\Gamma(\text{m}) = \{\ \Gamma_\text{m}[\text{k}] \mid \text{k} \in \mathcal{S}\ \}$. We use it to define the notion of *safe program*.

Definition 2.1.7 (Non-interfering program)**.** *A program is* safe *with respect to a table of method signature Γ if for all its method m, m is secure with respect to all signatures in* $\text{Policies}_\Gamma(\text{m})$.

In order to better understand the notion of partial function β in the definition of secure method, we end this section with a simpler property verified by particular secure methods.

Lemma 2.1.8. *Let m a method (in a program p) that returns a numerical value and is secure with respect to a signature $\mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}$. Let h, h_1, h_2 some heaps, ρ_1, ρ_2 two arrays of local variables such that for all variables x, $\mathsf{ft}(x) \leq k_{\text{obs}}$ implies $\rho_1(x) = \rho_2(x)$ (parameters are equal for low variables) and n_1, n_2 two numeric values such that*

$$\langle h, 0, \rho_1, \varepsilon\rangle \Downarrow_m (n_1, h_1) \qquad \textit{and} \qquad \langle h, 0, \rho_2, \varepsilon\rangle \Downarrow_m (n_2, h_2)$$

Then, if $\mathbf{k_r}[\text{n}] \leq k_{\text{obs}}$, both returned values are equal: $n_1 = n_2$.

Type system soundness

Theorem 2.1.9. *Let p be a program and Γ a table of signatures. If there exists a family of safe cdr results $(region_m)_m$ such that p is well-typed with respect to Γ and $(region_m)_m$ then p is non-interfering with respect to Γ.*

The proof is given in the companion report [28].

2.1.5 Related work

We refer to the survey article of Sabelfeld and Myers [156] for a more complete account of recent developments in language-based security, and only focus on related work that deals with low-level languages, or develops ideas that are relevant to consider in future work.

For convenience, we separate related work between works that deal with typed assembly languages, and higher-order low-level languages and finally with the JVM and Java. Related work about concurrency will be considered in the next sections of this deliverable.

Typed assembly languages

The idea of typing low-level programs and ensuring that compilation preserves typing is not original to information flow, and has been investigated in connection with type-directed compilation. Morrisett, Walker, Crary and Glew [125] develop a typed assembly language (TAL) based on a conventional RISC assembly language, and show that typable programs of System F can be compiled into typable TAL programs.

The study of non-interference for typed assembly languages has been initiated by Medel, Bonelli, and Compagnoni [47], who developed a sound information flow type system for a simple assembly language called SIFTAL. A specificity of SIFTAL is to introduce pseudo-instructions that are used to enforce structured control flow using a stack of continuations; more concretely, the pseudo-instructions are used to push or retrieve linear continuations from the continuation stack. Unlike the stack of call frames that is used in the JVM to handle method calls, the stack of continuations is used for control flow within the body of a method. The use of pseudo-instructions allows to formulate global constraints in the type system, and thus to guarantee non-interference. More recent work by the same authors [117] and by Yu and Islam [180] avoids the use of pseudo-instructions. In addition, Yu and Islam consider a richer assembly language and prove type-preserving compilation for an imperative language with procedures.

Higher-order low-level languages

Zdancewic and Myers [181] develop a sound information flow type system for a CPS calculus that uses linear continuations and prove type-preservation for a linear CPS translation from an imperative higher-order language inspired from SLAM [87] to their CPS language, providing thus one early type-preservation result for information flow. The use of linear continuations in the CPS translation is essential to guarantee type-preserving compilation.

In a similar line of work, Honda and Yoshida [91] develop a sound information flow type system for the π-calculus and prove type-preserving compilation for the Dependency Core Calculus [1] and for an imperative language inspired from Volpano and Smith [173]. Furthermore, they derive soundness of the source type systems from the soundness of the type system for the π-calculus. As in the work of Zdancewic and Myers, linearity is used crucially to ensure that the compilation is type-preserving.

JVM

Lanet *et al.* [45] provide an early study of information flow for the JVM. Their method consists of specifying in the SMV model checker an abstract transition semantics of the JVM that manipulates security levels, and that can be used to verify that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their method is directed towards smart card applications, and thus only covers a sequential fragment of the JVM. While their method has been used successfully to detect information leaks in a case study involving multi-application smartcards, it is not supported by any soundness result. In a series of papers initiating with [42], Bernardeschi and co-workers also propose to use abstract interpretation and model-checking techniques to verify secure information.

In a predecessor to the work presented in this chapter, Barthe, Basu and Rezk [25] propose a sound information flow type system for a simple assembly language that closely resembles the imperative fragment (*i.e.* expressive enough for compiling programs written in a simple imperative language) of the JVM studied in this paper, and show type-preserving compilation for the imperative language and type system of [173]. Later, Barthe and Rezk [30] extend this work to a language with objects and a simplified treatment of exceptions, and Barthe, Naumann and Rezk [27] show type-preserving compilation for a Java-like language with objects and a simplified treatment of exceptions.

Genaim and Spoto [81] have shown how to represent information flow for Java bytecode through boolean functions; the representation allows checking via binary decision diagrams. Their analysis is fully automatic and does not require that methods are annotated with security signatures, but it is less efficient than type checking.

Java

Jif is an extension of Java with information flow types developed by Myers and co-workers. Jif builds upon the decentralized label model and offers a flexible and expressive framework to define information flow policies. The rich set of features supported by Jif has proved useful in realistic case studies such as an implementation of mental poker [**?**], but makes it difficult to prove that the information flow type system is sound.

Banerjee and Naumann [22] develop a sound information flow type system for a fragment of Java with objects and methods. The type system is simpler than Jif:

- due to the absence of certain language features such as exceptions. For example, their return signatures are reduced to a single level, since abnormal termination is not considered.
- by design. For example, there is no mechanism for information release.

The type system has been formally verified in PVS [135], and [168] presents a type inference algorithm that dispenses users of writing all security annotations.

More recently, Hammer, Krinke and Snelting [84] have developed an information flow analysis based on control dependence regions; they use path conditions to achieve precision in their analysis, and to exhibit security leaks if the program is insecure. Their approach is automatic and flow-sensitive, but less efficient than type-based approach.

Both the type systems of [133] and of [22] rely on the assumption that references are opaque, i.e. the only observations that an attacker can make about a reference are those about the object to which it points. However, Hedin and Sands [**?**] have recently observed that the assumption is invalidated by methods from the Java API, and exhibited a Jif program that does not use declassification but leaks information through invoking API methods. Their attack relies on the assumption that the function that allocates new objects on the heap is deterministic; however, this assumption is perfectly reasonable and satisfied by many implementations of the JVM. In addition to demonstrating the attack, Hedin and Sands show how a refined information flow type system can thwart such attacks for a language that allows to cast references as integers. Intuitively, their type system tracks the security level of references as well as the security levels of the fields of the object it points to.

Logical analysis of non-interference for Java

In a different line of work, several authors have investigated the use of program logics to enforce non-interference of Java programs. Darvas and co-workers [63] use dynamic logic to verify information flow policies of Java Card programs. One of their encodings of non-interference is based on the idea of self-composition (see also [**?**]), where the program is composed with a renaming of itself to ensure properties that involve two executions of a program. The idea of self-composition has also been put in practice by Dufay and co-workers [74], who used an extension of the Krakatoa tool [116] with self-composition primitives to verify that data mining programs from the open source repository WEKA adhere to privacy policies cast in terms of information flow. Both [63, 74] are application-oriented and do not attempt to provide a theoretical study of self-composition for Java. In a recent article, Naumann [136] sets out the details of self-composition in presence of a dynamically allocated heap; in short, one main issue tackled by Naumann is the definition of a meaningful notion of "renaming" for the heap.

Independently, Banerjee and his co-workers [15] develop a logic that allows to verify non-interference without resorting to self-composition. The logic, which is tailored to object-oriented languages, handles the

heap using independence assertions inspired from separation logic.

Declassification

Information flow type systems have not found substantial applications in practice, in particular because information flow policies based on non-interference are too rigid and do not authorize information release. In contrast, many applications often release deliberately some amount of sensitive information. Typical examples of deliberate information release include sending an encrypted message through an untrusted network, or allowing confidential information to be used in statistics over large databases. In a recent survey [159], A. Sabelfeld and D. Sands provide an overview of relaxed policies that allow for some amount of information release, and a classification along several dimensions, for example who releases the information, and what information is released. A first solution to an integrated control for multiple dimensions of declassification is developed in [146, 111].

2.2 Security Types for Multithreaded Bytecode

2.2.1 Introduction

Information flow for multithreaded low-level programs has not been addressed so far. It is especially concerning because multithreaded bytecode is ubiquitous in mobile code scenarios. For example, multithreading is used for preventing screen lock-up in mobile applications [109]. In general, creating a new thread for long and/or potentially blocking computation, such as establishing a network connection, is a much recommended pattern [101].

This section is the first to propose a framework for enforcing secure information flow for multithreaded low-level programs. We present an approach for deriving security-type systems that provably guarantee noninterference. On the code consumer side, these type systems can be used for checking the security of programs before running them.

Our solution goes beyond guarantees offered by security-type checking to code consumers. To this end, we have developed a framework for security-type preserving compilation, which allows code producers to derive security types for low-level programs from security types for source programs. This makes our solution practical for the scenario of untrusted mobile code. Moreover, even if the code is trusted (and perhaps even immobile), compilers are often too complex to be a part of the trusted computing base. Security-type preserving compilation removes the need to trust the compiler, because the type annotations of compiled programs can be checked directly at bytecode level.

The single most attractive feature of our framework is that security is guaranteed by source type systems that are no more restrictive than ones for sequential programs. This might be counterintuitive: there are covert channels in the presence of threads, such as internal timing channels [174], that do not arise in a sequential setting. Indeed, special primitives for interacting with the scheduler have been designed (e.g., [150]) in order to control these channels. The pinnacle of our framework is that such primitives are automatically introduced in the compilation phase. This means that source-language programmers do not have to know about their existence and that there are no restrictions on dynamic thread creation at the source level. At the target level, the prevention of internal timing leaks does not introduce unexpected behaviours: the effect of interacting with the scheduler may only result in disallowing certain interleavings. Note that disallowing interleavings may, in general, affect the liveness properties of a program. Such a trade-off between between liveness and security is shared with other approaches (e.g., [164, 174, 162, 163, 150]).

For an example of an internal timing leak, consider a simple two-threaded source-level program, where *hi* is a sensitive (high) and *lo* is a public (low) variable:

$$\texttt{if } hi \ \{\texttt{sleep}(100)\}; lo := 1 \parallel \texttt{sleep}(50); lo := 0$$

If *hi* is originally non-zero, the last command to assign to *lo* is likely to be $lo := 1$. If *hi* is zero, the last command to assign to *lo* is likely to be $lo := 0$. Hence, this program is likely to leak information about *hi* into *lo*. In fact, all of *hi* can be leaked into *lo* via the internal timing channel, if the timing difference is magnified by a loop (see, e.g., [149]).

In order for the timing difference of the thread that branches on *hi* not to make a difference in the interleaving of the assignments to *lo*, we need to ensure that the scheduler treats the first thread as "hidden" from the second thread: the second thread should not be scheduled until the first thread reaches the junction point of the `if`. We will show that the compiler enforces such a discipline for the target code so that the compilation of such source programs as above is free of internal timing leaks.

Our work benefits from modularity, which is three-fold. First, the framework has the ability to modularly extend sequential semantics. This grants us with language-independence from the sequential part. Further, the framework allows modular extensions of sequential security type systems. Finally, security type preserving compilation is also a modular extension of the sequential counterpart.

To illustrate the applicability of the framework, we instantiate it with some scheduler examples. These examples clarify what is expected of a scheduler to prevent internal timing leaks. Also, we give an instantiation of the source language with a simple imperative language, as well as an instantiation of the target language with a simple assembly language that features an operand stack, conditions, and jumps. As we will discuss, these instantiations are for illustration only: we expect our results to apply to languages close to Java and Java bytecode, respectively.

Our approach pushes the feasibility of replacing trust assumptions by type checking for mobile-code security one step further. It is especially encouraging that we inherit the main benefit of recent results on enforcing secure information flow by security-type systems [29]: compatibility with bytecode verification, and no need to trust the compiler.

2.2.2 Syntax and semantics of multithreaded programs

This section sets the scene by defining the syntax and semantics for multithreaded programs. We introduce the notion of secure schedulers that help dealing with covert channels in the presence of multithreading.

Syntax and program structure. Assume we have a set Thread of thread identifiers, a partially ordered set Level of security levels, a set $\mathsf{LocState}$ of local states and a set $\mathsf{GMemory}$ of global memories. The definition of programs is parameterised by a set of sequential instructions SeqIns. The set of all instructions extends SeqIns by a dynamic thread creation primitive $\mathsf{start}\ pc$ that spawns a new thread with a start instruction at program point *pc*.

Definition 2.2.1 (Program)**.** *A program P consists of a set of program points $\mathcal{P}$, with a distinguished entry point* 1 *and a distinguished exit point* exit*, and an instruction map* $\mathsf{insmap}_P : \mathcal{P} \setminus \{\text{exit}\} \to \mathsf{Ins}$*, where* $\mathsf{Ins} = \mathsf{SeqIns} \cup \{\mathsf{start}\ pc\}$ *with* $pc \in \mathcal{P} \setminus \{\text{exit}\}$*. We sometimes write $P[i]$ instead of* $\mathsf{insmap}_P\ i$*.*

Each program has an associated successor relation $\mapsto \subseteq \mathcal{P} \times \mathcal{P}$ (see Section 2.1.2). The successor relation describes possible successor instructions in an execution. We assume that exit is the only program point without successors, and that any program point i s.t. $P[i] = \mathsf{start}\ pc$ is not branching, and has a single

successor, denoted by $i+1$ (if it exists); in particular, we do not require that $i \mapsto pc$. As common, we let $\mapsto^\star$ denote the reflexive and transitive closure of the relation $\mapsto$ (similar notation is used for other relations).

Definition 2.2.2 (Initial program points)**.** *The set $\mathcal{P}_{init}$ of initial program points is defined as:* $\{i \in \mathcal{P} \mid \exists j \in \mathcal{P},\ P[j] = \mathsf{start}\ i\} \cup \{1\}$.

We assume the attacker level $k \in \mathsf{Level}$ partitions all elements of Level into *low* and *high* elements. Low elements are no more sensitive than k: an element ℓ is low if $\ell \leq k$. All other elements (including incomparable ones) are high. We assume that the set of high elements is not empty. This partition reduces the set Level to a two-element set $\{low, high\}$, where $low < high$, which we will adopt without loss of generality.

Programs come equipped with a *security environment* [25] that assigns a security level to each program point and is used to prevent *implicit flows* [65]. The security environment is also used by the scheduler to select the thread to execute.

Definition 2.2.3 (Security environment, low, high, and always high program points)**.**

1. *A security environment is a function $se : \mathcal{P} \to \mathsf{Level}$.*
2. *A program point $i \in \mathcal{P}$ is* low*, written $L(i)$, if $se(i) = low$;* high*, written $H(i)$, if $se(i) = high$; and* always high*, written $AH(i)$, if $se(j) = high$ for all points j such that $i \mapsto^\star j$.*

Semantics. The operational semantics for multithreaded programs is built from an operational semantics for sequential programs and a scheduling function that picks the thread to be executed among the currently active threads. The scheduling function takes as parameters the current state, the execution history, and the security environment.

Definition 2.2.4 (State)**.**

1. *The set* SeqState *of sequential states is a product* LocState $\times$ GMemory *of the local state* LocState *and global memory* GMemory *sets.*
2. *The set* ConcState *of concurrent states is a product* (Thread $\rightharpoonup$ LocState) $\times$ GMemory *of the partial-function space* (Thread $\rightharpoonup$ LocState)*, mapping thread identifiers to local states, and the set* GMemory *of global memories.*

It is convenient to use accessors to extract components from states: we use $s \cdot \mathsf{lst}$ and $s \cdot \mathsf{gmem}$ to denote the first and second components of a state s. Then, we use $s \cdot \mathsf{act}$ to denote the set of active threads, i.e., $s \cdot \mathsf{act} = \mathsf{Dom}(s \cdot \mathsf{lst})$. We sometimes write $s(tid)$ instead of $s \cdot \mathsf{lst}(tid)$ for $tid \in s \cdot \mathsf{act}$. Furthermore, we assume given an accessor pc that extracts the program counter for a given thread from a local state.

We follow a concurrency model [150] that lets the scheduler distinguish between different types of threads. A thread is *low* (resp., *high*) if the security environment marks its program counter as low (resp., high). A high thread is *always high* if the program point corresponding to the program counter is always high. A high thread is *hidden* if it is high but not always high. (Intuitively, the thread is hidden in the sense that the scheduler will, independently from the hidden thread, pick the following low threads.) Formally, we have the following definitions:

$$\begin{aligned} s \cdot \mathsf{lowT} &= \{tid \in s \cdot \mathsf{act} \mid L(s \cdot \mathsf{pc}(tid))\} \\ s \cdot \mathsf{highT} &= \{tid \in s \cdot \mathsf{act} \mid H(s \cdot \mathsf{pc}(tid))\} \\ s \cdot \mathsf{ahighT} &= \{tid \in s \cdot \mathsf{act} \mid AH(s \cdot \mathsf{pc}(tid))\} \\ s \cdot \mathsf{hidT} &= \{tid \in s \cdot \mathsf{act} \mid H(s \cdot \mathsf{pc}(tid)) \wedge \neg AH(s \cdot \mathsf{pc}(tid))\} \end{aligned}$$

A scheduler treats different classes of threads differently. To see what guarantees are provided by the scheduler, it is helpful to foresee what discipline a type system would enforce for each kind of threads. From the point of view of the type system, a low thread becomes high while being inside of a branch of a conditional (or a body of a loop) with a high guard. Until reaching the respective junction point, the thread may not have any low side effects. In addition, until reaching the respective junction point, the high thread must be hidden by the scheduler: no low threads may be scheduled while the hidden thread is alive. This prevents the timing of the hidden thread from affecting the interleaving of low side effects in low threads. In addition, there are threads that are spawned inside of a branch of a conditional (or a body of a loop) with a high guard. These threads are always high: they may not have any low side effects. On the other hand, such threads do not have to be hidden in the same way: they can be interleaved with both low and high threads. Recall the example from Section 2.2.1. The intention is that the scheduler treats the first thread (which is high while it is inside the branch) as "hidden" from the second (low) thread: the second thread should not be scheduled until the first thread reaches the junction point of the `if`.

We proceed to defining computation history and secure schedulers, which operate on histories as parameters.

Definition 2.2.5 (History)**.**

1. *A history is a list of pairs* (tid, ℓ) *where* $tid \in \mathsf{Thread}$ *and* $\ell \in \mathsf{Level}$*. We denote the empty history by* ϵ^{hist}*.*
2. *Two histories* h *and* h' *are indistinguishable, written* $h \overset{\mathsf{hist}}{\sim} h'$*, if* $h|_{low} = h'|_{low}$*, where* $h|_{low}$ *is obtained from* h *by projecting out pairs with the high level in the second component.*

We denote the set of histories by $\mathsf{History}$. We now turn to the definition of a secure scheduler. The definition below is of a more algebraic nature than that of [150], but captures the same intuition, namely that a secure scheduler: i) always picks an active thread; ii) chooses a high thread whenever there is one hidden thread; and iii) only uses the names and levels of low and the low part of histories to pick a low thread.

Definition 2.2.6 (Secure scheduler)**.** *A secure scheduler is a function* $\mathsf{pickt} : \mathsf{ConcState} \times \mathsf{History} \rightharpoonup \mathsf{Thread}$*, subject to the following constraints, where* $s, s' \in \mathsf{ConcState}$ *and* $h, h' \in \mathsf{History}$*:*

1. *for every* s *such that* $s \cdot \mathsf{lowT} \cup s \cdot \mathsf{highT} \neq \emptyset$*,* $\mathsf{pickt}(s, h)$ *is defined, and* $\mathsf{pickt}(s, h) \in s \cdot \mathsf{act}$*;*
2. *if* $s \cdot \mathsf{hidT} \neq \emptyset$*, then* $\mathsf{pickt}(s, h) \in s \cdot \mathsf{highT}$*; and*
3. *if* $h \overset{\mathsf{hist}}{\sim} h'$ *and* $s \cdot \mathsf{lowT} = s' \cdot \mathsf{lowT}$*, then* $\langle \mathsf{pickt}(s, h), \ell \rangle :: h \overset{\mathsf{hist}}{\sim} \langle \mathsf{pickt}(s', h'), \ell' \rangle :: h'$*, where* $\ell = se(s \cdot pc(\mathsf{pickt}(s, h)))$ *and* $\ell' = se(s' \cdot pc(\mathsf{pickt}(s', h')))$*.*

Example 1. *Consider a round-robin policy:* $\mathsf{pickt}(s, h) = rr(AT, last(h))$*, where* $AT = s \cdot \mathsf{act}$*, and the partial function* $last(h)$ *returns the identity of the most recently picked thread recorded in* h *(if it exists). Given a set of thread ids, an auxiliary function* rr *returns the next thread id to pick according to a round-robin policy. This scheduler is insecure because low threads can be scheduled even if a hidden thread is present, which violates req.* 2 *above.*

Example 2. *An example of a secure round-robin scheduler is defined below. The scheduler takes turns in picking high and low threads.*

$$\mathsf{pickt}(s, h) = \begin{cases} rr(AT_L, last_L(h)), & \begin{array}{l} \text{if } h = \epsilon^{\mathsf{hist}} \text{ or} \\ h = (\mathsf{tid}, L) \cdot h' \text{ and } AT_H = \emptyset \text{ and } AT_L \neq \emptyset \text{ or} \\ h = (\mathsf{tid}, H) \cdot h' \text{ and } \mathsf{hidT} = \emptyset \text{ and } AT_L \neq \emptyset \end{array} \\ rr(AT_H, last_H(h)), & \begin{array}{l} \text{if } \mathsf{hidT} \neq \emptyset \text{ or} \\ h = (\mathsf{tid}, H) \cdot h' \text{ and } AT_L = \emptyset \text{ and } AT_H \neq \emptyset \text{ or} \\ h = (\mathsf{tid}, L) \cdot h' \text{ and } AT_H \neq \emptyset \end{array} \end{cases}$$

$$\frac{\begin{array}{c}\mathsf{pickt}(s,h) = ctid \quad s\cdot pc(ctid) = i \quad P[i] \in \mathsf{SeqIns} \\ \langle s(ctid), s\cdot\mathsf{gmem}\rangle \rightsquigarrow_{\mathrm{seq}} \sigma,\mu \quad \sigma\cdot\mathsf{pc} \neq \mathrm{exit}\end{array}}{s,h \rightsquigarrow_{\mathrm{conc}} s\cdot[\mathsf{lst}(ctid) := \sigma, \mathsf{gmem} := \mu], \langle ctid, se(i)\rangle :: h}$$

$$\frac{\begin{array}{c}\mathsf{pickt}(s,h) = ctid \quad s\cdot\mathsf{pc}(ctid) = i \quad P[i] \in \mathsf{SeqIns} \\ \langle s(ctid), s\cdot\mathsf{gmem}\rangle \rightsquigarrow_{\mathrm{seq}} \sigma,\mu \quad \sigma\cdot\mathsf{pc} = \mathrm{exit}\end{array}}{s,h \rightsquigarrow_{\mathrm{conc}} s\cdot[\mathsf{lst} := \mathsf{lst}\setminus ctid, \mathsf{gmem} := \mu], \langle ctid, se(i)\rangle :: h}$$

$$\frac{\begin{array}{c}\mathsf{pickt}(s,h) = ctid \quad s\cdot\mathsf{pc}(ctid) = i \quad P[i] = \mathsf{start}\ pc \\ \mathsf{fresht}_{se(i)}(s) = ntid \quad s(ctid)\cdot[\mathsf{pc} := i+1] = \sigma'\end{array}}{s,h \rightsquigarrow_{\mathrm{conc}} s\cdot[\mathsf{lst}(ctid) := \sigma', \mathsf{lst}(ntid) := \lambda_{\mathrm{init}}(pc)], \langle ctid, se(i)\rangle :: h}$$

Figure 2.5: Semantics of multithreaded programs

$$\frac{P[i] \in \mathsf{SeqIns} \qquad se, i \vdash_{\mathrm{seq}} \mathrm{S} \Rightarrow \mathrm{T}}{se, i \vdash \mathrm{S} \Rightarrow \mathrm{T}} \qquad \frac{P[i] = \mathsf{start}\ pc \quad se(i) \leq se(pc)}{se, i \vdash \mathrm{S} \Rightarrow \mathrm{S}}$$

Figure 2.6: Typing rules

We assume that AT_L and AT_H are functions of s that extract the set of identifiers of low and high threads, respectively, and the partial function $last_\ell$ returns the identity of the most recently picked thread at level ℓ recorded in h, if it exists. The scheduler may only pick active threads (cf. req. 1). In addition to the alternation between high and low threads, the scheduler may only pick a low thread if there are no hidden threads (cf. req. 2). The separation into high and low threads ensures that for low-equivalent histories, the observable choices of the scheduler are the same (cf. req. 3).

To define the execution of multithreaded programs, we assume given a (deterministic) sequential execution relation $\rightsquigarrow_{\mathrm{seq}} \subseteq \mathsf{SeqState} \times \mathsf{SeqState}$ that takes as input a current state and returns a new state, provided the current instruction is sequential.

We assume given a function $\lambda_{\mathrm{init}} : \mathcal{P} \to \mathsf{LocState}$ that takes a program point and produces an initial state with program pointer pointing to pc. We also assume given a family of functions fresht_ℓ that takes as input a set of thread identifiers and generates a new thread identifier at level ℓ. We assume that the ranges of fresht_ℓ and $\mathsf{fresht}_{\ell'}$ are disjoint whenever $\ell \neq \ell'$. We sometimes use fresht_ℓ as a function from states to Thread.

Definition 2.2.7 (Multithreaded execution)**.** *One step execution* $\rightsquigarrow_{\mathrm{conc}} \subseteq (\mathsf{ConcState}\times\mathsf{History})\times(\mathsf{ConcState}\times\mathsf{History})$ *is defined by the rules of Figure 2.5. We write $s,h \rightsquigarrow_{\mathrm{conc}} s',h'$ when executing s with history h leads to state s' and history h'.*

The first two rules of Figure 2.5 correspond to non-terminating and terminating sequential steps. In the case of termination, the current thread is removed from the domain of lst. The last rule describes dynamic thread creation caused by the instruction $\mathsf{start}\ pc$. A new thread receives a fresh name $ntid$ from $\mathsf{fresht}_{se(i)}$ where $se(i)$ records the security environment at the point of creation. This thread is added to the pool of threads under the name $ntid$. All rules update the history with the current thread id and the security environment of the current instruction. The evaluation semantics of programs can be derived from the small-step semantics in the usual way. We let *main* be the identity of the main thread.

Definition 2.2.8 (Evaluation semantics)**.** *The evaluation relation* $\Downarrow_{\mathrm{conc}} \subseteq (\mathsf{ConcState}\times\mathsf{History})\times\mathsf{GMemory}$ *is defined by the clause $s,h \Downarrow_{\mathrm{conc}} \mu$ iff $\exists s',h'.\ s,h \rightsquigarrow^\star_{\mathrm{conc}} s',h' \wedge s'\cdot\mathsf{act} = \emptyset \wedge s'\cdot\mathsf{gmem} = \mu$. We write $P,\mu \Downarrow_{\mathrm{conc}} \mu'$ as a shorthand for $\langle f,\mu\rangle, \epsilon^{\mathsf{hist}} \Downarrow_{\mathrm{conc}} \mu'$, where f is the function $\{\langle main, \lambda_{\mathrm{init}}(1)\rangle\}$.*

2.2.3 Security policy

Noninterference is defined relative to a notion of indistinguishability between global memories. For the purpose of this section, it is not necessary to specify the definition of memory indistinguishability.

Definition 2.2.9 (Noninterfering program)**.** *Let $\sim_g$ be an indistinguishability relation on global memories. A program P is* noninterfering *if for all memories $\mu_1, \mu_2, \mu_1', \mu_2'$:*

$$\mu_1 \sim_g \mu_2 \text{ and } P, \mu_1 \Downarrow \mu_1' \text{ and } P, \mu_2 \Downarrow \mu_2' \text{ implies } \mu_1' \sim_g \mu_2'$$

2.2.4 Type system

This section introduces a type system for multithreaded programs as an extension of a type system for noninterference for sequential programs. In Section 2.2.5, we show that the type system is sound for multithreaded programs, in that it enforces the noninterference property defined in the previous section. In Section 2.2.6, we instantiate the framework to a simple assembly language.

Assumptions on type system for sequential programs. We assume given a set LType of local types for typing local states, with a distinguished local type $\mathrm{T}_{\mathrm{init}}$ to type initial states, and a partial order $\leq$ on local types. Typing judgements in the sequential type system are of the form $se, i \vdash_{\mathrm{seq}} \mathrm{S} \Rightarrow \mathrm{T}$, where se is a security environment, i is a program point in program P, and S and T are local types.

Typing rules are used to establish a notion of typable program [7]; typable programs are assumed to satisfy several properties that are formulated precisely in Section 2.2.5.

Type system for multithreaded programs. The typing rules for the concurrent type system have the same form as those of the sequential type system and are given in Figure 2.6.

Definition 2.2.10 (Typable multithreaded program)**.** *A concurrent program P is typable w.r.t. type $\mathcal{S} : \mathcal{P} \to$ LType and security environment se, written $se, \mathcal{S} \vdash P$, if*

1. *$\mathcal{S}_i = \mathrm{T}_{\mathrm{init}}$ for all initial program points i of P (initial program point of main threads or spawn threads); and*
2. *for all $i \in \mathcal{P}$ and $j \in \mathcal{P}$: $i \mapsto j$ implies that there exists $\mathrm{S} \in$ LType such that $se, i \vdash \mathcal{S}_i \Rightarrow \mathrm{S}$ and $\mathcal{S}_j \leq \mathrm{S}$.*

2.2.5 Soundness

The purpose of this section is to prove, under sufficient hypotheses on the sequential type system and assuming that the scheduler is secure, that typable programs are noninterfering. Formally, we want to prove the following theorem:

Theorem 2.2.11. *If the scheduler is secure and $se, \mathcal{S} \vdash P$, then P is noninterfering.*

Throughout this section, we assume that P is a typable program, i.e., $se, \mathcal{S} \vdash P$, and that the scheduler is secure. Moreover, we state some general hypotheses that are used in the soundness proofs. We revisit these hypotheses in Section 2.2.6 and show how they can be fulfilled.

[7] The notion of typable sequential program is a particular case of typable multithreaded program.

State equivalence. In order to prove noninterference, we rely on a notion of state equivalence. The definition is modular, in that it is derived from an equivalence between global memories $\sim_g$ and a partial equivalence relation $\sim_l$ between local states. (Intuitively, partial equivalence relations on local and global memories represent the observational power of the adversary.) In comparison to [29], equivalence between local states (operand stacks and program counters for the JVM) is not indexed by local types, since these can be retrieved from the program counter and the global type of the program.

Definition 2.2.12 (State equivalence)**.** *Two concurrent states s and t are:*

1. *equivalent w.r.t. local states, written $s \overset{\mathsf{lmem}}{\sim} t$, iff $s \cdot \mathsf{lowT} = t \cdot \mathsf{lowT}$ and for every $tid \in s \cdot \mathsf{lowT}$, we have $s(tid) \sim_l t(tid)$.*
2. *equivalent w.r.t. global memories, written $s \overset{\mathsf{gmem}}{\sim} t$, iff $s \cdot \mathsf{gmem} \sim_g t \cdot \mathsf{gmem}$.*
3. *equivalent, written $s \sim t$, iff $s \overset{\mathsf{gmem}}{\sim} t$ and $s \overset{\mathsf{lmem}}{\sim} t$.*

In order to carry out the proofs, we also need a notion of program counter equivalence between two states.

Definition 2.2.13. *Two states s and s' are pc-equivalent, written, $s \overset{pc}{\sim} s'$ iff $s \cdot \mathsf{lowT} = t \cdot \mathsf{lowT}$ and for every $tid \in s \cdot \mathsf{lowT}$, we have $s \cdot \mathsf{pc}(tid) = t \cdot \mathsf{pc}(tid)$.*

Unwinding lemmas. In this section, we formulate unwinding hypotheses for sequential instructions and extend them to a concurrent setting. Two kinds of unwinding statements are considered: a *locally respects unwinding result*, which involves two executions and is used to deal with execution in low environments, and a *step consistent unwinding result*, which involves one execution and is used to deal with execution in high environments. From now on, we refer to local states and global memories as λ and μ, respectively.

Hypothesis 2.2.14 (Sequential locally respects unwinding)**.** *Assume $\lambda_1 \sim_l \lambda_2$ and $\mu_1 \sim_g \mu_2$ and $\lambda_1 \cdot \mathsf{pc} = \lambda_2 \cdot \mathsf{pc}$. If $\langle \lambda_1, \mu_1 \rangle \rightsquigarrow_{\text{seq}} \langle \lambda_1', \mu_1' \rangle$ and $\langle \lambda_2, \mu_2 \rangle \rightsquigarrow_{\text{seq}} \langle \lambda_2', \mu_2' \rangle$, then $\lambda_1' \sim_l \lambda_2'$ and $\mu_1' \sim_g \mu_2'$.*

In addition, we also need a hypothesis on the indistinguishability of initial local states.

Hypothesis 2.2.15 (Equivalence of local initial states)**.** *For every initial program point i, we have $\lambda_{\text{init}}(i) \sim_l \lambda_{\text{init}}(i)$.*

We now extend the unwinding statement to concurrent states; note that the hypothesis $s' \cdot \mathsf{lowT} = t' \cdot \mathsf{lowT}$ is required for the lemma to hold. This excludes the case of a thread becoming hidden in an execution and not another (i.e., a high while loop).

Lemma 2.2.16 (Concurrent locally respects unwinding)**.** *Assume $s \sim t$ and $h_s \overset{\mathsf{hist}}{\sim} h_t$ and $\mathsf{pickt}(s, h_s) = \mathsf{pickt}(t, h_t) = ctid$ and $s \cdot pc(ctid) = t \cdot pc(ctid)$. If $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ and $t, h_t \rightsquigarrow_{\text{conc}} t', h_{t'}$, and $s' \cdot \mathsf{lowT} = t' \cdot \mathsf{lowT}$, then $s' \sim t'$ and $h_{s'} \overset{\mathsf{hist}}{\sim} h_{t'}$.*

We now turn to the second, so-called step consistent, unwinding lemma. The lemma relies on the hypothesis that the current local memory is high, i.e., invisible by the attacker. Formally, highness is captured by a predicate $High^{\mathsf{lmem}}(\lambda)$ where λ is a local state.

Hypothesis 2.2.17 (Sequential step consistent unwinding)**.** *Assume $\lambda_1 \sim_l \lambda_2$ and $\mu_1 \sim_g \mu_2$. Let $\lambda_1 \cdot \mathsf{pc} = i$. If $\langle \lambda_1, \mu_1 \rangle \rightsquigarrow_{\text{seq}} \langle \lambda_1', \mu_1' \rangle$ and $High^{\mathsf{lmem}}(\lambda_1)$ and $H(i)$, then $\lambda_1' \sim_l \lambda_2$ and $\mu_1' \sim_g \mu_2$.*

Lemma 2.2.18 (Concurrent step consistent unwinding)**.** *Assume* $s \sim t$ *and* $h_s \overset{\mathsf{hist}}{\sim} h_t$ *and* $\mathsf{pickt}(s,h) = ctid$ *and* $s \cdot \mathsf{pc}(ctid) = i$ *and* $High^{\mathsf{lmem}}(s(ctid))$ *and* $H(i)$*. If* $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$ *and* $s' \cdot \mathsf{lowT} = t \cdot \mathsf{lowT}$*, then* $s' \sim t$ *and* $h_{s'} \overset{\mathsf{hist}}{\sim} h_t$*.*

The proofs of the unwinding lemmas are by a case analysis on the semantics of concurrent programs.

The next function. The soundness proof relies on the existence of a function next that satisfies several properties. Intuitively, next computes for any high program point its minimal observable successor, i.e., the first program point with a low security level reachable from it. If executing the instruction at program point i can result in a hidden thread (high if or high while), then $\mathsf{next}(i)$ is the first program point such that $i \mapsto^\star \mathsf{next}(i)$ and the thread becomes visible again.

Hypothesis 2.2.19 (Existence of next function)**.** *There exists a function* $\mathsf{next} : \mathcal{P} \rightharpoonup \mathcal{P}$ *such that the* next *properties (NeP) hold:*

NePd) $\mathsf{Dom}(\mathsf{next}) = \{i \in \mathcal{P} | H(i) \wedge \exists j \in \mathcal{P}.\ i \mapsto^\star j \wedge \neg H(j)\}$
NeP1) $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = \mathsf{next}(j)$
NeP2) $i \in \mathsf{Dom}(\mathsf{next}) \wedge j \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \Rightarrow \mathsf{next}(i) = j$
NeP3) $j, k \in \mathsf{Dom}(\mathsf{next}) \wedge i \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \wedge j \neq k \Rightarrow \mathsf{next}(j) = \mathsf{next}(k)$
NeP4) $i, j \in \mathsf{Dom}(\mathsf{next}) \wedge k \notin \mathsf{Dom}(\mathsf{next}) \wedge i \mapsto j \wedge i \mapsto k \Rightarrow \mathsf{next}(j) = k$

Intuitively, properties **NeP1**, **NeP2**, and **NeP3** ensure that the next of instructions within an outermost high conditional statement coincides with the junction point of the conditional; in addition, properties **NeP1**, **NeP2**, and **NeP4** ensure that the next of instructions within an outermost high loop coincides with the exit point of the loop.

In addition to the above assumptions, we also need another hypothesis that relates the domain of next to the operational semantics of programs. In essence, the hypothesis states that, under the assumptions of the concurrent locally respects unwinding lemma, either the executed instruction is a low instruction, in which case the program counter of the active thread remains equal after one step of execution, or that the executed instruction is a high instruction, in which case the active thread is hidden in one execution (high loop) or both (high conditional).

Hypothesis 2.2.20 (Preservation of pc equality)**.** *Assume* $s \sim t$*;* $\mathsf{pickt}(s, h_s) = \mathsf{pickt}(t, h_t) = ctid$*;* $s(ctid) \cdot \mathsf{pc} = t(ctid) \cdot \mathsf{pc}$*;* $s, h_s \rightsquigarrow_{\text{conc}} s', h_{s'}$*; and* $t, h_t \rightsquigarrow_{\text{conc}} t', h_{t'}$*. Then,* $s'(ctid) \cdot \mathsf{pc} = t'(ctid) \cdot \mathsf{pc}$*; or* $s'(ctid) \cdot \mathsf{pc} \in \mathsf{Dom}(\mathsf{next})$*; or* $t'(ctid) \cdot \mathsf{pc} \in \mathsf{Dom}(\mathsf{next})$*.*

The final hypothesis is about visibility by the attacker:

Hypothesis 2.2.21 (High hypotheses)**.**

1. *For every program point* i*, we have* $High^{\mathsf{lmem}}(\lambda_{\text{init}}(i))$*.*
2. *If* $\langle \lambda, \mu \rangle \rightsquigarrow_{\text{seq}} \langle \lambda', \mu' \rangle$ *and* $High^{\mathsf{lmem}}(\lambda)$ *and* $H(\lambda \cdot \mathsf{pc})$ *then* $High^{\mathsf{lmem}}(\lambda')$*.*
3. *If* $High^{\mathsf{lmem}}(\lambda_1)$ *and* $High^{\mathsf{lmem}}(\lambda_2)$ *then* $\lambda_1 \sim_l \lambda_2$*.*

Theorem 2.2.11 follows from the hypotheses above. For the proof details, we refer to the technical report [32].

$$e ::= x \mid n \mid e\ op\ e \qquad c ::= x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{fork}(c)$$

$instr$	::=	$\mathsf{binop}\ op$	binary operation on stack
	|	$\mathsf{push}\ n$	push value on top of stack
	|	$\mathsf{load}\ x$	load value of x on stack
	|	$\mathsf{store}\ x$	store top of stack in variable x
	|	$\mathsf{ifeq}\ j$	conditional jump
	|	$\mathsf{goto}\ j$	unconditional jump
	|	$\mathsf{start}\ j$	creation of a thread

where $op \in \{+, -, \times, /\}$, $n \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{P}$.

Figure 2.7: Source and target language

$$\frac{P[i] = \mathsf{push}\ n}{se, i \vdash_{\text{seq}} st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{se, i \vdash_{\text{seq}} k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2 \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \Gamma(x)}{se, i \vdash_{\text{seq}} k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{load}\ x}{se, i \vdash_{\text{seq}} st \Rightarrow (\Gamma(x) \sqcup se(i)) :: st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{se, i \vdash_{\text{seq}} st \Rightarrow st} \qquad \frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{reg}(i),\ k \leq se(j')}{se, i \vdash_{\text{seq}} k :: st \Rightarrow \mathsf{lift}_k(st)}$$

Figure 2.8: Transfer rules

2.2.6 Instantiation

In this section, we apply our main results to a simple assembly language with conditional jumps and dynamic thread creation. We present the assembly language with a semantics and a type system for noninterference but without considering concurrent primitives and plug these definitions into the framework for multithreading. Then, we present a compilation function from a simple while-language with dynamic thread creation into assembly code. The source and target languages are defined in Figure 2.7. The compilation function allows us to easily define control dependence regions and junction points in the target code. Function next is then defined using that information. Moreover, we prove that the obtained definition of next satisfies the properties required in Section 2.2.5. Finally, we conclude with a discussion about how a similar instantiation can be done for the JVM.

Sequential part of the language. The instantiation requires us to define the semantics and a type system to enforce noninterference for the sequential primitives in the language. On the semantics side, we assume that a local state is a pair $\langle os, pc\rangle$ where os is an operand stack, i.e., a stack of values, and pc is a program counter, whereas a global state μ is a map from variables to values. The operational semantics is standard and therefore we omit it. We also define $\lambda_{\text{init}}(pc)$ to be the local state $\langle \epsilon, pc\rangle$, where ϵ is the empty operand stack.

The enforcement mechanism consists of local types which are stacks of security levels, i.e., $\mathsf{LType} = \mathsf{Stack}(\mathsf{Level})$; we let $\top_{\text{init}}$ be the empty stack of security levels. Typing rules are summarised in Figure 2.8, where $\mathsf{lift}_k(st)$ denotes the point-wise extension of $\lambda k'.\ k \sqcup k'$ to stacks of security levels, and $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ denotes the region of branching points. We express the chosen security policy by assigning a security level $\Gamma(x)$ to each variable x.

Similarly to Section 2.1.3, the soundness of the transfer rules relies on some assumptions about control dependence regions in programs. Essentially, these regions represent an over-approximation of the range of branching points. This concept is formally introduced by the functions $\mathsf{reg} : \mathcal{P} \rightharpoonup \wp(\mathcal{P})$ and $\mathsf{jun} : \mathcal{P} \rightharpoonup \mathcal{P}$, which respectively compute the control dependence region and the junction point for a given instruction. Both functions need to satisfy some properties in order to guarantee noninterference in typable programs. These properties are known as SOAP properties and where already defined in Section 2.1.3. In this section,

$$\mathcal{E}(x) = \mathsf{load}\ x \qquad \mathcal{E}(n) = \mathsf{push}\ n \qquad \mathcal{E}(e\ op\ e') = \mathcal{E}(e) :: \mathcal{E}(e') :: \mathsf{binop}\ op$$

$$\begin{array}{rcl}
\mathcal{S}(x := e,\ T) & = & (\mathcal{E}(e) :: \underline{\mathsf{store}}\ x,\ T) \\
\mathcal{S}(c_1;\ c_2,\ T) & = & let\ \ (lc_1,\ T_1) = \mathcal{S}(c_1,\ T); (lc_2,\ T_2) = \mathcal{S}(c_2,\ T_1); \\
 & & in\ \ (lc_1 :: lc_2,\ T_2) \\
\mathcal{S}(\mathsf{while}\ e\ \mathsf{do}\ c,\ T) & = & let\ \ le = \mathcal{E}(e); (lc,\ T') = \mathcal{S}(c,\ T); \\
 & & in\ \ (\mathsf{goto}\ (pc + \#lc + 1) :: lc :: le :: \underline{\mathsf{ifeq}\ (pc - \#lc - \#le)}, \\
 & & \quad T') \\
\mathcal{S}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\ T) & = & let\ \ le = \mathcal{E}(e); (lc_1,\ T_1) = \mathcal{S}(c_1,\ T); (lc_2,\ T_2) = \mathcal{S}(c_2,\ T_1); \\
 & & in\ \ (le :: \underline{\mathsf{ifeq}\ (pc + \#lc_2 + 2)} :: lc_2 :: \mathsf{goto}\ (pc + \#lc_1 + 1) :: \\
 & & \quad lc_1,\ T_2) \\
\mathcal{S}(\mathsf{fork(c)},\ T) & = & let\ \ (lc,\ T') = \mathcal{S}(c,\ T); in\ (\underline{\mathsf{start}}\ (\#T' + 2),\ T' :: lc :: \mathsf{return}) \\
\mathcal{C}(c) & = & let\ \ (lc,\ T) = \mathcal{S}(c,\ []); in\ \mathsf{goto}\ (\#T + 2) :: T :: lc :: \mathsf{return}
\end{array}$$

Figure 2.9: Compilation function

we will show that these properties can be guaranteed by compilation.

Concurrent extension. As shown in Definition 2.2.7, the concurrent semantics is obtained from the semantics for sequential commands together with a transition for the instruction `start`. Moreover, the sequential type system in Figure 2.8 is extended by the typing rules presented in Figure 2.6 to consider concurrent programs.

The proof of noninterference for concurrent programs relies on the existence of the function next. Similarly to the technique of [27], we name program points where control flow can branch or writes can occur. We add natural number labels to the source language as follows:

$$c ::= [x := e]^n \mid c; c \mid [\texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c]^n \mid [\texttt{while}\ e\ \texttt{do}\ c]^n \mid [\texttt{fork}(c)]^n$$

This labelling allows us to define control dependence regions for the source code and use this information to derive control dependence regions for the assembly code. We introduce two functions, sregion and tregion, to deal with control dependence regions in the source and target code, respectively.

Definition 2.2.22 (function sregion)**.** *For each branching command $[c]^n$, sregion(n) is defined as the set of labels that are inside of the command c except for those ones that are inside of* `fork` *commands.*

As in [27], control dependence regions for low-level code are defined based on the function sregion and a compilation function. For a complete source program c, we define the compilation $\mathcal{C}(c)$ in Figure 2.9. We use symbol # to compute the length of lists. Symbol :: is used to insert one element to a list or to concatenate two existing lists. The current program point in a program is represented by pc. The function $\mathcal{C}(c)$ calls the auxiliary function $\mathcal{S}$ which returns a pair of programs. The first component of that pair stores the compiled code of the main program, while the second one stores the compilation code of spawned threads. We now define control dependence regions for assembly code and respective junction points.

Definition 2.2.23 (function tregion)**.** *For a branching instruction $[c]^n$ in the source code, tregion(n) is defined as the set of instructions obtained by compiling the commands $[c']^{n'}$, where $n' \in$ sregion(n). Moreover, if c is a while loop, then $n \in$ tregion(n). Otherwise, the* goto *instruction after the compilation of the else-branch also belongs to* tregion(n).

Junction points are computed by the function jun. The domain of this function consists of every branching point in the program. We define jun as follows:

Definition 2.2.24 (junction points)**.** *For every branching point $[c]^n$ in the source program, we define* jun$(n) = max\{i | i \in$ tregion$(n)\} + 1$.

$$\frac{\vdash_\alpha c : E \qquad \vdash_\alpha c' : E}{\vdash_\alpha c; c' \ : \ E} \qquad \frac{\vdash e : L \qquad \vdash_\alpha c \ : \ E}{\vdash_\alpha [\mathsf{while}\ e\ \mathsf{do}\ c]^n_\alpha \ : \ E} \qquad \frac{\vdash e : L \qquad \vdash_\alpha c : E \qquad \vdash_\alpha c' : E}{\vdash_\alpha [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_\alpha \ : \ E}$$

$$\frac{\vdash e : H \qquad \vdash_\bullet c \ : \ E}{\vdash_\bullet [\mathsf{while}\ e\ \mathsf{do}\ c]^n_\bullet \ : \ E} \qquad \frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad \vdash_\bullet c' : E}{\vdash_\bullet [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_\bullet \ : \ E} \qquad \frac{\vdash_\alpha c : E \qquad E = \mathrm{lift}_\alpha(E, \mathsf{labels}(c))}{\vdash_\alpha [\mathsf{fork}(c)]^n_\alpha \ : \ E}$$

$$\textsc{Assign}\ \frac{\vdash e : k \qquad k \sqcup E(n) \leq \Gamma(x)}{\vdash_\alpha [x := e]^n_\alpha \ : E} \qquad \textsc{Top-H-While}\ \frac{\vdash e : H \qquad \vdash_\bullet c \ : \ E \qquad E = \mathrm{lift}_H(E, \mathsf{sregion}(n))}{\vdash_\circ [\mathsf{while}\ e\ \mathsf{do}\ c]^n_\bullet \ : \ E}$$

$$\textsc{Top-H-Cond}\ \frac{\vdash e : H \qquad \vdash_\bullet c : E \qquad \vdash_\bullet c' : E \qquad E = \mathrm{lift}_H(E, \mathsf{sregion}(n))}{\vdash_\circ [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n_\bullet \ : \ E}$$

Figure 2.10: Intermediate typing rules for high-level language commands

Having defined control dependence regions and junction points for low-level code, we proceed to defining next. Intuitively, next is only defined for instructions that belong to regions corresponding to the outermost branching points whose guards involved secrets. For every instruction i inside of an outermost branching point $[c]^n$, we define $\mathsf{next}(i) = \mathsf{jun}(n)$. Observe that this definition captures the intuition about next given in the beginning of Section 2.2.5. However, it is necessary to know, for a given program, what are the outermost branching points whose guards involved secrets. With this in mind, we extend one of the type systems given in [27] to identify such points. We add some rules for outermost branching points that involved secrets together with some extra notations to know when a command is inside of one of those points or not.

A source program c is typable, written $\vdash_\circ c : E$, if its command part is typable with respect to E according to the rules given in Figure 2.10. The typing judgement has the form $\vdash_\alpha [c]^n_{\alpha'} \ : E$, where E is a function from labels to security levels. Function E can be seen as a security environment for the source code which allows to easily define the security environment for the target code. If R is a set of points, then $\mathsf{lift}_k(E, R)$ is the security environment E' such that $E'(n) = E(n)$ if $n \notin R$ and $E'(n) = k \sqcup E(n)$ for $n \in R$. For a given program c, $\mathsf{labels}(c)$ returns all the label annotations in c. Variable α denotes if c is part of a branching instruction that branches on secrets ($\bullet$) or public data ($\circ$). Variable α' represents the level of the guards in branching instructions. The most interesting rules are $TOP-H-COND$ and $TOP-H-WHILE$. These rules can be only applied when the branching commands are the outermost ones and when they branch on secrets. Observe that such commands are the only ones that are typable considering $\alpha = \circ$ and $\alpha' = \bullet$. Moreover, the type system prevents *explicit* (via assignment) and *implicit* (via control) flows [65]. To this end, the type system enforces the same constraints as standard security type systems for sequential languages (e.g., [**?**]). Explicit flows are prevented by rule $ASSIGN$, while implicit flows are ruled out by demanding a security environment of level H inside of commands that branch on secrets. The type system guarantees information-flow security at the same time as it identifies the outermost commands that branch on secrets. Function next is defined as follows:

Definition 2.2.25 (function next)**.** *For every branching point c in the source program such that $\vdash_\circ [c]^n_\bullet$, we have that $\forall k \in \mathsf{tregion}(n).\mathsf{next}(k) = \mathsf{jun}(n)$.*

This definition satisfies the properties from Section 2.2.5, as shown by the following lemma.

Lemma 2.2.26. *Definition 2.2.25 satisfies properties **NePd** and **NeP1–4**.*

Notice that one does not need to trust the compiler in order to verify that properties **NePd** and **NeP1–4** are satisfied. Indeed, these properties are intended to be checked independently from the compiler by code consumers. We are now in condition to show the soundness of the instantiation.

Corollary 2.2.27 (Soundness of the instantiation)**.** *Hypotheses 1–6 from Section 2.2.5 are satisfied by the instantiation, and therefore the derived type system guarantees noninterference for multithreaded assembly programs.*

Hypotheses 1–3 follow from the unwinding lemmas of [25]; Hypothesis 4 from Lemma 2.2.26, and Hypotheses 5 and 6 from the definitions of next and $High^{\mathsf{lmem}}$, respectively.

Type preserving compilation. The compilation of sequential programs is type-preserving, as shown in previous work [27]. Our framework allows extending type-preservation to multithreading. Moreover, it enables us to obtain a key *non-restrictiveness* result: although the source-level type system is no more restrictive than a typical type system for a sequential language (e.g., [**?**]), the compilation of (possibly multithreaded) typable programs is guaranteed to be typable at low-level. Due to the lack of space, we only give an instantiation of this result to the source and target languages of this section:

Theorem 2.2.28. *For a given source-level program c, assume $nf(c)$ is obtained from c by replacing all occurrences of* `fork`(d) *by d. If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [**?**] then $se, \mathcal{S} \vdash \mathcal{C}(c)$ for some se and $\mathcal{S}$.*

This theorem and Theorem 2.2.11 entail the following corollary:

Corollary 2.2.29. *If command $nf(c)$ is typable under the Volpano-Smith-Irvine type system [**?**] then $\mathcal{C}(c)$ is secure.*

Java Virtual Machine. The modular proof technique developed in the previous section is applicable to a Java-like language. If the sequential type system is compatible with bytecode verification, then the concurrent type system is also compatible with it. This implies that Java bytecode verification can be extended to perform security type checking. Note that the definition of a secure scheduler is compatible with the JVM, where the scheduler is mostly left unspecified. Moreover, it is possible to, in effect, override an arbitrary scheduler from any particular implementation of JVM with a secure scheduler that keeps track of high and low threads as a part of an application's own state (cf. [171]).

Note that the semantics of the multithreaded JVM obtained by the method described in Section 2.2.2 only partially reflects the JVM specification. In particular, it ignores object locks, which are used to perform synchronization throughout program execution. Nevertheless, extending our framework to include synchronization is relatively straightforward, as explained below.

Extension to synchronization Synchronization is of fundamental importance to concurrent programs. In principle, the expressiveness of the languages described in this section allows synchronization of threads by, for instance, implementing the Lamport's bakery algorithm. This algorithm, as many others, is based on *busy waiting* and consequently has a negative impact on performance. Conversely, *blocked waiting*, which commonly underlies semaphore implementations, does not have that disadvantage. Semaphores, and generally any other mechanism based on *blocked waiting*, can potentially affect the security of programs. Therefore, it is important to provide policies regarding the use of these kind of mechanisms to preserve confidentiality. For simplicity, we describe how to incorporate semaphores to the techniques described previously.

More precisely, semaphores are special variables, written *sem*, that range over nonnegative integers and can only be manipulated by two commands: `wait`(sem) and `signal`(sem) [72]. We assume, without losing generality, that every semaphore variable is initialised with 0. For simplicity, we show the semantics of these commands at the source level in Figure 2.11. A command c and a memory m together form a *command*

$$\frac{\langle sem, m\rangle \downarrow 0}{\langle \texttt{wait}(sem), m\rangle \stackrel{\otimes sem}{\rightharpoonup} \langle \texttt{stop}, m\rangle}$$

$$\frac{\langle sem, m\rangle \downarrow n \qquad n > 0}{\langle \texttt{wait}(sem), m\rangle \rightharpoonup \langle \texttt{stop}, m[sem \mapsto n-1]\rangle}$$

$$\langle \texttt{signal}(sem), m\rangle \stackrel{\odot sem}{\rightharpoonup} \langle \texttt{stop}, m[sem \mapsto n+1]\rangle$$

Figure 2.11: Semantics for `wait`() and `signal`()

```
fork(wait(s2); l := 0; signal(p); signal(f));
fork(wait(s1); l := 1; signal(p); signal(f));
if (h ≥ l) then signal(s1); wait(p); signal(s2)
           else signal(s2); wait(p); signal(s1)
wait(p); wait(f); wait(f);
```

Figure 2.12: Attack using semaphores

configuration $\langle c, m\rangle$. A semantic step has form $\langle c, m\rangle \stackrel{\alpha}{\rightharpoonup} \langle c', m'\rangle$ that updates the command and memory in the presence of a possible event α, where $\alpha \in \{\otimes sem, \odot sem\}$. Command `wait`($sem$) blocks a thread if sem has a value of 0, indicated by event $\otimes sem$, or decrements its value by 1 otherwise. Command `signal`(sem) increments sem by 1 and triggers the event $\odot sem$.

Confidentiality of data might be compromised if commands related to semaphores are freely allowed in programs. To illustrate this, we show an attack at the source language in Figure 2.12. The program contains semaphore variables s_1, s_2, p, and f, and variables h and l to store secret and public data, respectively. The code basically blocks and unblocks threads that assign to public variables in an order that depends on h. That is, the execution of $l := 1$ is followed by $l := 0$ when $h \geq l$, and $l := 0$ is followed by $l := 1$ otherwise. As a result, some information about h is revealed. It is then clear that some restrictions about the use of semaphore is needed in order to avoid such leaks.

It is relatively straightforward to extended the techniques describe previously to incorporate semaphores to the framework (see [152] for further details). Firstly, we need to include semaphores to the sequential part of the language as, for instance, we show in Figure 2.11. Secondly, threads that are blocked on semaphore variables cannot be scheduled. Clearly, schedulers need to know when threads are blocked (or not) in order to decide if they can be run. For this purpose, it is necessary to extend the scheduler state with a set of blocked threads. Thirdly and lastly, the type system needs to enforce the secure use of semaphores. As for variables, semaphores have associated security types (L or H) and are included in the corresponding typing environment. In essence, we need two new typing rules. The first one establishes that signals to any semaphore can be performed in low threads. However, signals to public semaphores cannot be done by high threads. To illustrate why this restriction is imposed, we can think about signals on low semaphores as updates on low variables, which must be avoided inside of high threads. The second rule imposes that threads can only wait on semaphores that matches their security level. In other words, high and low threads can only wait in high and low semaphores, respectively. Clearly, high threads should not wait on low semaphores since, as before, it can be seen as updates to low variables. Besides that, waiting on semaphores might affect the timing behaviour of threads. In particular, waiting on high semaphores might affect the timing behaviour of thread depending on secret data and probably introducing internal timing leaks. For this reason, low threads cannot perform wait operations on high semaphores.

2.2.7 Related work

As discussed in Section 2.1.5, information flow type systems for sequential low-level languages, including sequential fragments of JVML, and their relation to information flow type systems for structured source languages, have been studied by several authors (e.g., [30, 81, 117, 27, 29, 25]). Nevertheless, the present work provides, to the best of our knowledge, the first proof of noninterference for a concurrent low-level language, and the first proof of type-preserving compilation for languages with concurrency.

This work exploits recent results on interaction between the threads and the scheduler [150] in order to control internal timing leaks. Other approaches [164, 174, 162, 163] to handling internal timing rely on `protect`(c) which, by definition, hides the internal timing of command c. It is not clear how to implement `protect`() without modifying the scheduler (unless the scheduler is cooperative [151, 171]). It is possible to prevent internal timing leaks by spawning dedicated threads for computations that involve secrets and carefully synchronizing the resulting threads [149]. However, this implies high synchronization costs. Yet other approaches prevent internal timing leaks in code by disallowing any races on public data [182, 95]. However, they wind up rejecting such innocent programs as $lo := 0 \parallel lo := 1$ where lo is a public variable. Still other approaches prevent internal timing by disallowing low assignments after high branching [48, 14]. Less related work [4, 158, 153, 155, 103] considers external timing, where an attacker can use a stopwatch to measure computation time. This work considers a more powerful attacker, and, as a price paid for security, disallows loops branching on secrets. For further related work, we refer to an overview of language-based information-flow security [156].

2.2.8 Conclusions on security types for multithreaded bytecode

We have presented a framework for controlling information flow in multithreaded low-level code. Thanks to its modularity and language-independence, we have been able to reuse several results for sequential languages. An appealing feature enjoyed by the framework is that security-type preserving compilation is no more restrictive for programs with dynamic thread creation than it is for sequential programs. Primitives for interacting with the scheduler are introduced by the compiler behind the scenes, and in such a way that internal timing leaks are prevented.

We have demonstrated an instantiation of the framework to a simple imperative language and have argued that our approach is amenable to extensions to object-oriented languages. The compatibility with bytecode verification makes our framework a promising candidate for establishing mobile-code security via type checking.

2.3 Extensions of the Type System

This section presents further extensions to the approach in Section 2.1. It proposes alternatives to Section 2.2 to increase the flexibility of information flow type systems and to augment the coverage of language features.

2.3.1 Extensions of Type Systems for Multithreaded Bytecode

Type System enforcing Strong Security for Multithreaded Bytecode

As presented in Section 2.2, we can directly use a security type system for sequential programs like in Section 2.1 to enforce information flow security for multithreaded programs. This is sound, if we assume a cooperative scheduler. That is, the scheduler uses notions from the security analysis (e.g. security environment) to prevent internal timing leaks. Another approach is to prevent internal timing leaks by

enforcing an adequate security condition on the multithreaded program itself. One such condition is the strong security condition [158], that has further desirable properties [154]. We introduce a variant of the strong security condition for multithreaded bytecode and provide a sound security type system.

Strong Security Condition We consider bytecode programs as presented in Section 2.1, with some deviations. As presented there, we consider programs as sets of classes with methods. The methods consist of lists of instructions. Additionally, each class contains a designated run-method, that can be used to fork a new thread by executing a start-instruction. This is different from Section 2.2. There no methods are considered and the start-instruction has a program point as its parameter. Another difference between this section and Section 2.2 is the operational semantics for thread pools. A class of schedulers is defined there while here a non-deterministic interleaving semantics is assumed, as we present below. We assume that run-methods do not return a value, while other methods always return a value. As simplification, like in Section 2.2, we do not consider exceptions in this section.

In contrast to the security condition Definition 2.1.7 in Section 2.1, the strong security condition is not based on a big-step operational semantics, but on a small-step operational semantics. That is, the semantics is defined as transitions $\rightsquigarrow$ between program configurations. A configuration consists of a heap and of a list of thread states. Each thread state is a call stack of method frames. Each method frame consists of a program counter, a mapping of method-local variables to values, and an operand stack. The thread-local part of the operational semantics defines the possible transition in a given thread state for a given heap. Using the the thread-local part of the operational semantics, the thread-global part of the operational semantics defines the possible transitions in a given given list of thread states and a given heap. That is, if a transition is possible for one thread state in the list together with the given heap then a transition exists for the whole list and the heap, where the heap is modified like in the local transition and the respective thread state is replaced by its corresponding result of the local transition.

As in Section 2.1 we assume a lattice of security levels. However, we assume slightly different observational capabilities of possible attackers. We assume the attacker to be able to observe the input and output of the whole program, whereas we consider the initial heap as input and the final heap as output. We want to prevent internal timing leaks caused by parallel composition.

To capture the observational capabilities on the different parts of the memory (heap, local variables, operand stack), we adopt the indistinguishabilities $\sim_\beta$ of Subsection 2.1.4.[8] To capture the observational capabilities on the program execution we require an observational equivalence on alternative execution paths. For instance, if the guard of a branching instruction depends on a secret then the two branches must be observationally equivalent because, otherwise, an untrusted observer might be able to deduce the value of the guard and, thereby, the secret. The PER model [157] even defines a program to be secure, if it is equivalent to itself. The strong security condition is based on this idea. The definition of the strong security condition uses the notion of a strong k_{obs}-bisimulation $\approxeq_{k_{\text{obs}}}^{P,\text{ft},\Gamma,S}$ for $k_{\text{obs}} \in \mathcal{S}$. Here, we only introduce this bisimulation informally because the formal definition is technically somewhat involved. The strong k_{obs}-bisimulation is defined as a relation between lists of program counter stacks. We consider such lists as the non-memory parts of configurations. The intuition is, that $\approxeq_{k_{\text{obs}}}^{P,\text{ft},\Gamma,S}$ relates lists of program counter stacks, that behave indistinguishability for a k_{obs}-observer. The main properties of $\approxeq_{k_{\text{obs}}}^{P,\text{ft},\Gamma,S}$ are the following. Firstly, it is symmetric. Secondly, the related lists have equal length, and each pair of program counter stacks on the same position in both lists has equal length. Thirdly, it is a stepwise bisimulation, i.e. each step of one side needs a matching step on the other side. This captures the logical timing behaviour, i.e. it captures that we consider programs like the first example in Section 2.2.1 to be insecure. Fourthly, after each step, the condition "resets" the memory, i.e. it has to be met for all indistinguishable memory pairs. Fifthly, it is point-wise, i.e. the each step of one program counter stack needs a matching step of the program

[8] Actually, for the stack indistinguishability we consider a generalisation, where two stacks are indistinguishable, if their projections on the visible elements are equal. This permits more precision for our second approach below.

counter stack on the same position on the other side. These last two properties make the security condition compositional and scheduler independent.

With this in mind we define the security condition as follows:

Definition 2.3.1 (Strong Security for Bytecode)**.** *A program P is* strongly secure *with respect to the global policy $\mathsf{ft} : \mathcal{F}_P \mapsto \mathcal{S}$, iff there is a table Γ and a $S \in \mathcal{PP} \to \mathcal{S}^*$ such that $S(1_{c_{main}}) = \epsilon$ and $\forall k_{\text{obs}} \in \mathcal{S} : 1_{c_{main}} \approx^{P,\mathsf{ft},\Gamma,S}_{k_{\text{obs}}} 1_{c_{main}}$. Here $1_{c_{main}}$ is the initial instruction of the main-method.*

Type System Since security type systems are compositional by itself, most of the standard type rules like in Section 2.1 already enforce the compositional character of the strong security condition. We do not have to impose additional restrictions on the corresponding commands. The main rules we have to change are the rules for possible branches: ifeq-instructions, method calls (possible branching by polymorphism) and exceptions (not considered here).

To deal with ifeq-instructions, we refine the approach of control dependence regions (cdr). By lifting the security environments for the regions, the previous type system already enforces that no visible assignments occur in the branches. So what remains to be enforced is that the branches cannot be distinguished by the number of their execution steps. To accomplish this we structure the control dependence region by partitioning it according to the "distances" of programs points from the branching point. We define the successor relation similar to the one in Section 2.1. We name the set of all branching points (program points with ifeq-instruction) $\mathcal{PP}^{\#}$. The *successor relation* $\mapsto \subseteq \mathcal{PP} \times \mathcal{PP}$ of the program P is the smallest relation satisfying $\forall i, i' \in \mathcal{PP} : i \mapsto i'$, if $P[i] = \mathsf{goto}\ i'$, $\forall i, i' \in \mathcal{PP} : i \mapsto i'$, if $P[i] = \mathsf{ifeq}\ i'$ and $\forall i \in \mathcal{PP} : i \mapsto i+1$, if $\forall i' \in \mathcal{PP} : P[i] \neq \mathsf{goto}\ i'$ and $P[i] \neq \mathsf{return}$.

Definition 2.3.2 (Region Stepping)**.** *Let the program P be given. Let $region : \mathcal{PP}^{\#} \to \wp(\mathcal{PP})$ and $jun : \mathcal{PP}^{\#} \rightharpoonup \mathcal{PP}$ be such that they satisfy the SOAP from Subsection 2.1.3. Let $i \in \mathcal{PP}^{\#}$. Let $n \in \mathbb{N}$. We call $region_1(i), region_2(i), \ldots, region_n(i) \subseteq region(i)$ a* region stepping *of i iff*

- *$region_1(i), region_2(i), \ldots, region_n(i)$ is a disjoint partitioning of $region(i)$,*
- *$\forall i' \in \mathcal{PP} : (i \mapsto i' \Rightarrow i' \in region_1(i))$,*[9]
- *$\forall m \in \{1, \ldots, n-1\} : \forall i' \in region_m(i) : \forall i'' \in \mathcal{PP} : (i' \mapsto i'' \Rightarrow i'' \in region_{m+1}(i))$ and*
- *$\forall i' \in region_n(i) : \forall i'' \in \mathcal{PP} : (i' \mapsto i'' \Rightarrow i'' = jun(i))$.*

We define the predicate $steppable(i)$ for $i \in \mathcal{PP}^{\#}$ as "for i exists a region stepping".

The intuition behind the region stepping is, that if for a given program point i there is a region stepping with n "steps", then all possible executions reaching i reach $jun(i)$ after exactly another $n+1$ steps.

Figure 2.13 shows the type system. Note that most conditions of the rules are quite similar to the conditions of the rules in Section 2.1. However, in the rule for the ifeq-instruction we additionally require that i is steppable. Further, in the rule for the invokevirtual-instruction we additionally require that the reference to the object of the called method has the security type L, the bottom element of the security lattice. By this we ensure that for any two indistinguishable memories always the same method is called, even if there is polymorphism. If one would allow different methods to be called, then there is the danger that they would execute a different number of steps. As simplifying side effect of this additional condition is, that we do not need to consider the heap effect of the method signature. We have two rules for the return-instruction. The first is for return-instructions in run-methods, which do not return a value. The second rule is for return-instructions in other methods.

[9] Note that $i \mapsto i'$ implies $i' \neq jun(i)$ in this section.

$$\frac{P[i] = \mathsf{prim}\ op \quad se(i), k_1, k_2 \leq k}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k_1 :: k_2 :: st \ \Rightarrow\ k :: st} \qquad \frac{P[i] = \mathsf{push}\ n}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ st \ \Rightarrow\ se(i) :: st}$$

$$\frac{P[i] = \mathsf{pop}}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k :: st \ \Rightarrow\ st} \qquad \frac{P[i] = \mathsf{load}\ x \quad se(i), \mathbf{k_a}(x) \leq k}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ st \ \Rightarrow\ k :: st}$$

$$\frac{P[i] = \mathsf{store}\ x \quad se(i), k \leq \mathbf{k_a}(x)}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k :: st \ \Rightarrow\ st} \qquad \frac{P[i] = \mathsf{goto}\ i'}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ st \ \Rightarrow\ st}$$

$$\frac{P[i] = \mathsf{ifeq}\ i' \quad \forall i'' \in region(i) : k \leq se(i'') \quad steppable(i)}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k :: st \ \Rightarrow\ lift_k(st)}$$

$$\frac{P[i] = \mathsf{return} \quad \exists c \in \mathcal{C}_P : i \in \mathcal{PP}_c}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ st \ \Rightarrow} \qquad \frac{P[i] = \mathsf{return} \quad \forall c \in \mathcal{C}_P : i \notin \mathcal{PP}_c \quad k \leq k_\mathrm{r}}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k :: st \ \Rightarrow}$$

$$\frac{\begin{array}{c} P[i] = \mathsf{invokevirtual}\ m_{\mathsf{ID}} \quad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathsf{ID}}) \quad se(i) = L \\ \Gamma_{m_{\mathsf{ID}}} = \mathbf{k_a}' \longrightarrow k'_\mathrm{r} \quad \forall n \in \{0 \ldots \mathsf{length}(st_1) - 1\} : st_1[n] \leq \mathbf{k_a}'[n+1] \end{array}}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ st_1 :: L :: st_2 \ \Rightarrow\ k'_\mathrm{r} :: st_2}$$

$$\frac{P[i] = \mathsf{start}}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ L :: st \ \Rightarrow\ st}$$

$$\frac{P[i] = \mathsf{getfield}\ f \quad se(i), \mathsf{ft}(f), k' \leq k}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k' :: st \ \Rightarrow\ k :: st}$$

$$\frac{P[i] = \mathsf{putfield}\ f \quad se(i), k_1, k_2 \leq \mathsf{ft}(f)}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ k_1 :: k_2 :: st \ \Rightarrow\ st}$$

$$\frac{P[i] = \mathsf{new}\ c}{\Gamma, region, se, \mathbf{k_a} \longrightarrow k_\mathrm{r}, i \ \vdash\ st \ \Rightarrow\ se(i) :: st}$$

Figure 2.13: Typing rules to enforce strong security with a region stepping.

We can now define typable programs.

Definition 2.3.3 (Typable Program)**.** *Let P be a program and* ft *a global policy. A method $m \in \mathcal{M}_P$ is typable by $\mathbf{k_a} \longrightarrow k_{\mathrm{r}}, se, S$ with respect to $\Gamma, region$ (denoted $\Gamma, region \vdash m : \mathbf{k_a} \longrightarrow k_{\mathrm{r}}, se, S$), if*

- $S(1_m) = \epsilon$,
- *for all program points i, i' in m such that $i \mapsto i'$ there is a stack type st such that $\Gamma, region, se, \mathbf{k_a} \longrightarrow k_{\mathrm{r}}, i \vdash S(i) \Rightarrow st$ and $st \leq S(i')$ and*
- *for all program points i in m such that $i \mapsto$, we have $\Gamma, region, se, \mathbf{k_a} \longrightarrow k_{\mathrm{r}}, i \vdash S(i) \Rightarrow$.*

The program P is typable by Γ with respect to region (denoted $region \vdash P : \Gamma$), if

$$\exists se, \Gamma, S : \forall m \in \mathcal{M}_P : \forall m_{\mathsf{lD}} : (m \in m_{\mathsf{lD}} \wedge \Gamma, region \vdash m : \Gamma(m_{\mathsf{lD}}), se, S) \ .$$

We have the following soundness result.

Theorem 2.3.4 (Soundness of Type System for Strong Security)**.** *Let P be a program and* ft *be a global policy. If there exist the functions region and jun such that they satisfy the SOAP, and $region \vdash P : \Gamma$, then P is strongly secure.*

Alternative Type System The cdr-approach as presented in Section 2.1 and the previous paragraph is flexible and efficient. The regions can be provided according to the PCC-structure by the code producer. Then the consumer only needs to check them for the SOAP before execution. Alternatively, depending on its computing power, the consumer can compute them himself. However, one shortcoming of this approach is, that in regions of high-branches no memory visible to the observer can be written to, even if there are no visible differences between the branches. Further, in the case of enforcing the strong security conditions, the region stepping requires all paths in the branches to have equal length, even if the length-difference of some paths only depends on visible information.

To overcome these shortcomings, we propose a second, alternative approach, that replaces the control dependence regions by a relation between program points, that depends on the instructions at these program points. That is, we define a computable approximation of the strong k_{obs}-bisimulation, the k_{obs}-visible equivalence relation: $\doteq^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} \supseteq =$, which is defined inductively. The definition of $\doteq^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}}$ is fairly straightforward, but the complete exposition of the formal definition is quite lengthy. Hence, we do not present the complete definition here. As an illustration, we present two examples of the rules from this definition in Figure 2.14. We use the function $vislost_{k_{\mathrm{obs}}}$, which is defined such that it collects the stack-elements that are visible in the first argument but not anymore in the second. The first rule relates two program points with an identical store-instruction storing to a k_{obs}-visible variable. The condition requires, that the successive program points are related by $\doteq^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}}$. Further, the premise that is expressed using $vislost_{k_{\mathrm{obs}}}$ imposes restrictions on the stack-types of the successive program points, such that the indistinguishability of operand stacks is preserved by execution at both program points. The second rule relates two program points with ifeq-instructions, both branching on a k_{obs}-visible element on the top of an operand stack. Like the first rule, the second rule requires that corresponding successive program points are related by $\doteq^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}}$.

On the whole, the type system is the same as in Figure 2.13, but without the conditions on the security environment. The rule for the ifeq-instruction is as follows:

$$\frac{P[i] = \mathsf{ifeq}\ i' \quad \forall k' \in \mathcal{S} : (k \not\leq k' \Rightarrow i + 1 \doteq^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} i')}{i \vdash k :: st \Rightarrow st}$$

We show the following theorem for the k_{obs}-visible equivalence.

$$\frac{P[i] = P[i'] = \mathsf{store}\ x \quad i+1 \mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} i'+1 \quad \mathbf{k_a}[x] \leq k_{\mathrm{obs}} \\ S(i+1) = k :: st \quad S(i'+1) = k' :: st' \quad vislost_{k_{\mathrm{obs}}}(S(i), st) = vislost_{k_{\mathrm{obs}}}(S(i), st')}{i \mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} i'}$$

$$\frac{\begin{array}{l} P[i] = \mathsf{ifeq}\ i^* \quad P[i'] = \mathsf{ifeq}\ i'^* \quad i+1 \mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} i'+1 \quad i^* \mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} i'^* \\ S(i) = k :: st \quad S(i') = k' :: st' \quad k, k' \leq k_{\mathrm{obs}} \\ vislost_{k_{\mathrm{obs}}}(st, S(i+1)) = vislost_{k_{\mathrm{obs}}}(st', S(i'+1)) \\ vislost_{k_{\mathrm{obs}}}(st, S(i^*)) = vislost_{k_{\mathrm{obs}}}(st', S(i'^*)) \end{array}}{i \mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} i'}$$

Figure 2.14: Exemplary rules to define $\mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}}$.

Theorem 2.3.5 (k_{obs}-Visible Equivalence)**.** *Let* $\forall m \in \mathcal{M}_P : \Gamma \vdash m : \mathbf{k_a} \longrightarrow k_{\mathrm{r}}, S$. *Then*

$$\mathrel{\dot{\simeq}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}} \subseteq \mathrel{\dot{\approx}}^{P,\mathsf{ft},\Gamma,S}_{k_{\mathrm{obs}}}$$

From this we can conclude the soundness of the type system.

Combining Calculus

Currently precision is one of the main problems for a sound analysis of information flow for multithreaded programs. That is, for too many programs that should be considered secure the analysis fails to attest security. On the one hand this is due to declassification (also called intentional information release). Many security conditions and corresponding analysis techniques (including the ones presented in this chapter) do not capture the notion of declassification. It is the goal of MOBIUS Task 2.2 to overcome this limitation and to develop a good treatment of declassification for bytecode. On the other hand this imprecision is inherent to to analysis techniques itself. We develop the *combining calculus* [115, 105] to increase precision of the analysis by combining the strengths of various analysis techniques. We define plug-in rules for various existing analysis techniques and combining rules. By this we can apply different analysis techniques to different subprograms, depending on which analysis technique is able to attest security on which subprogram. For example we have plug-in rules for a the strong security condition [158] and for observational determinism [182].

Important for the instantiation of the combining calculus is a permissive security condition (baseline condition) according to that the plug-in-rules as well as the combining rules are sound. We show the soundness of the of the combining calculus in two settings. Firstly, in [115, 105] we consider a possibilistic security condition. Secondly, in [105] we consider a probabilistic security condition assuming a uniform probabilistic scheduler.

Assume some security conditions, here SC A and SC B and corresponding sound analysis techniques SA A and SA B. Some security conditions (here SC A) can directly be used as plug-ins, some (here SC B) only with additional conditions. Figure 2.3.1 visualises this scenario. As the ellipse for the combining calculus indicates, it admits all programs that are SC A secure, all programs that are SC B secure and satisfies side conditions and also some programs that are neither SC A or SC B secure, but that satisfy the baseline security characterisation. The combining calculus analysis is sound with respect to a permissive baseline security condition SC baseline and more precise than the analysis techniques SA A and SA B alone.

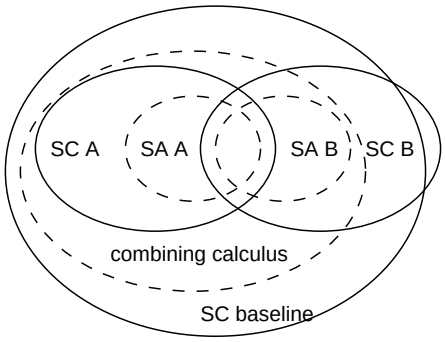

Figure 2.15: Scenario of Combining Calculus: organisation of security conditions (SC A, SC B, SC baseline) and analyses (SA A, SA B, combining calculus)

Transforming Type Systems

Security type systems provide a basis for automating the information flow analysis of concrete programs. If type checking succeeds then a program has secure information flow. If type checking fails then the program might be insecure and should not be run. After a failed type check, the task of correcting the program is often left to the programmer. Given the significance of the problem, it would be very desirable to have automated tools that better support the programmer in this task. For the future, we envision a framework for the information flow analysis that, firstly, gives more constructive advice on how a given program could be improved and, secondly, in some cases automatically corrects the program, or parts thereof, without any need for interaction by the programmer. In [103, 104] we focus on the second of these two aspects.

Obviously, one cannot allow an automatic transformation to modify programs in completely arbitrary ways as the behaviour of the transformed program should resemble the behaviour of the original program in some well-defined way. We capture such constraints by defining an equivalence relation on programs and demanding that the transformed program is equivalent to the original program under this relation. As we have seen in Section 2.3.1 with the definition of the strong security condition for bytecode, a second equivalence relation can be used to capture the objective of a transformation. Similar to above we define strong security with a strong low-bisimulation $\cong_L$ on lists of commands (thread pools). The problem of removing implicit information leaks from a program can be viewed as the problem of making alternative execution paths observationally equivalent.

In our approach, meta-variables are inserted into a program and are instantiated with programs during the transformation. The problem of making two program fragments equivalent is cast as a unification problem, which allows us to automatically compute suitable substitutions using existing unification algorithms. The approach is parametric in two equivalence relations. As semantics equivalence to be preserved by the transformation we define $\simeq$ as a weak possibilistic bisimulation. We identify a special set of of substitutions called *preserving*, such that preserving substitutions do not change the behaviour of programs with respect to $\simeq$. As adequate syntactic observational equivalence to be achieved by the transformation we define $\doteq_L$ on lists of commands, such that the following holds.

Theorem 2.3.6 (Adequacy of $\doteq_L$)**.** *If $V \doteq_L V'$ is derivable then $V \cong_L V'$ holds*

We integrate this instance of our approach into an existing transforming type system [158] for the strong security condition. That is we consider rules for type judgements of the form $V \hookrightarrow V' : S$, where C is the initial program, V' is the transformed program and S is a *low-slice* encoding the timing behaviour of V'. In the rule for branching commands this timing behaviour is used as input for a unification problem. We use

preserving substitutions to solve this problem and then apply these substitutions to both branches.

Theorem 2.3.7. *If $V \hookrightarrow V' : S$ can be derived then, firstly, V' has secure information flow, secondly, $V \simeq V'$ holds and thirdly, $V' \cong_L S$ holds.*

In [104] we also show how to suitably insert meta-variables into given programs before applying the transforming type system.

This instantiation of our approach results in a security type system that is capable of recognising some secure programs and of correcting some insecure programs that are rejected by the original type system. Moreover, the resulting programs are faster and often substantially smaller in size. Another advantage over the cross-copying technique [4], which constituted the state of the art in this area so far, is that security policies with more than two levels can be considered. Besides these technical advantages, the use of unification yields a very natural perspective on the problem of making two programs observationally equivalent. However, we do not claim that using unification will solve all problems with repairing insecure programs or that unification would be the only way to achieve the above technical advantages.

Related Work Section 2.1.5 presents related work for security type systems in general and Section 2.2.7 presents related work on security type systems for concurrent programs. In [5, 86] enforcement of timing sensitive security conditions for Java bytecode is explored. The approach to characterise security for multithreaded programs in Section 2.3.1 is based on the strong security condition [158]. We already mentioned some of the work building on the strong security condition that is not on the level of Java bytecode [153, 154, 112, 113, 114, 103, 111, 104].

Conclusion The strong security condition is the basis for various concepts in area of language-based information flow security, for example [153, 114, 103, 104, 111]. Our work is one step in making these results applicable to bytecode. This especially includes the work on declassification [114, 111], on which we will look at in MOBIUS Task 2.2. In another direction the combining calculus improves the precision of security analysis techniques by combining the strengths of existing analysis techniques. In a further direction the application of unification improves the abilities of security type systems not only the check programs, but also to correct them.

2.3.2 Types for Distributed Bytecode

The approaches to sequential and concurrent secure information flow described in this chapter so far would be correct assuming that computation proceeds without errors. Distributed systems (DS) are forms of computation in the presence of *partial failure* [126]. Important examples of partial failure are message loss, site failure, message corruption, and network partition. Typically, timing, error correction and redundancy are used to deal with partial failure. Because of partial failure, the techniques discussed in previous sections for guaranteeing secure information need to be augmented: timers for example, seemingly necessary for detecting and recovering from message failure and site crash, can also be used to mount external timing attacks [102], a potentially more powerful form of timing attack than the internal timing attack mentioned in Section 4.3.

As most ubiquitous forms of computing, especially where they use the Internet, have distributed aspects, an integration of type-based methods for secure information flow with DS is vital. We extend JVMs (see Section 2.1 above) and CJVMs (concurrent JVMs, i.e. JVMs communicating locally with other JVMs via shared memory, cf. Section 2.2) with primitives for distribution and typing systems that ensure secure information flow even in the presence of partial failure. Our extensions for DS are mostly orthogonal to the

techniques developed for JVMs and CJVMs discussed in the rest of this report. This decomposition allows for modular reasoning about secure information flow in DS. Our work proceeds in the three steps:

- First we extend JVMs with primitives for distribution and remote communication.
- Then we propose a typing system for secure information flow for the distributed JVMs.
- Finally, we add the timers to the system to be able to express more realistic error recovery scenarios. We adapt the typing system to deal with information leak enabled by timers.

Extending JVMs with Primitives for Distribution

The key mechanism JVMs provide for distributed computation is RMI (remote method invocation), which organises the call/return sequences involved when invoking a method on an object stored on a remote site. The objective of RMI is to make remote method invocations look like local invocations as much as possible. JVMs have no primitives for RMI, instead external libraries are called directly to orchestrate remote method invocation. This gives us much latitude for modelling RMI.

Our model considers CJVMs running in parallel, communicating by message passing:

$$[J_1] \mid [J_2] \mid ... \mid [J_n].$$

Each $[J_i]$ is called a *site*, running the CJVM J_i.

There is no general agreement on what kind failure are vital for good models of DS. In addition, the more complicated the failure model, the harder to reason about the ensuing system. Due to low-level error correction mechanisms in the underlying message passing mechanisms, some types of failure (e.g. message corruption) are orders of magnitude more infrequent than others (e.g. message loss). Hence it is not unreasonable to ignore such rare forms of failure. We have chosen to model one important kind of failure, *message loss*. Message within one site cannot be lost, only messages travelling between sites. The reason for this choice is in that it is already rich enough to exhibit the key problems in DS, while still being tractable. In addition it builds on previous work in this area [37, 36].

Since RMI is just a form of structured message passing, we incorporate it by simply adding channel-based message passing primitives (sending, receiving and hiding). The possibility of message loss can be represented straightforwardly in the semantics:

$$[J] \mid \overline{x}\langle v\rangle \mid [J'] \;\rightarrow\; [J] \mid [J'] \qquad\qquad [J] \mid \overline{x}\langle v\rangle \mid [J'] \;\rightarrow\; [J] \mid [\overline{x}\langle v\rangle | J']$$

Here, $\overline{x}\langle v\rangle$ is a message carrying value v to port x. This message could for example travel from $[J]$ to $[J']$ to invoke some object located at J'. The reduction step on the left simply drops this output, while that on the right lets the output reach its destination site. Within sites such message loss is not permitted. Hence sites feature only as units of (lack of) message loss.

The details of the formulation of RMI can be found in [6, 66], but one noteworthy feature of the approach in these works is that the detection of message loss is *implicit*, i.e. the invoker reaches an error-state without possibility of recovery. We call the resulting model DJVMs (distributed JVMs).

Secure Information Flow for DJVMs

The definition of secure information flow (SIF) and the development of a typing system for guaranteeing non-interference (NI) proceeds by technology transfer from the π-calculus: in [92] a comprehensive security

annotated linear/affine typing structure was developed, together with notions of SIF and NI. This was extended in [178] where *typed bisimulation* was studied, a powerful proof technique for obtaining NI-results. We used these works for obtaining typing disciplines for JVMs and CJVMs. This is possible because low-level languages with shared memory concurrency such as these two, and also others like the $\lambda\mu$-calculus have direct and precise embeddings into (a minor variant of) the linear/affine π-calculus [93]. Moreover, there is a precise correspondence between assembler languages and a sequential fragment of the linear/affine π-calculus [38]. The only essential difference between our approach and that discussed in Section 2.2 is that we do not currently consider thread-scheduling, relying on non-determinism instead. The reason for this choice is to simplify the typing system. We believe that the technique detailed in Section 2.2 for scheduling are applicable in our case as well, but leave this for future work.

In recent work [39] we extended the linear/affine typing discipline to a distributed π-calculus. The extension centred on the idea that linear channels are used in local communication (because they are guaranteed to be used exactly once). In remote communication one uses affine channels instead, with an at-most-once semantics. Linear channels cannot be used remotely due to message loss which is incompatible with the exactly-once semantics of linearity. This correspondence is precise and neither types needs to be added, nor typing rules changed when moving to distribution. It also gives a typing discipline for DJVMs that guarantees NI as a corresponding extension to the scheme described in Section 2.3.2.

This approach to SIF in DJVMs is also pleasing because it leads to a clean division of labour: to check that $[J_1] \mid [J_2] \mid ... \mid [J_n]$ is non-interfering, one checks that each CJVM J_i is locally non-interfering (under the assumption that all remote communication is non-interfering). Orthogonally, one checks that all the distributed communication is non-interfering, assuming that there is no local information leak. Under some mild side-conditions, that means the overall system is non-interfering.

Secure Information Flow for DJWMs with Timing

In a last step we made DJVMs more realistic by adding an explicit recovery mechanism from message loss [39]. Like in real systems this is achieved by way of timing: when we send a remote message, we start a timer. If the reply does not arrive within some expected time-frame, we assume the invocation message or the corresponding reply to have been lost. In that case a repeat invocation may be attempted, or some other recovery mechanism thank, like informing a user that the requested remote resource is currently unavailable. Adding timers causes two complications: (1) timers must be typed, and (2) timers allow powerful timing attacks [102] which are not possible without timers. For typing, we consider timers as *converters* of affine channels into linear channels. The idea is that a timer waits on some channel for a remote message that may or may not arrive. If it arrives, the local computation receives a message on x exactly once, reconciling local linearity with remote affine communication.

To achieve SIF in the presence of timers, the techniques of §2.2 are not applicable because timers allow more powerful information leaks than those considered in the work just cited. In addition, DS lack an explicit scheduler that can be asked or assumed to refrain certain threads from being activated at critical moments which is how Section 2.2 prevents internal timing leaks. Instead we follow Agat [4] and pad conditionals branching on high data.

Integration with and Extension from CJVM

Our work is intended to integrate with other Mobius projects, in particular that of Section 2.2, which decomposes the SIF of a shared-memory multithreaded language like the CJVM into two parts: (1) the SIF of the sequential part of a programming language, and (2) a security-preserving compilation for enforcing multithreaded SIF. This is achieved by putting suitable constraints on schedulers to prevent internal timing

leaks. Our proposals fit well into this scheme because our typing system provides a modular decomposition into distributed and concurrent features using typed timers that guarantee local liveness in the presence of potentially faulty remote communication.

An important area for improvement of our system is to consider more complicated failure models, such as message duplication, site failure and message corruption. It is likely that our approach generalises naturally to message duplication and site failure. However, message corruption undermines our type-based approach and needs other tools. We believe that a cryptography based approach like that of [52, 2] is feasible, where all remote communication is encrypted (formalised using Dolev-Yao style black-box cryptography [3]), and all carries typing information, also encrypted. Upon decryption, typing is checked dynamically, to ensure local type soundness. Finally, alternative techniques for preventing external timing attacks should be explored, since Agat's method leads to a marked reduction in program speed and is powerless against cache-based timing attacks [140].

Chapter 3

Types for Basic Resource Policies

This chapter presents different type systems and static analyses developed in the context of the MOBIUS project for inferring resource consumption. In Section 3.1, a type system is presented that bounds the amount of heap space consumed by bytecode programs. Similar to earlier work by Cachera et al. [53], loops are required not to contain heap allocating code. The type system is formally related to the MOBIUS base logic. Sections 3.2 and 3.3 present type systems for regulating access to external resources that are available via API calls. Given that the external resources cannot directly be modeled by means of cost models associating fixed costs to individual instructions, both type systems arise as static approximations for policies that could also be dynamically enforced, by execution monitoring. Section 3.2 considers the acquisition of access permissions and ensures that the program will never attempt to access a resource for which it does not have the right permissions. Extending the current programming model for MIDP, the system allows multiple permissions to be acquired in a single authorisation step. Section 3.3 proposes the notion of a resource manager, a programming abstraction (compatible with MIDP 2.0) enforcing policy-respecting usage of external resources at run-time. Again, the type system ensures that no dynamic checks are required. Finally, Section 3.4 presents a generic framework for the automatic cost analysis of sequential Java bytecode. The output of analysis is a set of *cost relations* which capture, at compile-time, the resource consumption of bytecode programs as a function of their input data size. The framework is parametric and can be used for different resource policies, including execution steps (complexity), memory consumption, external resources, etc. The proposed analysis has been implemented and is herein applied to reason about the complexity of several bytecode programs.

3.1 Heap consumption

In this section, we present a type system that ensures a constant bound on the heap consumption of bytecode programs. The type system is formally justified by a soundness proof with respect to the MOBIUS base logic, and may serve as the target formalism for type-transforming compilers.

The requirement imposed on programs is similar to that of the analysis presented by Cachera et. al. in [53] in that recursive program structures are denied the facility to allocate memory. However, our analysis is presented as a type system while the analysis presented in [53] is phrased as an abstract interpretation. In addition, Cachera et. al.'s approach involves the formalisation of the calculation of the program representation (control flow graph) and of the inference algorithm (fixed point iteration) in the theorem prover. In contrast, our presentation separates the algorithmic issues (type inference and checking) from semantic issues (the property expressed or guaranteed) as is typical for a type-based formulation. Depending on the verification infrastructure available at the code consumer side, the PCC certificate may either consist of

(a digest of) the typing derivation or an expansion of the interpretation of the typing judgments into the MOBIUS logic. The latter approach was employed in our earlier work [41] and consists of understanding typing judgments as derived proof rules in the program logic and using syntax-directed proof tactics to apply the rules in an automatic fashion. In contrast to [41], however, the interpretation given in the present section extends to non-terminating computations, albeit for a far simpler type system.

Having proved the typing rules sound w.r.t. a formal interpretation of the typing judgments in the base logic, we outline a connection to a simple first-order functional intermediate language. As MOBIUS targets compilation from Java, the use of a functional language may not immediately be required. However, it is well-known that highly impoverished functional languages (eg. without higher-order functions or polymorphism) are closely related to intermediate representations routinely employed in compilers for imperative and object-oriented lanuages [18]. Thus the work presented may be seen as an initial study towards demonstrating that the style of the bytecode-level type system is suited to serve as the target of a proof-transforming compiler, in preparation of work carried out in WP4. We prove that code resulting from compiling programs written in this language into bytecode satisfies the bound asserted by a high-level type system: derivability in the intermediate-level type system guarantees derivability in the bytecode level type system. The whole presentation is based on a formalization done in Coq , for the MOBIUS program logic. More precisely, we consider the same JVM fragment as in the MOBIUS base logic, i.e. consider method invocations, object creation and manipulation, handling and throwing exceptions etc.

Bytecode-level type system Following the notation used in the exposition of the MOBIUS logic, we consider an arbitrary but fixed bytecode program P that assigns to each method identifier a method implementation. Method identifiers m are pairs $m = (C, M)$ consisting of a class name and a method name. Program points $pc = m, l$ consist of a method identifier m and an instruction label l. We use $P(pc)$ to denote the instruction at program point pc in P, and $m \in \mathsf{dom}P$ to denote the fact that P provides an implementation for m. The initial label of the implementation of m is denoted by $init_m$, while $suc_m(l)$ denotes the successor instruction of instruction l in m.

The type system consists of judgments of the form $b \vdash_{\Sigma,\Lambda} pc : n$, expressing that the segment of bytecode whose initial instruction is located at pc is guaranteed not to allocate more than n memory cells. Here, signatures Σ and Λ assign types (natural numbers n) to identifiers of methods and bytecode instructions(in particular, when those are part of a loop), respectively. The boolean b which parametrises the rules is there to control if the assertions attached to instruction by the specification table Λ can be used as assumptions or not. This allows for properly using loop invariant information in unstructured code [1].

The rules are presented in Figure 3.1. The first rule, C-NEW, asserts that the memory consumption of a code fragment whose first instruction is $\mathsf{New}\ C$ is the increment of the remaining code. All instruction directed rules can be applied if the boolean parameter is ff. Rule C-INSTR applies to all basic instructions (in the case of $\mathsf{Goto}\ l'$ we take $suc_m(l)$ to be l'), except for $\mathsf{New}\ C$ – the predicate $basic(m, l)$ is defined as

$$basic(m,l) \equiv P(m,l) \in \left\{ \begin{array}{l} \mathsf{Iload}\ x, \mathsf{Istore}\ x, \ldots \mathsf{Const}\ t\ z, \mathsf{Ibinop}\ o, \ldots, \mathsf{New}\ C, \\ \mathsf{Getfield}\ F, \mathsf{Putfield}\ F, \mathsf{Getstatic}\ F, \mathsf{Putstatic}\ F, \ldots, \mathsf{Athrow} \end{array} \right\}.$$

in [124]. The memory effect of these instructions is zero, as is the case for return instructions, conditionals, and (static) method invocations in the case of normal termination. For exceptional termination, we require that the memory consumed after the instruction has thrown an exception is smaller with one memory unit. This is because when instructions which may throw runtime exceptions (such as Getfield, Putfield) throw such exception cause the creation of the exception object that is thrown. The rule C-ASSUM allows for using

[1] One could actually formulate the type system without the use of the boolean parameter but has to orient the type system from current instructions to previous instructions. But in order to be in compliance with the format of the MOBIUS base logic, we prefer such formulation

$$\text{C-New}\frac{n \geq 1 \quad P(m,l) = \mathsf{New}\ C \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, suc_m(l) : n-1}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n}$$

$$\text{C-Instr}\ \frac{\begin{array}{c} n \geq 1 \quad basic(m,l) \quad \neg P(m,l) = \mathsf{New}\ C \\ \mathsf{tt} \vdash_{\Sigma,\Lambda} m, suc_m(l) : n \quad \forall l', \mathsf{lookup\_handlers}(l,l') \Rightarrow\ \mathsf{tt} \vdash_{\Sigma,\Lambda} m, l' : n-1 \end{array}}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n}$$

$$\text{C-Ret}\frac{P(m,l) = \mathsf{Return}}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : 0} \qquad \text{C-If}\ \frac{n \geq 0 \quad P(m,l) = \mathsf{If0}\ l' \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, l' : n \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, suc_m(l) : n}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n}$$

$$\text{C-Invs}\frac{\begin{array}{c} n \geq 0 \quad k \geq 0 \\ P(m,l) = \mathsf{Invokestatic}\ m' \quad \Sigma(m') = k \\ m' \in \mathsf{dom} P \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, suc_m(l) : n \\ \forall l', \mathsf{lookup\_handlers}(l,l') \Rightarrow\ \mathsf{tt} \vdash_{\Sigma,\Lambda} m, l' : n-1 \end{array}}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n+k} \qquad \text{C-Invv}\frac{\begin{array}{c} n \geq 1 \quad k \geq 0 \\ P(m,l) = \mathsf{Invokevirtual}\ m' \quad \Sigma(m') = k \\ m' \in \mathsf{dom} P \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, suc_m(l) : n \\ \forall l', \mathsf{lookup\_handlers}(l,l') \Rightarrow\ \mathsf{tt} \vdash_{\Sigma,\Lambda} m, l' : n-1 \end{array}}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n+k}$$

$$\text{C-Sub}\ \frac{b \vdash_{\Sigma,\Lambda} pc : n \quad n \leq k}{b \vdash_{\Sigma,\Lambda} pc : k} \qquad \text{C-switch}\ \frac{\mathsf{ff} \vdash_{\Sigma,\Lambda} pc : n}{\mathsf{tt} \vdash_{\Sigma,\Lambda} pc : n} \qquad \text{C-assum}\ \frac{\Lambda(pc) = n}{\mathsf{tt} \vdash_{\Sigma,\Lambda} pc : n}$$

Figure 3.1: Derived proof rules for heap logic

the annotation attached to the instruction if it matches the type of the instruction. Note that in order to apply the rule the judgement to be proven must be parametrised with tt. Rule C-switch switches from an assumption mode (that is necessary if we do not have an assumption attached to an instruction as we would not be able to apply C-assum) to a mode where the instruction rules may be applied.

We call P *well-typed* for Σ, notation $\vdash_\Sigma P$, if for all m and n, $\Sigma(m) = n$ and specification table Λ for m implies $\mathsf{ff} \vdash_{\Sigma,\Lambda} m, init_m : n$ and for all points pc in the bytecode representing the body of m such that $\Lambda(pc) = k$ we have a derivation $\mathsf{ff} \vdash_{\Sigma,\Lambda} pc : k$.

Short summary of the MOBIUS logic Before outlining the interpretation of the type system, we briefly recapitulate the specification and verification structure of the MOBIUS logic. A more detailed exposition may be found in [124]. The global specification structure of a program P consists of a method specification table M, that contains for each method identifier m in P

- a method specification $\mathsf{S} = (R, T, \Phi)$, comprising a precondition R, a postcondition T, and a method invariants Φ,
- a local specification table G, i.e. a context of local proof assumptions, and
- a local annotation table Q that collects the (optional) assertions associated with labels in m.

An entry $\mathsf{M}(m) = ((R, T, \Phi), \mathsf{G}, \mathsf{Q})$ is to be understood as follows. The tuple (R, T) represents a partial-correctness specification, i.e. the postcondition $T(s_0, t)$ is expected to hold whenever an execution of m with initial state s_0 that satisfies $R(s_0)$ terminates, where t is the final state. The tuple (R, Φ) represents a method invariant, i.e. the assertion $\Phi(s_0, s)$ is expected to hold for any state s that arises during the (terminating or non-terminating) execution of m with initial state s_0 satisfying $R(s_0)$. The annotation table Q is a finite partial map from labels occurring in m to assertions $Q(s_0, s)$. If label l is annotated by Q, then $Q(s_0, s)$ will be expected to hold for any state s encountered at program point (m, l) during a terminating or non-terminating execution of m with initial state s_0 satisfying $R(s_0)$. Finally, the proof context G collects proof assumptions that may be used during the verification of the method. It consists of a finite partial map

from labels in m to the components (A, B, I) of local proof judgements $\mathsf{G}, \mathsf{Q}, b \vdash \{A\}\, pc\, \{B\}\, (I)$ parametrised with a boolean value b which indicates if the assertions in G can be used as assumptions in the proof in the same way as in the type system. In the following, we shall omit the boolean parameter for the sake of clarity.

The verification of bytecode phrases uses local judgements of the form $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$. Here,

1. A is a (local) precondition, i.e. a predicate $A(s_0, s)$ that relates the state s at program point pc (i.e. the state prior to executing the instruction at that program point) to the initial state s_0 of the current method invocation,

2. B is a (local) postcondition, i.e. a predicate $B(s_0, s, t)$ that relates the state s at pc to the initial state s_0 and the final state t of the current method invocation, provided the execution of the current method invocation terminates,s

3. I is a (local) invariant, i.e. a predicate $I(s_0, s, H)$ that relates the state s at pc to the initial state s_0 of the current method invocation and the heap component H of any future state encountered during the continued execution of the current method, including those arising in sub-frames.

4. G is the proof context which may be used to store recursive proof assumptions, as needed e.g. for the verification of loops.

Additionally, if $pc = (m, l)$ and $\mathsf{Q}(l) = Q$, then the judgement $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$ implicitly also mandates that $Q(s_0, s)$ holds for all states s encountered at l, where s_0 is as before.

The verification task for full programs consists of showing that M is justified. For each entry $\mathsf{M}(m) = (\mathsf{S}, \mathsf{G}, \mathsf{Q})$, we need to show that:

1. The body b_m of m satisfies the method specification. This amounts to deriving the judgement $\mathsf{G}, \mathsf{Q} \vdash \{A_0\}\, m, \mathit{init}_m\, \{B_0\}\, (I_0)$ where the assertion A_0, B_0, and I_0 are obtained by converting the method specification S into the format suitable for the local proof judgement.

2. All entries in the proof context G are justified.

3. The specification table M satisfies the behavioural subtyping condition, i.e. the specification of an overriding method implies the specification of the overridden method.

Together, these conditions form the verified-program property, which we denote by $\mathsf{M} \vdash P$.

Interpretation of the type system The interpretation for the above type system is now obtained by defining for each number n a triple $[\![n]\!] = (A, B, I)$ consisting of a precondition A, a postcondition B, and an invariant I, as follows.

$$[\![n]\!] \equiv \left(\begin{array}{l} \lambda\, (s_0, s).\ \mathit{True}, \\ \lambda\, (s_0, s, t).\ |\mathit{heap}(t)| \leq |\mathit{heap}(s)| + n, \\ \lambda\, (s_0, s, H).\ |H| \leq |\mathit{heap}(s)| + n \end{array} \right)$$

Here, $|H|$ denotes the size of heap H. We specialise the main judgement form of the bytecode logic to

$$\mathsf{G}, \mathsf{Q} \vdash pc\ \{n\} \quad \equiv \quad \mathit{let}\ (A, B, I) = [\![n]\!]\ \mathit{in}\ \mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I).$$

By the soundness of the MOBIUS logic, the derivability of a judgement $\mathsf{G}, \mathsf{Q} \vdash pc\ \{n\}$ guarantees that executing the code located at pc will not allocate more that n items, in terminating (postcondition B) and non-terminating (invariant I) cases. For $(A, B, I) = [\![n]\!]$ we also define the method specification

$$Spec\ n \equiv (\lambda\ s_0.\ True, \lambda\ (s_0, t).\ B(s_0, state(s_0), t), \lambda\ (s_0, H).\ I(s_0, state(s_0), H)).$$

Finally, we say that M satisfies Σ, notation $\mathsf{M} \models \Sigma$, if for all methods m provided with a specification table Λ and a natural number n, such that $\Sigma(m) = n$ and for all instructions pc in m $\Lambda(pc) = k$ holds exactly if $\mathsf{M}(m) = (Spec\ n, Spec\ \Lambda, \emptyset)$. Thus, we require annotation tables Q to be empty.

We can now prove the soundness of the typing rules with respect to this interpretation. By induction on the typing rules, we first show that the interpretation of a typing judgement is derivable in the logic.

Proposition 3.1.1. *Let method m be such that $\mathsf{M} \models \Sigma$, $m \in \mathsf{dom}\Sigma$ and m is provided with specification table Λ such that $b \vdash_{\Sigma,\Lambda} m, l : n$. Then $\emptyset, \Lambda \vdash \{m, l\}\ n$.*

Based on this result, the fact that the behavioural subtyping condition trivial due to the absence of virtual methods, and the fact that proof context are empty, it is easy to see that well-typed programs satisfy the verified-program property:

Theorem 3.1.2. *Let $\mathsf{M} \models \Sigma$ and $\vdash_\Sigma P$. Then $\mathsf{M} \vdash P$.*

Implementation in Coq We have implemented the type systems in terms of the Coq implementation of the Mobius base logic. The implementation of the type system effort consists in around 1220 lines in Coq , it contains 11 lemmas which show that the rules of the type system are derivable in the MOBIUS base logic.

In particular, the implementation provides a rule for every group of instructions with similar behavior. We illustrate this by two excerpts from the implementation - the rule for basic instructions and static method calls. The following rule treats all the basic instructions (basic instructions in the sense of MOBIUS formalization are defined as all instructions except method calls, return instruction and jump instructions) except for instance creation. The rule imposes several structural restrictions on the format of the specification and annotation. The first condition (*isMethSpecTableDerivedAss ME*) requires that all method specifications must be of the interpretation of the type format of the type system as shown in the previous paragraph. In order to fit the format of type system, the derived rule requires that there are no internal specifications to be proven (*LAT Assertion*) = (*PCM.empty Assertion*). Finally, annotations (assertions for helping the proof) must be also of the format used by the type system (*isLSTDerivedAss LST*). The next conditions restrict the application of the rule only to basic instructions (*isBasicInstr M I*) different from instance creation (*not* (*isNewInstr M I*)). The rest of the conditions coincide with the condition of the type system, i.e. the executions starting from the next instruction uses as much memory as the executions starting from the current instructions. Because of the semantics of throwing a runtime exception, the executions from the exception handler instructions consume at most one object less than the executions from the current instruction.

```
Lemma derivableBASIC: ∀ (P :Program) (I :PC) (n:Z) (M :Method) ME LST LAT,
  (isMethSpecTableDerivedAss ME) →
  (LAT Assertion) = (PCM.empty Assertion)→
  isLSTDerivedAss LST →
  isBasicInstr M I →
  not ( isNewInstr M I) →
  1 ≤ n →
  (∀ II, next M I = Some II →
```

SP_Judgement P ME M LST (LAT Assertion) true II (derivedLocalPre n) (derivedLocalPost n) (derivedStrongInv n)) →
(∀ II e bm,
METHOD.body M = Some bm →
lookup_handlers P (BYTECODEMETHOD.exceptionHandlers bm) I e II →
SP_Judgement P ME M LST (LAT Assertion) true II
(derivedLocalPre (n-1)) (derivedLocalPost(n-1)) (derivedStrongInv (n - 1))) →
SP_Judgement P ME M LST (LAT Assertion) false I (derivedLocalPre n)(derivedLocalPost n) (derivedStrongInv n).

The rule for static methods also requires restrictions on the format of assertions as in the above case. Moreover, it requires that the method specification of the invoked method is of the expected form (interpretation of the type). The rule requires that the successor instruction consumes at most n memory cells. The executions from instructions at which a possible exception handler for the method call starts must consume at most n - *1* memory cells.

Lemma *derivableINVS*: ∀ (*P*:*Program*) (*I* :*PC*)(*msig*:*MethodSignature*)(*k n*:*Z*)(*M MM* :*Method*)
ME LST LAT (*LS*:*Logic.LocalSpecTable*) (*LA*:*LocalAnnoTable*) (*Nargs* :*nat*),
isMethSpecTableDerivedAss ME →
LAT = (PCM.empty Assertion) →
isLSTDerivedAss LST →
instructionAt M I = Some(Invokestatic msig) →
findMethod P msig = Some MM →
METHOD.isNative MM = false →
Nargs = length (METHODSIGNATURE.parameters (snd msig)) →
MSPEC_LOOKUP ME msig =
Some (((((fun s0 : InitState ⇒ True),,
(fun s0 s ⇒ heapSize (getHeapR s)- heapSize (getHeapI s0) ≤ k)),,
(fun s0 s ⇒ heapSize (getHeapL s) - heapSize (getHeapI s0) ≤ k)),,LS),,LA) →
0 ≤ n →
0 ≤ k →
(∀ ll, next M I = Some ll →
SP_Judgement P ME M LST LAT true ll
(derivedLocalPre n) (derivedLocalPost n) (derivedStrongInv n)) →

(∀ ll e bm,
METHOD.body M = Some bm →
lookup_handlers P (BYTECODEMETHOD.exceptionHandlers bm) I e ll →
SP_Judgement P ME M LST LAT true ll (derivedLocalPre n -1) (derivedLocalPost n -1)
(derivedStrongInv n - 1)) →

SP_Judgement P ME M LST LAT false I (derivedLocalPre (n + k)) (derivedLocalPost (n + k))
(derivedStrongInv (n + k)) .

Example To illustrate how a derivation in the type system will work, we consider the following simple Java program:

```
public class A {
  int val;
  A next;
```

```
  A containsVal(A list ) {
   while (list.next != null ) {
     if (list.val == val) {
        return list;
     }
     list = list.next;
   }
   return new A();
  }
}
```

The class `A` is an implementation of a list data structure, where the value of a single node is stored in the field `val` and the pointer to the next element in the list is the field `next`. The class is provided with a simple method `m` which iterates over the argument list object and if it contains an element with value `val` it returns the rest of the argument `list`. If no element in the argument list is equal to the integer value `val` the method returns a new list. Note that the loop statement in the method implementation dereferences (e.g. `list.next` = val!) the argument fields which may potentially throw a runtime exception of type NullPointerException if the dereferenced object reference is null. As we said previously, raising a runtime exception by the Java Virtual Machine is related to a creation of a new instance object of the respective exception type. The other source of memory consumption is the instance creation of a new list.

One could notice that method `containsVal` consumes at most one memory unit - if a null reference is dereferenced this triggers an exception and thus, one memory unit is consumed and the method terminates abruptly execution. If no exception is thrown (no memory used by the exception mechanism) during the loop execution, then the rest of the program execution will consume one memory unit. In both of the cases - normal or abrupt termination, the program will consume one memory unit. However, the type system can infer successfully that the method `containsVal` consumes not more than two memory units during any of its executions. This is an over-approximation, due to the fact that the type system is not "aware" if a reference is null or not.

For verifying the example program with the Coq implementation of the derived logical rules for the type system we used the Bico tool which allows for converting Java class files into the Mobius formalization Bicolano of Java bytecode programs which is as follows:

```
Definition mInstructions : MapN.t (Instruction×option PC) :=
  (bc_cons 0%N (Vload Aval 1%N) 1%N
  (bc_cons 1%N (Getfield ASignature.nextFieldSignature) 4%N
  (bc_cons 4%N (Ifnull EqRef 28%Z) 7%N
  (bc_cons 7%N (Vload Aval 1%N) 8%N
  (bc_cons 8%N (Getfield ASignature.nextFieldSignature) 11%N
  (bc_cons 11%N (Vload Aval 0%N) 12%N
  (bc_cons 12%N (Getfield ASignature.valFieldSignature ) 15%N
  (bc_cons 15%N (If_icmp EqInt 20%N) 18%N
  (bc_cons 18%N (Vload Aval 1%N) 19%N
  (bc_cons 19%N (Vreturn Aval)
  (bc_cons 20%N (Vload Aval 1%N) 21%N
  (bc_cons 21%N (Getfield ASignature.nextFieldSignature) 24%N
  (bc_cons 24%N (Vstore Aval 1%N) 25%N
  (bc_cons 25%N (Goto (-25)%Z) 28%N
  (bc_cons 28%N (New AType.className) 31%N
  (bc_cons 31%N (Dup) 32%N
  (bc_cons 32%N (Invokevirtual ASignature._init_Signature) 35%N
  (bc_single 35%N (Vreturn Aval))))))))))))))))))))).
```

The method `containsVaInstructions` is given the following specification. First the precondition of the method is *True*:

Definition *preconditionContainsVal* := *fun* (*s0* : *InitState*) ⇒ *True*.

The postcondition states that the method execution is not consuming more than two memory units:

Definition *postconditionContainsVal* :=
fun (*s0* : *InitState*) (*s* :*ReturnState*) ⇒
heapSize (*getHeapR s*) - *heapSize* (*getHeapI s0*) ≤ 2.

The method strong invariants states at every reachable state of the method execution the memory consumed so far is not greater than two memory units: Definition *strongInvariantContainsVal*:=

fun (*s0* : *InitState*) (*s* :*LocalState*) ⇒
heapSize (*getHeapL s*) - *heapSize* (*getHeapI s0*) ≤ 2.

We thus construct an object of type *MethSpec* which represents the method contract of `SpecContainsVal`: Definition *methodSpecContainsVal*: *MethSpec* :=

((*preconditionContainsVal*,,*postconditionContainsVal*),, *strongInvariantContainsVal*).

In order to conform to the format of the type assertions the *localAnnotTableContainsVal* should be empty:

Definition *localAnnotTableContainsVal* := *PCM.empty Assertion*.

The specification table for the method contains information for the property that must hold every time the loop entry instruction 0 is reached. The loop invariant states, that at this point the memory consumed so far is 2. Note that a stronger invariant holds here, namely that the memory consumed so far is 0 but the invariant that we chose is sufficient for the proof:

Definition *locSpecM* : *LocalSpecTable* :=
PCM.update
(*PCM.empty* _)
0%*N*
((*derivedLocalPre* 2,,*derivedLocalPost* 2),, *derivedStrongInv* 2).

Finally, we construct the Coq specification object of the method as follows:

Definition *specM* : (*MethSpec* ** *LocalSpecTable* ** *LocalAnnoTable*) :=
((*methodSpecM*,, *locSpecM*),, *localAnnotTableM*).

Finally, the lemma that we want to proof about our program is the following:

Lemma :
SP_Judgement program program_spec A.mMethod
locSpecM localAnnotTableM false 0%*N* (*derivedLocalPre* 2) (*derivedLocalPost* 2) (*derivedStrongInv* 2).

The proof of this lemma contains around 600 Coq commands. The proof basically follows the type derivation. The logical rules used are the derived type rules and the rest are applications of the Coq reduction tactics *compute* and *simpl* and the resolution tactic *auto*. The proof is routine and one can identify patterns for a single instructions which can allow for the automatic generation of a Coq proof from a type derivation.

Discussion As the example illustrates, the type system makes an overestimation for the memory consumption of a program. To deal with this a possibility is to combine the type system with a NullPointer analysis for instance. For this, we should specialise the proof rules for instructions which might throw a NullPointer exception. If we can assert that an instruction which dereferences a reference will always deal with a non null reference we may use a weaker typing rule. For instance, there will be two rules for a getfield

$$\text{T-INT}\ \frac{}{\Sigma \rhd i : 0} \qquad \text{TP-UN}\ \frac{}{\Sigma \rhd \mathsf{uop}\ u\ x : 0} \qquad \text{TP-BIN}\ \frac{}{\Sigma \rhd \mathsf{bop}\ o\ x\ y : 0}$$

$$\text{TP-NIL}\ \frac{}{\Sigma \rhd \mathsf{Nil} : 0} \qquad \text{T-CONS}\ \frac{}{\Sigma \rhd \mathsf{Cons}(x, y) : 1} \qquad \text{T-CALL}\ \frac{\Sigma(m) = n}{\Sigma \rhd m(x) : n}$$

$$\text{T-PRIM}\ \frac{\Sigma \rhd p : n}{\Sigma \rhd \mathsf{prim}\ p : n} \quad \text{T-LET}\ \frac{\Sigma \rhd p : n \quad \Sigma \rhd e : k}{\Sigma \rhd \mathsf{let}\ x = p\ \mathsf{in}\ e : n + k} \quad \text{T-SUB}\ \frac{\Sigma \rhd e : k \quad k \leq n}{\Sigma \rhd e : n}$$

$$\text{T-COND}\ \frac{\Sigma \rhd e_1 : n \quad \Sigma \rhd e_2 : n}{\Sigma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : n} \qquad \text{T-CASE}\ \frac{\Sigma \rhd e_1 : n \quad \Sigma \rhd e_2 : n}{\Sigma \rhd \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons}(x, y)\ \Rightarrow e_2 : n}$$

Figure 3.2: High-level typing rules

instruction, one where we have that the NullPointer analysis *refNotNull* shows that the dereferenced object is always different from null:

$$\text{C-GETFLD1}\frac{\mathit{getField}(m, l) \quad \mathit{refNotNull}(m, l) \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n}$$

and the other one, when the NullPointerAnalysis cannot infer such information would resemble the original rule that we saw before for basic instructions:

$$\text{C-GETFLD2}\frac{n \geq 1 \quad \mathit{getField}(m, l) \quad \mathsf{tt} \vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n \quad \forall l', \mathsf{lookup\_handlers}(l, l') \Rightarrow\ \mathsf{tt} \vdash_{\Sigma,\Lambda} m, l' : n - 1}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n}$$

We can also work out a special rule for basic instructions that can or not throw a Runtime exception. For instance, if we specialise a rule for the instructions that do not throw runtime exceptions (such as load, store, dup) would look as follows:

$$\text{C-NORTE}\ \frac{\mathsf{tt} \vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n \quad \mathit{noExceptionInstr}(m, l)}{\mathsf{ff} \vdash_{\Sigma,\Lambda} m, l : n}$$

This allows not to consider the exceptional case for instructions whose semantics makes them always terminate normally.

Intermediate-level type system We consider a functional language that is suitably restricted to serve as an intermediate code representation, similar to the one presented in [17]. The syntax is stratified into primitive expressions and general expressions similar to the A-normal form discipline [80]. We include primitives for constructing empty and non-empty lists and a case expression former for deconstructing lists – other algebraic data types could be included in a similar way. In order to simplify the translation into bytecode, we employ bytecode-level method identifiers m as function names. Functions are restricted to have only a single formal parameter.

$$\begin{aligned} \mathcal{P} \ni p &::= i \mid \mathsf{uop}\ u\ x \mid \mathsf{bop}\ o\ x\ y \mid \mathsf{Nil} \mid \mathsf{Cons}(x, y) \mid m(x) \\ \mathcal{E} \ni e &::= \mathsf{prim}\ p \mid \mathsf{let}\ x = p\ \mathsf{in}\ e \mid \mathsf{if}\ x\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \Rightarrow e \mid \mathsf{Cons}(x, y)\ \Rightarrow e) \end{aligned}$$

A program F consists of a collection of function declarations in the standard way, i.e. for function name m, the declaration $F(m) = (x, e)$ consists of an expression e with at most the free variable x.

Figure 3.2 presents the rules for a type system with judgements of the form $\Sigma \rhd p : n$ and $\Sigma \rhd e : n$. As before, signatures Σ map function identifiers to types n. Apart from the construction of a non-empty list and function calls, all primitive expressions have the trivial type 0. This includes Nil which is compiled to a null reference. Intuitively, the types play the same role as at the low level n, i.e. a typing $\Sigma \rhd e : n$ is intended to represent the fact that the evaluation of e consumes no more than n allocations, provided any function f evaluated en route conforms to its specification in Σ. In particular, recursive functions will only be typeable for type 0.

$$
\begin{array}{rcl}
[\![i]\!]_l^C & = & (C[l \mapsto \mathsf{const}\ i], l+1) \\
[\![\mathsf{uop}\ u\ x]\!]_l^C & = & (C[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{unop}\ u], l+2) \\
[\![\mathsf{bop}\ o\ x\ y]\!]_l^C & = & ([l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{load}\ y, l+2 \mapsto \mathsf{binop}\ o], l+3) \\
[\![\mathsf{Nil}]\!]_l^C & = & (C[l \mapsto \mathsf{const\ Null}], l+1) \\
[\![\mathsf{Cons}(x,y)]\!]_l^C & = & (C \left[\begin{array}{l} l \mapsto \mathsf{load}\ y, l+1 \mapsto \mathsf{load}\ x, l+2 \mapsto \mathsf{new\ LIST}, \\ l+3 \mapsto \mathsf{store\ t}, l+4 \mapsto \mathsf{load\ t}, \\ l+5 \mapsto \mathsf{putfield\ LIST\ HD}, l+6 \mapsto \mathsf{load\ t}, \\ l+7 \mapsto \mathsf{putfield\ LIST\ TL}, l+8 \mapsto \mathsf{load\ t} \end{array}\right], l+9) \\
[\![m(x)]\!]_l^C & = & (C[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{Invokestatic}\ m], l+2) \\
[\![\mathsf{prim}\ p]\!]_l^C & = & \mathit{let}\ (C_1, l_1) = [\![p]\!]_l^C\ \mathit{in}\ (C_1[l_1 \mapsto \mathsf{Return}], l_1+1) \\
[\![\mathsf{let}\ x = p\ \mathsf{in}\ e]\!]_l^C & = & \mathit{let}\ (C_1, l_1) = [\![p]\!]_l^C, (C_2, l_2) = (C_1[l_1 \mapsto \mathsf{store}\ x], l_1+1) \\
 & & \mathit{in}\ [\![e]\!]_{l_2}^{C_2} \\
[\![\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_l^C & = & \mathit{let}\ (C_E, l_2) = [\![e_2]\!]_{l+2}^C, (C_T, l_1) = [\![e_1]\!]_{l_2}^{C_E} \\
 & & \mathit{in}\ (C_T[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{If0}\ l_2], l_1) \\
\left[\!\!\left[\begin{array}{l} \mathsf{case}\ x\ \mathsf{of} \\ \quad \mathsf{Nil} \Rightarrow e_1 \\ \quad \mid \mathsf{Cons}(x,y)\ \Rightarrow e_2 \end{array}\right]\!\!\right]_l^C & = & \mathit{let}\ (C_C, l_N) = [\![e_2]\!]_{l+9}^C, (C_N, l_1) = [\![e_1]\!]_{l_N}^{C_C}\ \mathit{in} \\
 & & (C_N \left[\begin{array}{l} l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{unop}\ (\lambda\ v.\ v = Nullref), \\ l+2 \mapsto \mathsf{If0}\ l_N, l+3 \mapsto \mathsf{Load}\ x, \\ l+4 \mapsto \mathsf{Getfield\ LIST\ HD}, l+5 \mapsto \mathsf{Store}\ h, \\ l+6 \mapsto \mathsf{Load}\ x, l+7 \mapsto \mathsf{Getfield\ LIST\ TL}, \\ l+8 \mapsto \mathsf{Store}\ t \end{array}\right], l_1)
\end{array}
$$

Figure 3.3: Translation into bytecode

Definition 3.1.3. *Program F is well-typed w.r.t. signature Σ, notation $\Sigma \rhd F$, if $\mathsf{dom}\Sigma = \mathsf{dom}F$ and for all m, e and x, $F(m) = (x, e)$ implies $\Sigma \rhd e : \Sigma(m)$.*

Figure 3.3 defines a compilation $[\![e]\!]_l^C$ into the bytecode language. The translation is defined using an auxiliary compilation function $[\![p]\!]_l^C$ for primitive expressions. In both cases, the result (C', l') extends the code fragment C by a code block starting at l such that l' is the next free label. Primitive expressions leave an item on the operand stack while proper expressions translate into method suffixes.

We write $P = [\![F]\!]$ if P contains precisely the translations of the function declarations in F, i.e. for all m, x, and e we have $F(m) = (x, e)$ precisely if $m \in \mathsf{dom}P$ and the implementation $P(m)$ is $([\![e]\!]_l^{[]}, \mathit{init}_m)$.

Type soundness for primitive expressions justifies the high-level typing judgements by showing the derivability of suitable low-level judgements for the compiled code. It shows that an execution commencing at l satisfies the bound that is obtained by adding the costs for the subject expression to the costs for the program continuation.

Proposition 3.1.4. *If $\Sigma \rhd p : n$, $[\![p]\!]_l^C = (C_1, l_1)$, $\mathsf{dom}\Sigma \subseteq \mathsf{dom}P$, and $b \vdash_{\Sigma,\Lambda} m, l_1 : k$, then $b \vdash_{\Sigma,\Lambda} m, l : n + k$.*

For proper expressions, the soundness result does not mention program continuations, since expressions compile to code blocks that terminate with a method return.

Proposition 3.1.5. *If $\Sigma \rhd e : n$, $\mathsf{dom}\Sigma \subseteq \mathsf{dom}P$, and $[\![e]\!]_l^C = (C_1, l_1)$, then $b \vdash_{\Sigma,\Lambda} m, l : n$.*

Both results are easily proven by induction on the typing judgement. We thus have that well-typed high-level programs yield well-typed bytecode programs.

Proposition 3.1.6. *If $\Sigma \rhd F$ and $P = [\![F]\!]$ then $\vdash_\Sigma P$.*

Combining this result with Theorem 3.1.2 yields the final soundness result.

Theorem 3.1.7. *If $\Sigma \rhd F$, $P = [\![F]\!]$ and $\mathsf{M} \models \Sigma$, then $\mathsf{M} \vdash P$.*

Discussion on the style of the soundness proof In this section, we presented a bytecode-level type system with a formalised soundness proof with respect to the MOBIUS program logic, and a translation from a high-level type system into the bytecode level formalism. Together, these results yield a soundness proof for the high-level type system with respect to a particular compilation strategy.

Traditionally, soundness proofs of type systems have often been performed purely on the high language level, i.e. w.r.t. an operational semantics for the functional language. Usually, the soundness proof is then performed by induction on the syntax or the typing rules (subject-reduction), possibly aided by substitution lemmas. In the context of MOBIUS however, such a syntactic proof is unsatisfactory, for two reasons:

1. it results in a way to certify the behaviour of transmitted bytecode only if the compilation from the functional language into bytecode is trusted or certified. In the case of intensional properties such as memory consumption, a soundness result for the compilation function would have to include a (formalised/trusted) proof that the allocation annotations in the high-level type system correctly describe the memory allocations performed by the JVM

2. depending on the style of operational semantics (big-step evaluation relation vs. small-step reductions), high-level soundness results often do not apply to non-terminating executions, even if these are covered by intermediate auxiliary lemmas or stronger proof invariants.

For these reasons, we argue that in the context of the MOBIUS project, purely syntactic soundness results are insufficient. The proof as presented above avoids the definition of an operational semantics at the functional language level, but contains a formalised translation. In previous work [41] we have explored a further alternative which avoids the formalisation of the compilation function $[\![.]\!]$. In this approach, interpretations of high-level typing judgements are directly derived in the program logic for code segments that correspond to the high-level expression formers. This derived-proof-rules-approach is closely related to the approach presented in the present document: it replaces the formulation of the low-level type system as a set of inductive proof rules by a set of derived lemmas whose justifications are identical to the soundness proofs of the low-level typing rules. Thus, no formal relationship between the two language levels needs to be established – in fact, no type judgements need to be represented explicitly in the theorem prover, as the specialised proof rules only operate on their interpretations. As was demonstrated in [41], this alternative approach may be applied for more complex type systems that involve sharing constraints and memory reuse, provided that the interpretations are sufficiently strong. The omission of the high-level operational model and the language translation from the trusted code base represents an improvement w.r.t. formalisation effort and manageability of the TCB. Compared to the abstract interpretation approach presented in [53], our approach avoids the calculation of the control flow graph, the (admittedly reusable) representation of the abstract-interpretation framework, and the inference mechanism from the TCB.

3.2 Permission analysis

Beyond the computational resources of memory space and CPU time, MOBIUS Deliverable 1.1 [123] identifies a number of other resources worth statically bounding on connected mobile devices, e.g., mobile phones. Among these resources are "billable events" such as initiating a phone call or sending a text message. That is, unlike for the computational resources the cost of these *external* resources is not defined via a computational cost model where each instruction costs, and where the total cost of a program execution does not depend much on the cost of a single instruction execution. Rather, the cost of external resources is defined by external, non-computational entities, e.g., the business model of the phone operator, and the cost of a program execution may well strongly depend on the cost of few or even a single instruction execution.

In order to access external resources, a MIDlet has to call the respective MIDP API methods. The current MIDP security model protects each resource access by user interaction as all API methods accessing a resource must pop up a confirmation screen, which makes midlets soft targets for social engineering attacks (see [123] paragraph 3.3.1). Reducing the number of user interactions (as advocated in the resource scenarios

in subsection 5.2 of [123]) would reduce the social engineering threat but does not comply with the current MIDP security architecture.

INRIA develops an enhanced security model which improves on the current MIDP architecture. (A paper based on this work was published at the ESORICS'06 conference [44].) The important features of this enhanced security model are:

- the possibility for applications to request multiple permissions in advance;
- a static enforcement of the security model.

Because the proposed model does not require security screens to pop-up before each resource access, it reduces the need for user-interactions: it is more flexible and user-friendly. However, ensuring that programs do not abuse resources is not as straightforward as it is for MIDP where a permission request immediately matches a resource access. In our novel setting, this property is established by *static analysis*. Precisely, it enforces that *a program will never attempt to access a resource for which it does not have permission*. Hence, our enhanced model comes without extra runtime checks. This is a crucial advantage for devices with reduced computing capabilities.

In Section 3.3, UEDIN develops a different approach for tackling the same problem of certifying that programs do not abuse external resources.

3.2.1 The Java MIDP security model

The Java MIDP programming model for mobile telephones [169] proposes a thoroughly developed security architecture which is the starting point of our work. In the MIDP security model, applications (called *midlets* in the MIDP jargon) are downloaded and executed by a Java virtual machine. Midlets are made of a single archive (a jar file) containing complete programs. At load time, the midlet is assigned a protection domain which determines how the midlet can access resources. It can be seen as a labelling function which classifies a resource access as either `allowed` or `user`.

- `allowed` means that the midlet is granted unrestricted access to the resource;
- `user` means that, prior to an access, an interaction with the user is initiated in order to ask for permission to perform the access and to determine how often this permission can be exercised. Within this protection domain, the MIDP model operates with three possibilities:
  - `blanket`: the permission is granted for as long as the midlet remains installed;
  - `session`: the permission is granted for as long as the midlet is running;
  - `oneshot`: the permission is granted for a single use.

The `oneshot` permissions correspond to dynamic security checks in which each access is protected by a user interaction. This clearly provides a secure access to resources but the potentially numerous user interactions are at the detriment of the usability and make user interactions are at the detriment of the usability and make social engineering attacks easier. At the other end of the spectrum, the `allowed` mode which gets granted through signing provides a maximum of usability but leaves the user with absolutely no assurance on how resources are used, as a signature is only a certificate of integrity and origin.

In the following we will propose a security model which extends the MIDP model by introducing permissions with multiplicities and by adding flexibility to the way in which permissions are granted by the user and used by applications. In this model, we can express:

- the `allowed` mode and `blanket` permissions as initial permissions with multiplicity ∞;
- the `session` permissions by prompting the user at application start-up whether he grants the permission for the session and by assigning an infinite number of the given permission;

- the `oneshot` permissions by prompting the user for a permission with a `grant` just before consuming it with a `consume`.

The added flexibility is obtained by allowing the programmer to insert user interactions for obtaining permissions at any point in the program (rather than only at the beginning and just before an access) and to ask for a batch of permissions in one interaction. The added flexibility can be used to improve the usability of access control in a midlet but will require formal methods to ensure that the midlet will not abuse permissions (security concern) and will be granted by the programmer sufficient permissions for a correct execution (usability concern). The analysis presented in section 3.2.5 is addressing these two concerns.

3.2.2 The structure of permissions

In classical access control models, permissions held by a subject (user, program, ...) authorise certain *actions* to be performed on certain *resources*. Such permissions can be represented as a relation between actions and resources. To obtain a better fit with access control architectures such as that of Java MIDP we combine this permission model with multiplicities and resource types, in a way inspired by the resource allocation matrices used by Millen in his resource allocation model [121]. However, we add more structure to the set of resources. Concrete MIDP permissions are strings whose prefixes encode package names and whose suffixes encode a specific permission. For instance, one finds permissions `javax.microedition.io.Connector.http` and `javax.microedition.io.Connector.sms.send` which enable applets to make connections using the http protocol or to send a SMS, respectively. Thus, permissions are structured entities that for a given resource type define which actions can be applied to which resources of that type and how many times.

To model this formally, we assume given a set *ResType* of resource types. For each resource type rt there is a set of resources Res_{rt} of that type and a set of actions Act_{rt} applicable to resources of that type. We incorporate the notion of multiplicities by attaching to a set of actions a and a set of resources r a multiplicity m indicating how many times actions a can be performed on resources from r. Multiplicities are taken from the ordered set:

$$Mul \triangleq (\mathbb{N} \cup \{\bot_{Mul}, \infty\}, \leq).$$

The 0 multiplicity represents absence of a given permission and the ∞ multiplicity means that the permission is permanently granted. The $\bot_{Mul}$ multiplicity represents an error arising from trying to decrement the 0 multiplicity. We define the operation of decrementing a multiplicity as follows:

$$m - 1 = \begin{cases} \infty & \text{if } m = \infty \\ m - 1 & \text{if } m \in \mathbb{N}, m \neq 0 \\ \bot_{Mul} & \text{if } m = 0 \text{ or } m = \bot_{Mul} \end{cases}$$

Several implementations of permissions include an *implication ordering* on permissions. One permission implies another if the former allows to apply a particular action to more resources than the latter. However, the underlying object-oriented nature of permissions imposes that only permissions of the same resource type can be compared. We capture this in our model by organising permissions as a dependent product of permission sets for a given resource type.

Definition 3.2.1 (Permissions)**.** *Given a set ResType of resource types and ResType-indexed families of resources Res_{rt} and actions Act_{rt}, the set of atomic permissions $Perm_{rt}$ is defined as:*

$$Perm_{rt} \triangleq (\mathcal{P}(Res_{rt}) \times \mathcal{P}(Act_{rt})) \cup \{\bot\}$$

relating a type of resources with the actions that can be performed on it. The element $\bot$ represents an invalid permission. By extension, we define the set of permissions Perm as the dependent product:

$$Perm \triangleq \prod_{rt \in ResType} Perm_{rt} \times Mul$$

relating for all resource types an atomic permission and a multiplicity stating how many times it can be used. For $\rho \in Perm$ and $rt \in ResType$, we use the notations $\rho(rt)$ to denote the pair of atomic permissions and multiplicities associated with rt in ρ. Similarly, $\mapsto$ is used to update the permission associated to a resource type, i.e., *$(\rho[rt \mapsto (p, m)])(rt) = (p, m)$.*

Example 3. *Given a resource type $SMS \in ResType$, the permission $\rho \in Perm$ satisfying $\rho(SMS) = ((+1800*, \{send\}), 2)$ grants two accesses to a send action of the resource $+1800*$ (phone number starting with $+1800$) with the type SMS.*

Definition 3.2.2. *The ordering $\sqsubseteq_p \subseteq Perm \times Perm$ on permissions is given by*

$$\rho_1 \sqsubseteq_p \rho_2 \stackrel{\triangle}{=} \forall rt \in ResType \quad \rho_1(rt) \sqsubseteq \rho_2(rt)$$

where $\sqsubseteq$ is the product of the subset ordering $\sqsubseteq_{rt}$ on $Perm_{rt}$ and the $\leq$ ordering on multiplicities.

Intuitively, being higher up in the ordering means having more permissions to access a larger set of resources. The ordering induces a greatest lower bound operator $\sqcap : Perm \times Perm \to Perm$ on permissions. For example, for $\rho \in Perm$

$$\rho[File \mapsto ((/tmp/*, \{read, write\}), 1)] \ \sqcap \ \rho[File \mapsto ((*/dupont/*, \{read\}), \infty)] = \rho[File \mapsto ((/tmp/*/dupont/*, \{read\}), 1)]$$

There are two operations on permissions that will be of essential use:

- consumption (removal) of a specific permission from a collection of permissions;
- update of a collection of permissions with a newly granted permission.

Definition 3.2.3. *Let $\rho \in Perm$, $rt \in ResType$, $p, p' \in Perm_{rt}$, $m \in Mul$ and assume that $\rho(rt) = (p, m)$. The operation consume : $Perm_{rt} \to Perm \to Perm$ is defined by*

$$consume(p')(\rho) = \begin{cases} \rho[rt \mapsto (p, m-1)] & \text{if } p' \sqsubseteq_{rt} p \\ \rho[rt \mapsto (\bot, m-1)] & \text{otherwise} \end{cases}$$

There are two possible error situations when trying to consume a permission. Attempting to consume a resource for which there is no permission ($p' \not\sqsubseteq_{rt} p$) is an error. Similarly, consuming a resource for which the multiplicity is zero will result in setting the multiplicity to $\bot_{Mul}$.

Definition 3.2.4. *A permission $\rho \in Perm$ is an error, written $Error(\rho)$, if:*

$$\exists rt \in ResType, \exists (p, m) \in Perm_{rt} \times Mul, \rho(rt) = (p, m) \wedge (p = \bot \vee m = \bot_{Mul}).$$

Granting a number of accesses to a resource of a particular resource type is modeled by updating the component corresponding to that resource type.

Definition 3.2.5. *Let $\rho \in Perm$, $rt \in ResType$, the operation grant : $Perm_{rt} \times Mul \to Perm \to Perm$ for granting a number of permissions to access a resource of a given type is defined by*

$$grant(p, m)(\rho) = \rho[rt \mapsto (p, m)]$$

Notice that granting such a permission erases all previously held permissions for that resource type, *i.e.*, permissions do not accumulate. This is a design choice: the model forbids that permissions be granted for performing one task and then used later on to accomplish another. The *grant* operation could also add the granted permission to the existing ones rather than replace the corresponding one. Besides cumulating the number of permissions for permissions sharing the same type and resource, this would allow different resources for the same resource type. However, the `consume` operation becomes much more complex, as a choice between the overlapping permissions may occur. Analysis would require handling multisets of permissions or backtracking.

Another consequence of the fact that permissions do not accumulate is that our model can impose scopes to permissions. This common programming pattern is naturally captured by inserting a `grant` instruction with null multiplicity at the end of the permission scope.

3.2.3 Program model

We model a program by a control-flow graph (CFG) that captures the manipulations of permissions (grant and consume), the handling of program loops, method calls and returns, and models the way that exceptions are thrown and handled in a language like Java. These operations are respectively represented by the instructions $\texttt{grant}(p, m)$, $\texttt{consume}(p)$, $\texttt{call}^i$, $\texttt{return}$, $\texttt{throw}(ex)$, with $i \in N, ex \in EX, rt \in ResType, p \in Perm_{rt}$ and $m \in Mul$. This is an enriched version of the models used in previous work on modelling access control for Java [98, 43, 33]. The instruction $\texttt{call}^i$ combines method calls and iteration. Its semantics is that it will call a particular method i times at a given point in the execution, unless an exception is raised. Ordinary method calls correspond to $\texttt{call}^i$ with $i = 1$. The $\texttt{throw}(ex)$ instruction will throw the exception ex. This exception can be caught inside the method currently executing in which case an edge indicates the way to the relevant handler. Otherwise, it escapes the method and will be handled in the enclosing methods.

Definition 3.2.6. *A control-flow graph is a 7-tuple*

$$G = (NO, EX, KD, TG, CG, EG, n_0)$$

where:

- *NO is the set of nodes of the graph;*
- *EX is the set of exceptions;*
- *$KD : NO \to \{\texttt{grant}(p, m), \texttt{consume}(p), \texttt{call}^i, \texttt{return}, \texttt{throw}(ex)\}$, associates a kind to each node, indicating which instruction the node represents;*
- *$TG \subseteq NO \times NO$ is the set of intra-procedural edges;*
- *$CG \subseteq NO \times NO$ is the set of inter-procedural edges, which can capture dynamic method calls;*
- *$EG \subseteq EX \times NO \times NO$ is the set of intra-procedural exception edges that will be followed if an exception is raised at that node;*
- *n_0 is the entry point of the graph.*

To rule out certain graphs that do not model Java programs, control-flow graphs are subject to the following additional structural constraints: 1) a return node has no successor in the graph; 2) all nodes (except return nodes) have a successor by the TG relation; 3) a call node has (at least) one successor by the CG relation; 4) a method has a unique entry point.

In the following, given $n, n' \in NO$ and $ex \in EX$, we will use the notations $n \overset{TG}{\to} n'$ for $(n, n') \in TG$, $n \overset{CG}{\to} n'$ for $(n, n') \in CG$ and $n \overset{ex}{\to} n'$ for $(ex, n, n') \in EG$.

Figure 3.4 contains an example of the control flow graph of `grant` and `consume` operations from a fictitious flight-booking transaction. For simplicity, actions related to permissions, such as $\{connect\}$ or $\{read\}$, are omitted. In this transaction, the user first transmits his request to a travel agency, *site*. He can then modify his request or get additional information. Once satisfied with information provided, he can either book the flight or pay the desired flight. In both cases, the identity of a credit card is required, hence the corresponding permission is asked form the outset.

In the case of payment, the application asks for permission to access information concerning banking detail such as the credit card number *etc.* This could also have been asked from the start as part of p_{init}, but is instead obtained via a dynamic request that is being executed only if the payment branch of the application is actually chosen. In the example, the developer has chosen to delay asking for the permission of accessing credit card information until it is certain that this permission is indeed needed. Another design choice would be to grant this permission from the outset. This would minimise user interaction because it allows to remove the querying `grant` operation. However, the initial permission p_{init} would then contain $file \mapsto (/wallet/*, 2)$ instead of $file \mapsto (/wallet/id, 1)$ which goes against the Principle of Least Privilege.

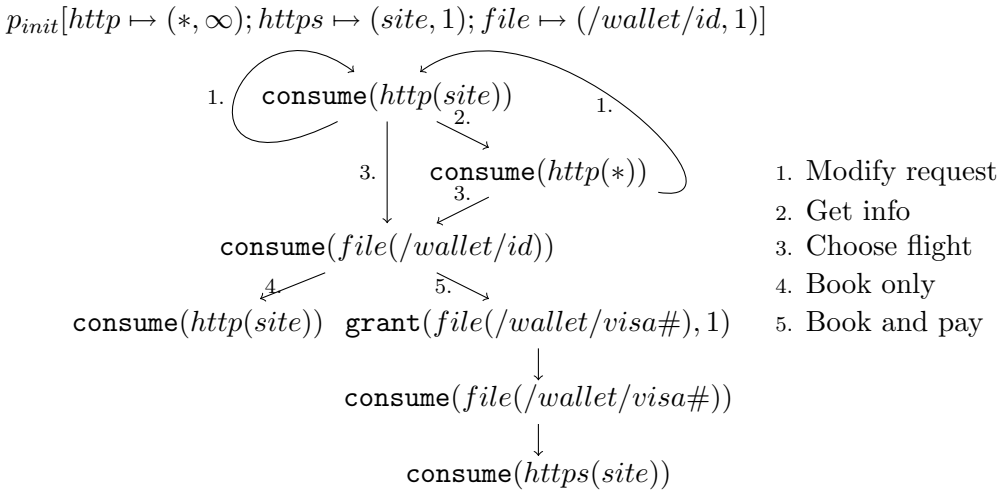

Figure 3.4: Example of grant/consume permissions patterns

3.2.4 Operational semantics

We define the small-step operational semantics of CFGs in Figure 3.5. The semantics is stack-based and follows the behaviour of a standard programming language with exceptions, e.g., as Java or C♯. Instantiating this model to such languages consists of identifying in the code the desired `grant` and `consume` operations, building the control-flow graph and describing the action of the other instructions on the stack.

The operational semantics operates on a state consisting of a standard control-flow stack of nodes, enriched with the permissions held at that point in the execution. Thus, the small-step semantics is given by a relation $\twoheadrightarrow$ between elements of $D = (NO \times (NO \times \mathbb{N})^* \times (EX \cup \{\epsilon\}) \times Perm)$. Given $(n, s, e, \rho) \in D$, n is the current control point, s is a call stack of nodes labelled by counters that model iteration loops, e is an exception (if any) and ρ is the current set of permissions. For example, for the instruction $\texttt{call}^i$ of Figure 3.5, if the current node n leads through an inter-procedural step to a node m, then the node m is added to the top of the stack $n^i{:}s$, with $s \in (NO \times \mathbb{N})^*$. In order to take into account the repetitive aspects of the $\texttt{call}^i$ instruction, we add a formal exponent i on the n component of the stack. This exponent is used to remember how many times the method should be called and is updated in the rules for the method return instruction `return` as follows. If the exponent i is positive, then the method has to be executed again, in a context where the exponent now is decremented by one. Otherwise, if the exponent is zero, the iteration is finished and execution proceeds at the node following the call node n.

Instructions may change the value of the permission along with the current state. *E.g.*, for the instruction `grant` of Figure 3.5, the current permission ρ of the state will be updated with the new granted permissions. The current node of the stack n will also be updated, at least to change the program counter, depending on the desired implementation of `grant`. Note that the instrumentation is *non-intrusive*, *i.e.* a transition will not be blocked due to the absence of a permission. Thus, for $(n, s, e, \rho) \in D$ if there exists $(n', s', e', \rho') \in D$ such that $n, s, e, \rho \twoheadrightarrow n', s', e', \rho'$, then for all ρ there is a ρ' such that the same transition holds.

This operational semantics will be the basis for the notion of program execution traces, on which global results on the execution of a program will be expressed.

Definition 3.2.7 (Trace of a CFG)**.** *A partial trace $tr \in NO^*$ of a CFG is a sequence of nodes $n_0 :: n_1 :: \ldots :: n_j$ that is produced by semantics steps of the form $n_0, \epsilon, \epsilon, p_{init} \twoheadrightarrow n_1, s_1, e_1, \rho_1 \ldots \twoheadrightarrow n_j, s_j, e_j, \rho_j$. For a program P represented by its control-flow graph G, we will denote by $[\![P]\!]$ the set of all partial traces of G.*

To state and verify the safety of a program that acquires and consumes permissions, we first define what it means for an execution trace to be safe. We define the permission set available at the end of a trace by

$$\frac{KD(n) = \texttt{grant}(p,m) \quad n \overset{TG}{\to} n'}{n, s, \epsilon, \rho \twoheadrightarrow n', s, \epsilon, grant(p,m)(\rho)} \qquad \frac{KD(n) = \texttt{consume}(p) \quad n \overset{TG}{\to} n'}{n, s, \epsilon, \rho \twoheadrightarrow n', s, \epsilon, consume(p)(\rho)}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m}{n, s, \epsilon, \rho \twoheadrightarrow m, n^i{:}s, \epsilon, \rho}$$

$$\frac{KD(r) = \texttt{return} \quad n \overset{CG}{\to} m \quad m \overset{TG^*}{\to} r \quad i \geq 1}{r, n^i{:}s, \epsilon, \rho \twoheadrightarrow m, n^{i-1}{:}s, \epsilon, \rho} \qquad \frac{KD(r) = \texttt{return} \quad n \overset{TG}{\to} n' \quad i = 1}{r, n^1{:}s, \epsilon, \rho \twoheadrightarrow n', s, \epsilon, \rho}$$

$$\frac{KD(n) = \texttt{throw}(ex) \quad n \overset{ex}{\to} h}{n, s, \epsilon, \rho \twoheadrightarrow h, s, \epsilon, \rho} \qquad \frac{KD(n) = \texttt{throw}(ex) \quad \forall h, n \overset{ex}{\not\to} h}{n, s, \epsilon, \rho \twoheadrightarrow n, s, ex, \rho}$$

$$\frac{\forall h, n \overset{ex}{\not\to} h}{t, n{:}s, ex, \rho \twoheadrightarrow n, s, ex, \rho} \qquad \frac{n \overset{ex}{\to} h}{t, n{:}s, ex, \rho \twoheadrightarrow h, s, \epsilon, \rho}$$

Figure 3.5: Small-step operational semantics

induction over its length.

$$\begin{array}{lcll} PermsOf(nil) & \triangleq & p_{init} & \\ PermsOf(tr :: n) & \triangleq & consume(p, PermsOf(tr)) & \text{if} \quad KD(n) = \texttt{consume}(p) \\ PermsOf(tr :: n) & \triangleq & grant((p,m), PermsOf(tr)) & \text{if} \quad KD(n) = \texttt{grant}(p) \\ PermsOf(tr :: n) & \triangleq & PermsOf(tr) & \text{otherwise} \end{array}$$

p_{init} is the initial permission of the program, for the state n_0. By default, if no permission is granted at the beginning of the execution, it will contain $((\emptyset, \emptyset), 0)$ for each resource type. The `allowed` mode and `blanket` permissions for a resource r of a given resource type can be modeled by associating the permission $((\{r\}, Act), \infty)$ with that resource type.

A trace is *safe* if none of its prefixes end in an error situation due to the access of resources for which the necessary permissions have not been obtained.

Definition 3.2.8 (Safe trace)**.** *A partial trace $tr \in NO^*$ is* safe*, written $Safe(tr)$, if for all prefixes $tr' \in prefix(tr)$, $\neg Error(PermsOf(tr'))$.*

3.2.5 Static analysis of permission usage

We now define a constraint-based static flow analysis for computing a safe approximation, denoted P_n, of the permissions that are guaranteed to be available at each program point n in a CFG when execution reaches that point. Thus, safe means that P_n underestimates the set of permissions that will be held at n during the execution. The approximation will be defined as a solution to a system of constraints over P_n, derived from the CFG following the rules in Figure 3.6. The rules for P_n are straightforward data flow rules: *e.g.*, for `grant` and `consume` we use the corresponding semantic operations *grant* and *consume* applied to the start state P_n to get an upper bound on the permissions that can be held at end state $P_{n'}$. Notice that the set $P_{n'}$ can be further constrained if there is another flow into n'. The effect of a method call on the set of permissions will be modeled by a transfer function R defined below. These transfer functions describe the effect of one execution of a method so to take into account the iteration in a $\texttt{call}^i$, we apply the R function i times to the set of permissions available at point of the method call.

Finally, throwing an exception at node n that will be caught at node m means that the set of permissions at n will be transferred to m and hence form an upper bound on the set of available permissions at this point.

$$\frac{KD(n) = \texttt{grant}(p,m) \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq_p grant(p,m)(P_n)} \qquad \frac{KD(n) = \texttt{consume}(p) \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq_p consume(p)(P_n)}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq_p R^i_m(P_n)} \qquad \frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m}{P_m \sqsubseteq_p P_n}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{ex}{\to} h}{P_h \sqsubseteq_p (R^{i-1}_m; R^{ex}_m)(P_n)} \qquad \frac{KD(n) = \texttt{throw}(ex) \quad n \overset{ex}{\to} m}{P_m \sqsubseteq_p P_n}$$

Figure 3.6: Constraints on minimal permissions

Our CFG program model includes procedure calls which means that the analysis must be inter-procedural. We deal with procedures by computing *summary functions* for each procedure. These functions summarise how a given procedure consumes resources from the entry of the procedure to the exit, which can happen either normally by reaching a `return` node, or by raising an exception which is not handled in the procedure. More precisely, for a given CFG we compute the quantity $R : (EX \cup \{\epsilon\}) \to NO \to (Perm \to Perm)$ with the following meaning:

- the partial application of R to ϵ is the effect on a given initial permission of the execution from a node until return;
- the partial application of R to $ex \in EX$ is the effect on a given initial permission of the execution from a node until reaching a node which throws an exception ex that is not caught in the same method.

Given nodes $n, n' \in NO$, we will use the notation R_n and R^{ex}_n for the partial applications of $R\ \epsilon\ n$ and $R\ ex\ n$. The rules are written using diagrammatic function composition ; such that $F; F'(\rho) = F'(F(\rho))$. We define an order $\sqsubseteq$ on functions $F, F' : Perm \to Perm$ by extensionality such that $F \sqsubseteq F'$ if $\forall \rho \in Perm, F(\rho) \sqsubseteq_p F'(\rho)$.

As for the entities P_n, the function R is defined as solutions to a system of constraints. The rules for generating these constraints are given in Figure 3.7 (with $e \in EX \cup \{\epsilon\}$). The rules all have the same structure: compose the effect of the current node n on the permission set with the function describing the effect of the computation starting at n's successors in the control flow. This provides an upper bound on the effect on permissions when starting from n. As with the constraints for P, we use the functions *grant* and *consume* to model the effect of `grant` and `consume` nodes, respectively. A method call $\texttt{call}^i$ at node n that does not cause an exception is modeled by i compositions of the R_m function of the start node of the called method m followed by the effect of the computation starting at the successor node n' of call node n.

The correctness of our analysis is stated on execution traces. For a given program, if a solution of the constraints computed during the analysis does not contain errors in permissions (*cf.* Definition 3.2.4), then the program will behave safely. Formally,

Theorem 3.2.9 (Basic Security Property)**.** *Given a program P, let P_n and R_n be a solution to the constraints generated by Pg. If R_n are monotone functions, then*

$$\forall n, (\forall p, KD(n) = consume(p) \Rightarrow \neg Error(P_n)) \Rightarrow \forall tr \in [\![P]\!], Safe(tr).$$

3.2.6 Constraint solving

Computing a solution to the constraints generated by the analysis in Section 3.2.5 is complicated by the fact that solutions to the R-constraints (see Figure 3.7) are functions from $Perm$ to $Perm$ that have infinite domains and hence cannot be represented by a naive tabulation. To solve this problem, we identify a class of functions that are sufficient to encode solutions to the constraints while restricted enough to allow

$$\frac{KD(n) = \texttt{grant}(p,m) \quad n \overset{TG}{\to} n'}{R^e_n \sqsubseteq grant(p,m); R^e_{n'}} \qquad \frac{KD(n) = \texttt{consume}(p) \quad n \overset{TG}{\to} n'}{R^e_n \sqsubseteq consume(p); R^e_{n'}}$$

$$\frac{KD(n) = \texttt{return}}{R_n \sqsubseteq \lambda\rho.\rho} \qquad \frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{TG}{\to} n'}{R^e_n \sqsubseteq R^i_m; R^e_{n'}}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad \forall n', n \overset{ex}{\nrightarrow} n'}{R^{ex}_n \sqsubseteq R^{ex}_m} \qquad \frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{ex}{\to} h}{R_n \sqsubseteq R^{i-1}_m; R^{ex}_m; R_h}$$

$$\frac{KD(n) = \texttt{throw}(ex) \quad n \overset{ex}{\to} h}{R^{ex}_n \sqsubseteq R^{ex}_h} \qquad \frac{KD(n) = \texttt{throw}(ex) \quad \forall n', n \overset{ex}{\nrightarrow} n'}{R^{ex}_n \sqsubseteq \lambda\rho.\rho}$$

Figure 3.7: Summary functions of the effect of the execution on initial permission

effective computations. Given a solution to the R-constraints, the P-constraints (see Figure 3.6) are solved by standard fixpoint iteration.

The rest of this section is devoted to the resolution of the R-constraints. The resolution technique consists in applying solution-preserving transformations to the constraints until they can be solved either symbolically or iteratively.

On simplifying R-constraints

In our model, resources are partitioned depending on their resource type. At the semantic level, *grant* and *consume* operations ensure that permissions of different types do not interfere *i.e.*, that it is impossible to use a resource of a given type with a permission of a different type. We exploit this property to derive from the original system of constraints a family of independent *ResType*-indexed constraint systems. A system modelling a given resource type, say rt, is a copy of the original system except that *grant* and *consume* are indexed by rt and are specialized accordingly:

$$grant_{rt}(p'_{rt'}, m') = \begin{cases} \lambda(p,m).(p',m') & \text{if } rt = rt' \\ \lambda(p,m).(p,m) & \text{otherwise} \end{cases}$$

$$consume_{rt}(p'_{rt'}) = \begin{cases} \lambda(p,m).(\text{if } p' \sqsubseteq_{rt'} p \text{ then } p \text{ else } \bot, m-1) & \text{if } rt = rt' \\ \lambda(p,m).(p,m) & \text{otherwise} \end{cases}$$

Further inspection of these operators shows that multiplicities and atomic permissions also behave in an independent manner. As a result, each *ResType* indexed system can be split into a pair of systems: one modelling the evolution of atomic permissions; the other modelling the evolution of multiplicities. Hence, solving the R-constraints amounts to computing for each exception e, node n and resource type rt a pair of mappings:

- an atomic permission transformer ($Perm_{rt} \to Perm_{rt}$) and
- a multiplicity transformer ($Mul \to Mul$).

In the next sections, we define syntactic representations of these multiplicity transformers that are amenable to symbolic computations.

Constraints on multiplicity transformers

Before presenting our encoding of multiplicity transformers, we identify the structure of the constraints we have to solve. Multiplicity constraints are terms of the form $x \dot{\leq} e$ where $x : Mul \to Mul$ is a variable over

multiplicity transformers, $\dot{\leq}$ is the point-wise ordering of multiplicity transformers induced by $\leq$ and e is an expression built over the terms

$$e ::= v \mid grant_{Mul}(m) \mid consume_{Mul}(m) \mid id \mid e; e$$

where

- v is a variable;
- $grant_{Mul}(m)$ is the constant function $\lambda x.m$;
- $consume_{Mul}(m)$ is the decrementing function $\lambda x.x - m$;
- id is the identity function $\lambda x.x$;
- and $f; g$ is function composition ($f; g = g \circ f$).

We define $MulF = \{\lambda x.min(c, x-d) | (c, d) \in Mul \times Mul\}$ as a restricted class of multiplicity transformers that is sufficiently expressive to represent the solution to the constraints. Elements of $MulF$ encode constant functions, decrementing functions and are closed under function composition as shown by the following equalities:

$$grant_{Mul}(m) = \lambda x.min(m, x - \bot_{Mul})$$
$$consume_{Mul}(m) = \lambda x.min(\infty, x - m)$$
$$\lambda x.min(c, x - d'); \lambda x.min(c', x - d') = \lambda x.min(min(c - d', c'), x - (d' + d))$$

We represent a function $\lambda x.min(c, x-d) \in MulF$ by the pair (c, d) of multiplicities. Constraint solving over $MulF$ can therefore be recast into constraint solving over the domain $MulF^\sharp = Mul \times Mul$ equipped with the interpretation $[\![(c, d)]\!] \stackrel{\triangle}{=} \lambda x.min(c, x - d)$ and the ordering $\sqsubseteq^\sharp$ defined as $(c, d) \sqsubseteq^\sharp (c', d') \stackrel{\triangle}{=} c \leq c' \wedge d' \leq d$.

Solving multiplicity constraints

The domain $MulF^\sharp$ does not satisfy the *descending chain condition*. This means that iterative solving of the constraints might not terminate. Instead, we use an elimination-based algorithm. First, we split our constraint system over $MulF^\sharp = Mul \times Mul$ into two constraint systems over Mul. Example 4 shows this transformation for a representative set of constraints.

Example 4. *$C = \{Y \sqsubseteq^\sharp (c, d), Y' \sqsubseteq^\sharp X, X \sqsubseteq^\sharp Y;^\sharp Y'\}$ is transformed into $C' = C_1 \cup C_2$ with $C_1 = \{Y_1 \leq c, Y_1' \leq X_1, X_1 \leq min(Y_1 - Y_2', Y_1')\}$ and $C_2 = \{Y_2 \geq d, Y_2' \geq X_2, X_2 \geq Y_2' + Y_2\}$.*

Notice that C_1 depends on C_2 but C_2 is independent from C_1. This result holds generally and, as a consequence, these sets of constraints can be solved in sequence: C_2 first, then C_1.

To be solved, C_2 is converted into an equivalent system of fixpoint equations defined over the complete lattice $(Mul, \leq, max, \bot_{Mul})$. The equations have the general form $x = e$ where $e ::= var \mid max(e, e) \mid e + e$. The elimination-based algorithm unfolds equations until a direct recursion is found. After a normalisation step, recursions are eliminated using a generalisation of Proposition 3.2.10 for an arbitrary number of occurrences of the x variable.

Proposition 3.2.10. *$x = max(x + e_1, e_2)$ is equivalent to $x = max(e_2 + \infty \times e_1, e_2)$.*

Given a solution for C_2, the solution of C_1 can be computed by standard fixpoint iteration as the domain $(Mul, \leq, min, \infty)$ does not have infinite descending chains. This provides multiplicity transformer solutions of the R-constraints.

3.2.7 Towards relational permission analysis

We have proposed an access control model for programs which dynamically acquire permissions to access resources. The model extends the current access control model of the Java MIDP profile for mobile telephones by introducing multiplicities of permissions together with explicit instructions for granting and consuming permissions. These instructions allow to improve the usability of an application by fine-tuning the number and placement of user interactions that ask for permissions. In addition, programs written in our access control model can be formally and statically verified to satisfy the fundamental property that a program does not attempt to access a resource for which it does not have the appropriate permission. The formalisation is based on a model of permissions which extends the standard object $\times$ action model with multiplicities. We have given a formal semantics for the access control model, defined a constraint-based analysis for computing the permissions available at each point of a program, and shown how the resulting constraint systems can be solved. To the best of our knowledge, it is the first time that a formal treatment of the Java MIDP model has been proposed.

The present model and analysis has been developed in terms of control-flow graphs and has ignored the treatment of data such as integers *etc.* By combining our analysis with standard data flow analysis we can obtain a better approximation of integer variables and hence, e.g., the number of times a permission-consuming loop is executed. Allowing a `grant` to take a variable as multiplicity parameter combined with a relational analysis based on polyhedra would allow to verify a program as the following.

```
1 grant(sendSMS(∗),addr_book.length) ;
2   // 0 ≤ addr_book.length = #SMSpermissions
3 for (i = 0 , i < addr_book.length, i++)
4   // 0 < addr_book.length − i ≤ #SMSpermissions
5   if (∗) consume(SMS(addr_book[i].no),send);
6   // 0 ≤ #SMSpermissions
```

Here, the number of requested permissions depends on the size of the address book data structure, and where the verification needs to establish a relation between several program entities in order to prove that the number of permissions is always non-negative.

The permission analysis presented here is not complete in the sense that there are certain graphs whose traces are all safe but that are not deemed secure by the analysis. This happens when there are inaccessible sub-graphs in the CFG. These may be flagged by the analysis as responsible for security violations despite the fact that they will never be executed. Hence, the analysis can be strengthened by first computing reachable nodes and only launching the permission analysis on these. We conjecture that this lack of *reachability sensitivity* is the only origin of incompleteness. As reachability can be computed exactly for our model of programs, the permission analysis would then be complete. This property is useful to understand and eliminate certain potential sources of false alarms when our analysis is combined with other analyses.

3.3 Explicit Accounting of External Resources

As explained in the previous section, abuse of external resources (like sending text messages) is a major concern on mobile devices. UEDIN has developed a Java library for trapping abuse of external resources by monitoring their use at run-time. Complementary to the run-time library is a type system for statically certifying applications as *resource safe*, *i.e.,* never being trapped due to abuse of resources. Both approaches share a common API, based on *resource managers*, for explicitly accounting at run-time which external resources an application is granted to use and how often. If the type system certifies an application as resource safe then the run-time accounting is unnecessary and can be erased without changing the observable behaviour of the application.

The work described in this section tackles the same problem as Section 3.2, but with a different approach. Both approaches extend the current MIDP security model to authorise a number of resources (or permissions) before use; Section 3.2 introduces special `grant` and `consume` instructions for this purpose whereas we provide

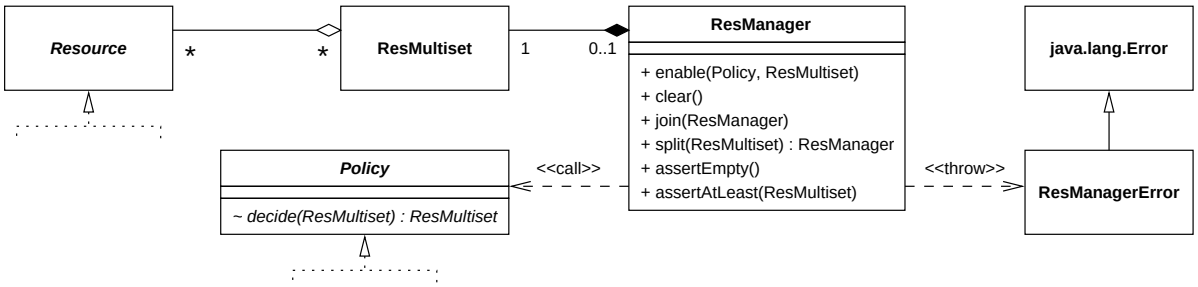

Figure 3.8: UML class diagram of the resource management API.

a general Java API for tracking and monitoring resources at run-time. Both approaches also aim at static guarantees of resource safety, rendering run-time checks unnecessary; Section 3.2 achieves this goal by static analysis whereas we accomplish it by a type system.

3.3.1 Monitoring External Resources in MIDP

This section presents a Java library for monitoring the use of external resources at run-time. The API introduces special objects, called *resource managers*, which encapsulate multisets of resources that a MIDlet may legally use (according to the user or a policy) and which are passed as arguments into instrumented MIDP methods that actually use the resources. These methods, *e.g.,* the method for sending text messages, check the resource manager before consuming the resources. If the required resources are not present, the instrumented methods abort the MIDlet with a run-time error. A paper [21] on this work has been accepted for publication.

Resource Management API

The core of the Java package for resource management consists of five classes, see the class diagram in Figure 3.8. The abstract class Resource serves as an abstract type for resources; actual resources (*e.g.,* the permission to send one text message to a given phone number) are instances of subclasses. The class ResMultiset provides modifiable multisets of resources (internally represented by hash tables), with the usual operations on multisets. The class ResManager implements resource managers, see below for details. The abstract class Policy serves as an abstract type for policies deciding which resources a MIDlet is granted; actual policies (*e.g.,* a dialogue letting the user choose which resources to grant) are coded as subclasses. The class ResManagerError serves to signal resource-related run-time errors.

The class ResManager encapsulates a multiset of resources via a private field rs of type ResMultiset. Its public methods are synchronised to avoid races in case different threads access the same resource manager. The table below lists the method signatures with a JML-style[2] semantics, where the symbols $\subseteq$, $\uplus$ and $\cap$ stand for multiset inclusion, sum and intersection, respectively.

	requires	**ensures**	**modifies**
ResManager()	*true*	this.rs = $\emptyset$	this.rs
void enable(Policy p, ResMultiset req)	*true*	this.rs $\uplus$ req = \old(this.rs) $\uplus$ \old(req) $\wedge$ req $\subseteq$ \old(req)	this.rs, req
void clear()	*true*	this.rs = $\emptyset$	this.rs
void join(ResManager mgr)	*true*	this.rs = \old(this.rs) $\uplus$ \old(mgr.rs) $\wedge$ mgr.rs = $\emptyset$	this.rs, mgr.rs
ResManager split(ResMultiset bound)	*true*	\fresh(\result) $\wedge$ \result.rs = \old(this.rs) $\cap$ bound $\wedge$ \result.rs $\uplus$ this.rs = \old(this.rs)	this.rs
void assertEmpty()	this.rs = $\emptyset$	*true*	\nothing
void assertAtLeast(ResMultiset bound)	bound $\subseteq$ this.rs	*true*	\nothing

[2]Here, unlike in JML, \old(e) refers to the pre-state of expression e in the pre-state of the heap.

```
1  void sendBulk(MessageConnection conn,
2                Message msg,
3                PhonebookEntry[] grp)
4  {
5    ResMultiset rs = new ResMultiset();
6    for (int i=0; i < grp.length; i++) {
7      String num = grp[i].getMobileNum();
8      rs.insert(new MsgResource(num), 1);
9    }
10
11   ResManager mgr = new ResManager();
12   mgr.enable(MsgUserPolicy.getPolicy(this), rs);
13
14   if (rs.isEmpty()) {
15     for (int i=0; i < grp.length; i++) {
16       String num = grp[i].getMobileNum();
17       msg.setAddress(num);
18       conn.send(mgr, msg);
19     }
20   }
21
22   mgr.assertEmpty();
23 }
```

```
1  public void send(ResManager mgr, Message msg)
2  throws IOException, InterruptedIOException
3  {
4    synchronized (msg) {
5      String num = msg.getAddress();
6      ResMultiset rs = new ResMultiset();
7      rs.insert(new MsgResource(num), 1);
8
9      ResManager local_mgr = mgr.split(rs);
10     local_mgr.assertAtLeast(rs);
11
12     try {
13       send(msg);
14       local_mgr.clear(); local_mgr = null;
15     } catch (InterruptedIOException e) {
16       local_mgr.clear(); local_mgr = null;
17       throw e;
18     } catch (IOException e) {
19       mgr.join(local_mgr); local_mgr = null;
20       throw e;
21     }
22   }
23 }
```

Figure 3.9: Bulk messaging example, left: MIDlet code, right: instrumented MIDP method.

The semantics of enable is non-deterministic because it is not statically known how many resources will be added to the manager's multiset rs and how many will be left in the argument req. At run-time, the actual decision is taken by the policy p, which decides which resources to add to the manager's multiset rs and which resources to deny. Upon return of enable, the MIDlet should check the argument req to learn which of the requested resources it is being denied; in particular, if req is empty then all of the requested resources have been granted.

The methods clear, split and join provide some control over the contents of a resource manager, by consuming all its resources, transferring some resources to a new manager, or joining the resources in two managers, respectively. Thanks to split and join, resource managers can be kept thread local, avoiding unnecessary synchronisation on shared managers.

The assertion methods check whether their preconditions hold. If so they behave like no-ops, otherwise they throw an instance of ResManagerError. The latter case must be seen as a violation of the MIDlet's own logic (much like failing an assertion), and the MIDlet should not be allowed attempts at repairing the situation (by catching the error), which is why ResManagerError extends java.lang.Error rather than java.lang.Exception.

Example: Bulk Messaging MIDlet

We illustrate the use of resource managers by an example application built on top of the *Wireless Messaging API* (WMA, current version 2.0 [99]), a bulk messaging MIDlet, which lets the user send a text message to group of recipients from his phone book; a similar scenario was described in Deliverable 1.1 [123]. Figure 3.9 (left-hand side) shows the MIDlet's method that actually sends the message. The method takes an (already open) message connection, a message and a group of recipients (represented as array of phone book entries). First, the MIDlet builds up a multiset of resources rs by iterating over the group of recipients and for each one, extracting the mobile phone number, converting it into a resource by constructing an instance of MsgResource (which is a subclass of Resource), and adding one occurrence of that instance to the multiset. Next, the MIDlet creates an empty resource manager mgr and enables it to use the resources in the multiset rs. Enabling requires a policy deciding which resources to grant. In this case, the policy is an instance of MsgUserPolicy (which is a subclass of Policy), the decide method of which pops up a dialogue box for the user to approve or deny the planned resource usage. Only if the user approves of all messages to be sent, *i.e.,* if enable leaves rs empty, does the code proceed to the actual send loop. The send loop again iterates over the group of recipients, extracting for each one the mobile phone number, setting the address field

of the message and sending the message using the instrumented send method, see below. After the loop, assertEmpty checks that the resource manager mgr is really empty, *i.e.,* all enabled resources have been used.

To monitor whether a MIDlet uses only granted resources, we have to wrap methods that consume resources with instrumentation code checking whether a given resource manager holds the required resources. In the case of text messaging, this requires wrapping the WMA method send(Message) as shown in Figure 3.9 (right-hand side). The wrapper method send(ResManager, Message) extracts the phone number num from the message and constructs a multiset rs containing a single occurrence of the resource corresponding to num. Then it splits the resources in rs off from the resource manager mgr and stores them in the new local resource manager local_mgr, which is checked for containing at least the resources in rs. If this check fails a ResManagerError will be thrown, aborting the calling MIDlet; if the check succeeds we know that local_mgr holds exactly the resources in rs. Finally, the message is actually sent by calling the uninstrumented send method. Clearing local_mgr and nulling the reference afterwards is not strictly necessary but considered good practise; it signals that the resources in the local manager are now used up and that the manager itself is ready to be reclaimed by garbage collection.

In case of a send failure, the event that actually spends the resources (*i.e.,* delivering the text message to the operator's network) may or may not have happened yet. We assume that an IOException is thrown before actually sending the message (*e.g.,* because the connection to the operator's network is down), so the resources are not yet consumed, and the handler can return them to the caller (by joining the local manager to mgr) before propagating the exception. However, if an InterruptedIOException is raised, we do not know whether the send event has already happened, so we assume that the resources are already spent. In this case, the handler consumes the resources (by clearing the local manager) before propagating the exception.

Note that the instrumented send method method must synchronise on msg, which is accessed twice, but there is no need to synchronise on mgr (for there are no data dependencies between the first and second access) or on **this** (for it is accessed only once).

Secure Deployment of the Resource Management Library

The resource management library provides a pure Java solution to the problem of monitoring external resources; in particular, it does not require any changes to the JVM or to MIDP classes, so it can be deployed, as a trusted library, to any platform conforming to MIDP2. Unfortunately, MIDP does not let the user install additional libraries, which is why the resource management library (including subclasses representing concrete resources (*e.g.,* MsgResource) and policies (*e.g.,* MsgUserPolicy), and including instrumented methods (*e.g.,* send(ResManager, Message))) has to be shipped as part of the MIDlet using it.

Trust in the library is established by having the MIDlet signed by the network operator. Of course, the network operator will not sign the MIDlet unless it meets the following requirements:

- The binaries comprising the resource management library are identical to network operator's own trusted implementation.
- The MIDlet uses the resource management API correctly (for instance, does not bypass the instrumentation, or subvert the hash-based implementation of multisets).

Thanks to the limitations of MIDP (in particular, no reflection, no custom class loading), both properties can be checked statically by inspecting the MIDlet's JAR.

Properties of the Resource Management Library

By design, the resource management library is extensible to accommodate new policies and resources. New policies (*e.g.,* for automatically granting text messages to national numbers) are added as new subclasses of Policy. Adding new resources (*e.g.,* the space available in the persistent record store) amounts to adding new resource types (by subclassing Resource) plus adding the appropriate instrumentation. Besides being extensible, the library also has the following security properties.

1. *MIDlets using the resource management API cannot consume more resources than granted; any attempt to do so will result in the MIDlet being aborted before the abuse happens.*

   On the one hand, this property is ensured by the instrumentation, which checks a resource manager for sufficient resources *before* calling the method that really spends the resources. On the other hand, this property relies on the fact that policies can't be bypassed. That is, there is no way to sneak new resources into the managers other than by calling enable, in which case a policy gets to decide which resources to grant and which to deny.

2. *If a MIDlet is resource safe then erasing the resource managers does not change its observable behaviour.*

   We call a MIDlet is *resource safe* if it does not fail resource assertions, *i.e.,* if it cannot ever throw a ResManagerError. Later, we will present a type system for statically certifying resource safety (albeit not of Java MIDlets).

   Erasing resource managers is possible, for by the resource management API, the value of a resource manager can only affect the values of other resource managers; it cannot affect the values of other types.

   Technically, erasure can be achieved by retaining the public interface of class ResManager but replacing its implementation with a stateless dummy implementation. More precisely, erasure removes the private field rs (storing the manager's multiset), which turns all public methods into no-ops, except for split and enable. The latter still calls the policy and reports the denied resources back to the MIDlet, whereas the former creates a fresh (erased) manager. Thus, erasure removes most of the run-time overhead induced by resource managers, which is an advantage particularly on small devices.

3. *If a MIDlet is resource safe then its resource managers do not leak information.*

   This information-flow safety property is a corollary of erasure. If a MIDlet is resource safe, the resource managers can be erased without changing the MIDlet's observable behaviour. Yet, erased resource managers are stateless, so they cannot leak information. Hence, no leakage is observable.

3.3.2 A Type System for Resource Safety

This section presents a type system for statically certifying resource safety, *i.e.,* the absence of run-time errors triggered by the abuse of resources. The initial version of the type system presented here does not cover the full Java language but is targeted at a simpler procedural programming language. Yet, language and type system are expressive enough to type non-trivial programs, like the bulk messaging example shown in the previous section. More details on this work can be found in a technical report [110].

Syntax and Semantics of a Language for Resource Managers

This subsection introduces a simple procedural programming language. with built-in constructs for handling resource managers. The operations on resource managers are essentially the same as the methods of the Java class ResManager. Note that the language can be translated to a fragment of Java (using static methods only) in a straightforward way.

Syntax. A program is a collection of procedures, where each procedure consists of a name, declaration of input and output parameters, declaration of local variables and a statement.

Statements are composed by conditionals and sequencing from primitive statements like assignments $y := e$ of expression e to variable y, or procedure calls $p(x_1, \ldots, x_m \mid y_1, \ldots, y_n)$, where the variables x_i and y_j are the actual input and output parameters, respectively, of procedure p. The language does not provide loop constructs, iteration is done by recursion like in GRAIL [19]. By $def(s)$, we denote the set of variables defined in a statement s, *i.e.,* the variables occurring on the left-hand side of an assignment or among the output parameters of a procedure call.

Expressions are built from constants, variables and operators according to their data types. Besides the unit type and the types of Booleans, integers and strings (with the usual operators), the language features

- extensible array types (with the usual query and update operators),
- a type **res** of *resources* (which can be constructed from strings and compared for equality),
- a type **mres** of *multisets* of resources (with the usual operators), and
- a type **mgr** of *resource managers* (with no operators at all).

Since there are no operators on resource managers, the language provides built-in procedures to access them, inspired by the methods of the Java class ResManager.

- **enable**$(m, r \mid m', r')$ tries to top up the manager m with a multiset of requested resources r. It returns the new manager m' holding the granted multiset of resources and the multiset r' of resources that have been denied, *i.e.,* the multisets in m' and r' sum up to m and r.
- **clear**$(\mid m')$ creates an empty manager m'.
- **join**$(m_1, m_2 \mid m')$ adds the multisets held by the managers m_1 and m_2 and stores their sum in manager m'.
- **split**$(m, r \mid m'_1, m'_2)$ splits the multiset held by manager m and distributes it to the managers m'_1 and m'_2 such that m'_2 gets the largest possible submultiset of r.
- **assertEmpty**$(m \mid m')$ checks whether the manager m is empty. If not aborts with a run-time error, otherwise copies the manager m to the manager m'.
- **assertAtLeast**$(m, r \mid m')$ checks whether the multiset r is contained in the multiset held by the manager m. If not aborts with a run-time error, otherwise copies the manager m to the manager m'.

The language imposes three syntactic restrictions on programs.

- Expression evaluation occurs only in assignments. I. e., conditions in if-statements and input parameters in procedure calls must be variables. This restriction simplifies the operational semantics since expression evaluation can fail only in assignments.
- All statements are in SSA form, *i.e.,* each variable is defined only once. Note that this implies the absence of input or output aliasing in procedure calls $p(x_1, \ldots, x_m \mid y_1, \ldots, y_n)$, *i.e.,* the variables x_i and y_j are all different. This restriction simplifies the effect type system presented later, for assignments behave like let bindings in a functional language (*e.g.,* GRAIL [19]).
- All resource managers are linear, *i.e.,* used at most once. This restriction is motivated by the nature of resource managers, which are stateful objects. As the language does not feature objects, each state of a resource manager has to be realised by its own variable. The linearity restriction enforces that a we cannot re-use a previously used state, *e.g.,* we cannot double our resources by joining the same managers twice as in **join**$(m_1, m_2 \mid m')$ and **join**$(m_1, m_2 \mid m'')$.

Operational semantics. We present the operational semantics of our programming language as a big-step state transformer. A state β is either the error state (denoted by $\bot$) or a value environment mapping variables x to their values $\beta(x)$. Since variables are typed by a type environment Γ mapping variables x to their types $\Gamma(x)$, states have to respect typing, *i.e.,* the value $\beta(x)$ must be of type $\Gamma(x)$ for all variables x. Given a non-error state β, a variable x and a value v', we write $\beta[x \mapsto v']$ to denote the non-error state β' which assigns v' to x and $\beta(y)$ to all variables $y \neq x$. In the following, we will use the implicit convention that β refers to states whereas α refers to non-error states.

Big-step semantics of statements $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$

$$\text{(OS-}\bot\text{)}\ \frac{}{\Pi, \Gamma \vdash s \triangleright \bot \rightsquigarrow \bot} \qquad \text{(OS-skip)}\ \frac{}{\Pi, \Gamma \vdash \mathbf{skip} \triangleright \alpha \rightsquigarrow \alpha}$$

$$\text{(OS-if-then)}\ \frac{\alpha(z) = \mathit{true} \qquad \Pi, \Gamma \vdash s_1 \triangleright \alpha \rightsquigarrow \beta'}{\Pi, \Gamma \vdash \mathbf{if}\ z\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \triangleright \alpha \rightsquigarrow \beta'} \qquad \text{(OS-if-else)}\ \frac{\alpha(z) = \mathit{false} \qquad \Pi, \Gamma \vdash s_2 \triangleright \alpha \rightsquigarrow \beta'}{\Pi, \Gamma \vdash \mathbf{if}\ z\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \triangleright \alpha \rightsquigarrow \beta'}$$

$$\text{(OS-seq)}\ \frac{\Pi, \Gamma \vdash s_1 \triangleright \alpha \rightsquigarrow \beta' \qquad \Pi, \Gamma \vdash s_2 \triangleright \beta' \rightsquigarrow \beta''}{\Pi, \Gamma \vdash s_1\ ;\ s_2 \triangleright \alpha \rightsquigarrow \beta''} \qquad \text{(OS-assign)}\ \frac{\Gamma, \alpha \vdash e \downarrow v \qquad \alpha' = \alpha[y \mapsto v]}{\Pi, \Gamma \vdash y := e \triangleright \alpha \rightsquigarrow \alpha'}$$

$$\text{(OS-call-}\bot\text{)}\ \frac{\begin{array}{c}\Pi(p) = p(x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n)\{z_1{:}\kappa_1, \ldots, z_l{:}\kappa_l;\ s_p\} \\ \alpha_p \text{ is a } p\text{-state with } \alpha_p(x_1) = \alpha(u_1), \ldots, \alpha_p(x_m) = \alpha(u_m) \\ \Pi, \Gamma_p \vdash s_p \triangleright \alpha_p \rightsquigarrow \bot\end{array}}{\Pi, \Gamma \vdash p(u_1, \ldots, u_m \mid v_1, \ldots, v_n) \triangleright \alpha \rightsquigarrow \bot}$$

$$\text{(OS-call)}\ \frac{\begin{array}{c}\Pi(p) = p(x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n)\{z_1{:}\kappa_1, \ldots, z_l{:}\kappa_l;\ s_p\} \\ \alpha_p \text{ is a } p\text{-state with } \alpha_p(x_1) = \alpha(u_1), \ldots, \alpha_p(x_m) = \alpha(u_m) \\ \Pi, \Gamma_p \vdash s_p \triangleright \alpha_p \rightsquigarrow \alpha'_p \qquad \alpha' = \alpha[v_1 \mapsto \alpha'_p(y_1)] \ldots [v_n \mapsto \alpha'_p(y_n)]\end{array}}{\Pi, \Gamma \vdash p(u_1, \ldots, u_m \mid v_1, \ldots, v_n) \triangleright \alpha \rightsquigarrow \alpha'}$$

$$\text{(OS-}q\text{-}\bot\text{)}\ \frac{\begin{array}{c}\Pi(q) = q(x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n) \\ \alpha_q \text{ is a } q\text{-state with } \alpha_q(x_1) = \alpha(u_1), \ldots, \alpha_q(x_m) = \alpha(u_m) \qquad \alpha_q \not\models \Phi_q\end{array}}{\Pi, \Gamma \vdash q(u_1, \ldots, u_m \mid v_1, \ldots, v_n) \triangleright \alpha \rightsquigarrow \bot}$$

$$\text{(OS-}q\text{)}\ \frac{\begin{array}{c}\Pi(q) = q(x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n) \\ \alpha_q \text{ is a } q\text{-state with } \alpha_q(x_1) = \alpha(u_1), \ldots, \alpha_q(x_m) = \alpha(u_m) \qquad \alpha_q \models \Phi_q \\ \alpha_q \models \Psi_q \qquad \alpha' = \alpha[v_1 \mapsto \alpha_q(y_1)] \ldots [v_n \mapsto \alpha_q(y_n)]\end{array}}{\Pi, \Gamma \vdash q(u_1, \ldots, u_m \mid v_1, \ldots, v_n) \triangleright \alpha \rightsquigarrow \alpha'}$$

Preconditions Φ_q and effects Ψ_q of built-ins $\Pi(q)$

$\Pi(q)$	Φ_q	Ψ_q
enable(m:mgr, r:mres | m′:mgr, r′:mres)	*true*	$\mathbf{m'} \uplus \mathbf{r'} = \mathbf{m} \uplus \mathbf{r} \wedge \mathbf{r'} \subseteq \mathbf{r}$
clear(| m′:mgr)	*true*	$\mathbf{m'} = \emptyset$
join(m₁:mgr, m₂:mgr | m′:mgr)	*true*	$\mathbf{m'} = \mathbf{m_1} \uplus \mathbf{m_2}$
split(m:mgr, r:mres | m₁′:mgr, m₂′:mgr)	*true*	$\mathbf{m_1'} \uplus \mathbf{m_2'} = \mathbf{m} \wedge \mathbf{m_2'} = \mathbf{m} \cap \mathbf{r}$
assertEmpty(m:mgr | m′:mgr)	$\mathbf{m} = \emptyset$	$\mathbf{m'} = \mathbf{m}$
assertAtLeast(m:mgr, r:mres | m′:mgr)	$\mathbf{r} \subseteq \mathbf{m}$	$\mathbf{m'} = \mathbf{m}$

Figure 3.10: Operational semantics.

The big-step state transformer semantics of a statement s is denoted by the judgement $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$, where Π is a program, Γ a type environment (for s) and β and β' are the pre- resp. post-states of the execution of s. Figure 3.10 displays the rules defining the semantics.

The rule (OS-$\bot$) propagates the error state. (OS-skip), (OS-if-then), (OS-if-else) and (OS-seq) are the standard rules for the no-op, conditional and sequencing constructs of the language. Note that the condition z is a Boolean variable rather than an arbitrary Boolean expression. (OS-assign) evaluates the expression e to a value v in the pre-state (denoted by the judgement $\Gamma, \alpha \vdash e \downarrow v$, which is defined by the usual rules for strict evaluation) and updates the variable y to v in the post-state. Note that the semantics gets stuck if an expression cannot be evaluated (*e.g.,* if evaluation would raise an exception like division by zero, or array index out of bounds). (OS-call) and (OS-call-$\bot$) call a procedure p with actual input and output parameters u_i and v_j (all of which are variables). Execution of the procedure body s starts in a p-state α_p (*i.e.,* a state storing the values of the parameters and local variables of procedure p and complying with the type environment Γ_p) with the caller's input parameters u_i copied to the callee's input parameters x_i. In the case of (OS-call-$\bot$), execution of the callee aborts with an error, which is propagated to the caller. In the case of (OS-call), execution of the callee terminates normally in a p-state α'_p, and the callee's output parameters y_j are copied back to the caller's output parameters v_j. The rule schemas (OS-q) and (OS-q-$\bot$) are dealing with the built-ins $q = \mathbf{enable}, \mathbf{clear}, \mathbf{join}, \mathbf{split}, \mathbf{assertEmpty}, \mathbf{assertAtLeast}$. Rule (OS-$q$) executes a call to q by creating a q-state α_q (*i.e.,* a state storing the values of the parameters of q), copying the values of the caller's input parameters to the callee's input parameters, checking the precondition Φ_q, checking the effect Ψ_q, and copying the values of the callee's output parameters back to the caller's output parameters. In other words, calling a built-in $q(u_1, \ldots, u_m \mid v_1, \ldots, v_n)$ means choosing values for its output parameters such that the effect is satisfied; this choice may be non-deterministic (and in fact, it is for $q = \mathbf{enable}$). Rule (OS-q-$\bot$) (which is deterministic) covers the case when the precondition of the built-in q fails to hold, which can only happen for $q = \mathbf{assertEmpty}, \mathbf{assertAtLeast}$. The built-ins mimic the methods of the Java class ResManager, with the exception that policies are abstracted away. Instead, **enable** nondeterministically chooses which resources to grant and which to deny.

Monotonicity of resource usage. To justify our modelling of resource managers (which in Java are represented as stateful objects) as stateless variables subject to linearity restrictions, we show Theorem 3.3.1 stating that the sum of resources held in all resource managers in the system is going to decrease in any execution which does not call **enable**. To formally express this statement, we define the sum of resources in the system *before* and *after* the execution of a statement s. Let Γ be a type environment for s and let β and β' be the pre- resp. post-states of an execution of s. By $pre^s_\Gamma(\beta)$, we denote the multiset sum of resources over all resource managers in β except those that will be defined in s (since these may have bogus values in the pre-state β). By $post^s_\Gamma(\beta')$, we denote the sum of resources over all resource managers in β' except those that have been used in s (since their values are inaccessible in the post-state β').

Theorem 3.3.1. *Let Π be a program, let Γ a type environment for a statement s, and let β and β' be states. If Δ_{OS} is a derivation of $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$ which does not call* **enable** *(i.e., the rule (OS-*$\mathbf{enable}$*) does not occur in Δ_{OS}) then $pre^s_\Gamma(\beta) \supseteq post^s_\Gamma(\beta')$.*

Informally, this theorem assures us that resource managers can only hold resources that have previously been approved by **enable**, *i.e.,* the policy behind **enable** cannot be bypassed. However, it does not guarantee that the resources held in managers will be sufficient so that no run-time errors can occur when calling **assertAtLeast**.

Effect Types

Run-time errors occur because the preconditions of the built-in procedures **assertEmpty** and **assertAtLeast** fail to hold. Certifying the absence of run-time errors thus calls for a program logic (*e.g.,* a Hoare logic) or a type system that can track preconditions. In this subsection, we present such a type system for inferring preconditions and effects of statements.

Typing of statement effects $\Pi, \Theta, \Gamma \vdash s : \Phi \to \Psi$

$$\text{(T-weak)}\ \frac{\Pi, \Theta, \Gamma \vdash s : \Phi \to \Psi}{\Pi, \Theta, \Gamma \vdash s : \hat{\Phi} \to \hat{\Psi}}\ \text{if}\ \begin{cases} \hat{\Phi} \models \Phi \wedge \\ (\hat{\Phi} \wedge \Psi) \models \hat{\Psi} \end{cases}$$

$$\text{(T-skip)}\ \frac{}{\Pi, \Theta, \Gamma \vdash \mathbf{skip} : \mathit{true} \to \mathit{true}} \qquad \text{(T-assign)}\ \frac{}{\Pi, \Theta, \Gamma \vdash y := e : \mathit{true} \to y = e}$$

$$\text{(T-call)}\ \frac{}{\Pi, \Theta, \Gamma \vdash p(x'_1, \ldots, x'_m \mid y'_1, \ldots, y'_n) : \Phi' \to \Psi'}\ \text{if } (*)$$

$$\text{where } (*)\ \begin{cases} \Pi(q) = p(x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n)[\{\ldots\}] \wedge \\ \Theta(p) = \Phi \to \Psi \end{cases}$$

$$\text{(T-seq)}\ \frac{\Pi, \Theta, \Gamma \vdash s_1 : \Phi \to \Psi_1 \qquad \Pi, \Theta, \Gamma \vdash s_2 : \Phi \wedge \Psi_1 \to \Psi_2}{\Pi, \Theta, \Gamma \vdash s_1 \,;\, s_2 : \Phi \to \Psi_1 \wedge \Psi_2}$$

$$\text{(T-if)}\ \frac{\Pi, \Theta, \Gamma \vdash s_1 : z \wedge \Phi \to \Psi \qquad \Pi, \Theta, \Gamma \vdash s_2 : \neg z \wedge \Phi \to \Psi}{\Pi, \Theta, \Gamma \vdash \mathbf{if}\ z\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 : \Phi \to \Psi}$$

Typing of procedures effects $\Pi, \Theta \vdash p$

$$\text{(T-proc)}\ \frac{\Pi, \Theta, \Gamma_p \vdash s : \Phi \to \Psi}{\Pi, \Theta \vdash p}\ \text{if}\ \begin{cases} \Pi(p) = p(x_1{:}\tau_1, \ldots, x_m{:}\tau_m \mid y_1{:}\sigma_1, \ldots, y_n{:}\sigma_n)\{z_1{:}\kappa_1, \ldots, z_l{:}\kappa_l;\ s\} \wedge \\ \Theta(p) = \Phi \to \Psi \end{cases}$$

Figure 3.11: Typing rules for effect types.

Types and typing judgement. Effect types are built from constraints, where the constraint language is a typed first-order logic. More specifically, the constraint language is the expressions of the programming language augmented with the standard quantifiers of first-order logic. Moreover, the constraint language identifies the programming language types **mres** and **mgr**, thus giving up the distinction between a resource manager and the multiset of resources held by that manager.

Given two constraints Φ and Ψ, we call $\Phi \to \Psi$ an *effect type*; we call the constraints Φ and Ψ *precondition* and *effect*, respectively. $\Phi \to \Psi$ is an effect type for a statement s if $\mathit{free}(\Phi) \cap \mathit{def}(s) = \emptyset$, *i.e.,* if the precondition does not constrain any variables that are to be defined during the execution of s. $\Phi \to \Psi$ is an effect type for a procedure p if $\mathit{free}(\Phi) \subseteq \{x_1, \ldots, x_m\}$ and $\mathit{free}(\Psi) \subseteq \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$, where $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ are the procedure's input and output parameters, respectively. Thus, the precondition may only constrain the input parameters, whereas the effect may constrain all parameters but no local variables.

Let Π be a program and Θ an *effect environment* (*i.e.,* Θ maps procedures p to effect types $\Theta(p)$ such that $\Theta(p)$ is an effect type for p). $\Pi, \Theta, \Gamma \vdash s : \Phi \to \Psi$ is an *effect typing judgement* (for statements) if s a statement, Γ a type environment for s and $\Phi \to \Psi$ an effect type for s. $\Pi, \Theta \vdash p$ is an *effect type checking judgement* (for procedures) if p is a procedure in Π.

Typing rules. Figure 3.11 displays the typing rules for effect types. The typing rules for statement effects fall into three groups, namely a weakening rule, five rules for the five statement constructors, and a rule for procedures.

The weakening rule (T-weak) admits to weaken an effect type by strengthening the precondition and weakening the effect; note that due to the new type $\hat{\Phi} \to \hat{\Psi}$ being effect type for the statement s there is an implicit extra side condition $\mathit{free}(\hat{\Phi}) \cap \mathit{def}(s) = \emptyset$.

According to the rules for the statement constructors, the effect type for a **skip** is trivial. The effect of an assignment $y := e$ is the equation $y = e$; note that y cannot occur in e thanks to the SSA restriction. For a call $p(x'_1, \ldots, x'_m \mid y'_1, \ldots, y'_n)$, the effect type $\Phi' \to \Psi'$ is derived from the type $\Theta(p) = \Phi \to \Psi$ of the callee p (which could be a procedure or a built-in) by substituting the actual input and output parameters x'_i and

y'_j for the formal parameters x_i and y_j in Φ and Ψ. The rule (T-seq) requires that the precondition of the second statement entails the effect of the first, and if so the precondition of the composition s_1 **;** s_2 is the precondition of s_1 whereas its effect is the conjunction of the effects of s_1 and s_2; note that the soundness of (T-seq) relies on the SSA restriction. Reading the rule (T-if) backwards, it requires that both branches of a conditional type with essentially the same effect type, except that the then-branch gets to assume the branching condition z positively (in its precondition), the else-branch negatively.

Finally, the rule (T-proc) checks that the body of a procedure p types with respect to the effect type $\Phi \to \Psi$ assigned by the environment Θ; note that due to $\Phi \to \Psi$ being an effect type for the procedure p there are two implicit extra side conditions $\mathit{free}\Phi \subseteq \{x_1, \ldots, x_m\}$ and $\mathit{free}\Psi \subseteq \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$.

Type soundness. We call an effect environment Θ *admissible* for a program Π, denoted by $\Pi \vdash \Theta$, if

- for all procedures $p \in \Pi$, we have $\Pi, \Theta \vdash p$, and
- for all built-ins $q \in \Pi$, we have $\Theta(q) = \Phi_q \to \Psi_q$, where preconditions Φ_q and effects Ψ_q are taken form the table in figure 3.10.

Theorem 3.3.2 below states that for a statement s of effect type $\Phi \to \Psi$, if a terminating execution of s is starting in a non-error state β satisfying the precondition Φ then (1) the final state β' is not the error state, and (2) the final state satisfies the effect Ψ (and the precondition Φ thanks to the SSA restriction). Mostly, (1) is important to us as it means that typable statements are *resource safe*, *i.e.,* safe from run-time errors (as long as the initial state satisfies the precondition) caused by the built-ins **assertEmpty** and **assertAtLeast**. (2) associates the effect Ψ with a Hoare-style postcondition, however, note that Ψ may mention variables that were used (and not defined, thanks to the SSA restriction) in s, so Ψ really may specify an input-output relation (or effect). Here, SSA form simplifies the handling of effects, as it saves the introduction of ghost variables for storing input values.

Theorem 3.3.2. *Let Π be a program, let Θ be an effect environment, let Γ a type environment for a statement s, let $\Phi \to \Psi$ be an effect type for s, and let β and β' be states. If $\Pi \vdash \Theta$ and $\Pi, \Theta, \Gamma \vdash s : \Phi \to \Psi$ and $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$ and $\beta \neq \bot$ and $\beta \models \Phi$ then (1) $\beta' \neq \bot$ and (2) $\beta' \models \Phi \wedge \Psi$.*

The type soundness theorem attributes the quality of being safe from run-time errors only to terminating executions. This is due to the big-step operational semantics, which ignores non-terminating executions. However, it is not difficult to define a compatible small-step semantics and prove that non-terminating executions cannot raise run-time errors (as raising an error would quickly terminate the execution).

Erasing Resource Managers

Recall that the operations on resource managers do not admit to observe the contents of a resource manager other than by calling **assertEmpty** or **assertAtLeast**, which may or may not abort the program with a run-time error. Consequently, the resource managers of a resource safe program cannot be observed. In fact, resource safe programs do not change their behaviour if resource managers are erased.

Technically, erasure is achieved by mapping the manager type to the unit type, *i.e.,* τ°, the erasure of a type τ, is defined as follows.

$$\tau^\circ = \begin{cases} \mathbf{unit} & \text{if } \tau = \mathbf{mgr} \\ \tau & \text{otherwise} \end{cases}$$

Erasure on types determines Π°, the erasure on a program Π, which is defined by applying the type erasure to all variable declarations of procedures and built-ins. Similarly, erasure on types determines β°, the erasure on a state β, which preserves the error state (*i.e.,* $\bot^\circ = \bot$) but transforms non-error states α by mapping the values of **mgr**-variables (*i.e.,* variables of type **mgr** in the type environment Γ) to $\star$, the value of the **unit** type.

$$\alpha^\circ(x) = \begin{cases} \star & \text{if } \Gamma(x) = \mathbf{mgr} \\ \alpha(x) & \text{otherwise} \end{cases}$$

Recall the state-transformer semantics for the built-in procedures q from Figure 3.10, logically expressed as a precondition Φ_q and an effect Ψ_q. Applying erasure to pre- and post-states determines a new state-transformer semantics. Logically, this is equivalent to existentially quantifying the **mgr**-variables in precondition and effect. As a result, the new semantics trivialises all built-in procedures $q \neq$ **enable**, *i.e.,* both precondition Φ_q and effect Ψ_q are just *true*. For $q =$ **enable**, we get precondition $\Phi_q = true$ and effect $\Psi_q = \mathbf{r'} \subseteq \mathbf{r}$, *i.e.,* **enable** still gets to decide which of the requested resources to grant and which to deny, yet it does not record its decision in a resource manager.

As a consequence of erasure, the preconditions of all built-ins are *true*, hence they can no longer cause any run-time errors. Moreover, for resource safe programs, the executions of the original program are the same as the executions of the erased one, up to erasure on states.

Theorem 3.3.3. *Let* Π *be a program, let* Γ *a type environment for a statement* s*, and let* β *and* β' *be states. If* s *is resource safe then* $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$ *iff* $\Pi^\circ, \Gamma^\circ \vdash s \triangleright \beta^\circ \rightsquigarrow \beta'^\circ$.

Since type soundness (Theorem 3.3.2) guarantees resource safety, resource managers can be erased from typable programs. As a corollary, no information can leak from resource managers, for if that could happen then erasing the resource managers should really change the executions. In other words, as a consequence of erasure, resource managers are information flow safe.

3.3.3 Related Work

Several frameworks have been proposed for enhancing Java with run-time monitoring of resource consumption, for example JRes [62], J-Seal [46] and J-RAF [96]. These frameworks monitor specific resources (CPU, memory, network bandwidth, threads), relying on instrumentation of either the JVM (for CPU time), low level system classes (for memory and network bandwidth) and the bytecode itself (for memory and instruction counting). Where our resource management library is designed to enforce security, these frameworks were developed to support resource aware applications, which can adapt their behaviour in response to resource fluctuation, for example by trading precision for time (by returning an imprecise result to meet a deadline), or time for memory (by caching less to reduce memory consumption).

Nanevski et al. [134] describe a type theory which contains a type for Hoare triples and typing rules for reasoning with Hoare triples. However, they do not consider resources but instead focus on reasoning about heap-manipulating programs and higher-order functions.

Similar to our approach, Chander et al. [55] uses a regime of reserving resources before their actual use. They require the programmer (or program optimiser) to annotate the program with primitives `acquire` and `consume`, loop invariants and function pre- and postconditions. To statically check that the program does not consume more resources than acquired, they generate verification conditions and feed them to a theorem prover. Their approach also admits a policy deciding at run-time whether a request to acquire resources should succeed. However, programs cannot recover from failure to acquire resources. In contrast, our system admits a program to determine to which extent a request for resources has been successful, and to react accordingly. Also, our system can track an unbounded and input-dependent number of different resources, whereas [55] only deals with one fixed resource.

3.3.4 Future Work

Integration of our type system with the MOBIUS logic [124] will be done in the style of the MRG project. This requires translating our effect type judgements $\Pi, \Theta, \Gamma \vdash s : \Phi \to \Psi$ into MOBIUS logic judgements of the form $\mathsf{G}, \mathsf{Q} \vdash \{A\} pc \{B\}(I)$. Thereby, the translation will map effect environment Θ to proof context G, precondition Φ to precondition A and effect Ψ to postcondition B. The MOBIUS logic assumes that a byte code program is implicitly given (as a mapping from labels to instructions), so we will also require a translation of the program Π into byte code. Note that our type system does not deal with local assertions, so the local annotation table Q will be empty. Also, our system provides no guarantees for non-terminating computations, so the invariant I will be true. Using the translation, we will map effect type derivations

into derivations of the MOBIUS logic, where the effect type rules will be mimicked by corresponding derived rules in the logic.

Work on this type system will be continued in Task 2.4. Besides extending the system to cover a larger fragment of Java we will work on the automation of type checking and type inference. Since type checking and inference involve proving validity of first-order formulae (in the side condition of the weakening rule), the focus will be on the application of state-of-the-art theorem proving technology, as is also used to check verification conditions Workpackage 4. Note that the type system has been developed with theorem proving support in mind; the theories involved (integers, multisets, arrays) are supported (more or less) by most modern automated SMT provers.

Our type system presupposes type annotations for all procedures. To relieve the annotation burden on the programmer, we will investigate (necessarily incomplete) approaches to type inference. Here, INRIA's static analysis (see Section 3.2) will help to gather enough information for synthesising effect types.

3.4 Cost Analysis of Java Bytecode

Cost analysis has been intensively studied in the context of declarative (see, e.g., [148, 144, 161, 82, 35] for functional programming and [**?**, 64] for logic programming) and *high-level* imperative programming languages (mainly focused on the estimation of worst case execution times and the design of cost models [176]). Traditionally, cost analysis has been formulated at the source level. However, there are situations where we do not have access to the source code, but only to compiled code. An example of this is *mobile code*, where the *code consumer* receives code to be executed. In this context, Java bytecode is widely used, mainly due to its security features and the fact that it is *platform-independent*. Automatic cost analysis has interesting applications in this context. For instance, the receiver of the code may want to infer cost information in order to decide whether to reject code which has too large cost requirements in terms of computing resources (in time and/or space), and to accept code which meets the established requirements [59, 88]. In fact, this is the main motivation for the *Mobile Resource Guarantees* (MRG) research project [20], which establishes a *Proof-Carrying Code* [138] framework for guaranteeing resource consumption. Also, in parallel systems, knowledge about the cost of different procedures can be used in order to guide the partitioning, allocation and scheduling of parallel processes.

The aim of this part is to develop an automatic approach to the cost analysis of Java bytecode [8, 11, 9, 10] which statically generates *cost relations*. These relations define the cost of a program as a function of its input data size. This approach was proposed by Debray and Lin [**?**] for logic programs, and by Rabhi and Manson [144] for functional programs. In these approaches, cost functions are expressed by means of *recurrence equations* generated by abstracting the recursive structure of the program and by inferring size relations between arguments. A low-level object-oriented language such as Java bytecode introduces novel challenges, mainly due to: 1) its unstructured control flow, e.g., the use of goto statements rather than recursive structures; 2) its object-oriented features, like virtual method invocation, which may influence the cost; and 3) its stack-based model, in which stack cells store intermediate values. This paper addresses these difficulties and develops a generic framework for the automatic cost analysis of Java bytecode programs. The process takes as input the bytecode corresponding to a method and yields a cost relation after performing these steps:

1. The input bytecode is first transformed into a *control flow graph* (CFG). This allows making the unstructured control flow of the bytecode explicit (challenge 1 above). Advanced features like virtual invocation and exceptions are simply dealt as additional nodes in the graph (challenge 2).

2. The CFG is then represented as a set of rules by using an *intermediate recursive representation* in which we *flatten* the local stack by converting its contents into a series of additional local variables (challenge 3).[3]

[3] Note that this is possible since in every *valid* bytecode program the height of the local stack at each program point is fixed and therefore can be computed statically.

3. In the third step, we infer *size relations* among the input variables for all calls in the rules by means of static analysis. These size relations are constraints on the possible values of variables (for integers) and constraints on the length of the longest reachable path (for references).

4. The fourth phase provides, for each rule of the recursive representation, a safe approximation of the set of input arguments which are "relevant" to the cost. This is performed using a simple static analysis.

5. From the recursive representation, its relevant arguments, and the size relations, the fifth step automatically yields as output the *cost relation* which expresses the cost of the method as a function of its input arguments.

We point out that computed cost relations, in many cases, can be simplified to the point of deriving statically an *upper and lower* threshold cost for the input size arguments and/or obtaining a *closed form* solution. Such simplifications have been well-studied in the field of algorithmic complexity (see e.g. [175]).

3.4.1 The Java Bytecode Language

Java bytecode is a low-level object-oriented programming language with unstructured control and an *operand stack* to hold intermediate computational results. Moreover, objects are stored in dynamic memory (the *heap*). A Java bytecode program consists of a set of *class files*, one for each class or interface. A class file contains information about its *name* $c \in Class\_Name$, the class it extends, the interfaces it implements, and the fields and methods it defines. In particular, for each method, the class file contains: a method signature $m \in Meth\_Sig$ which consists of its name $name(m) \in Meth\_Name$ and its type $type(m) = \tau_1, \ldots, \tau_n \rightarrow \tau \in Meth\_Type$ where $\tau, \tau_i \in Type$; its bytecode $bc_m = \langle pc_0{:}b_0, \ldots, pc_{n_m}{:}b_{n_m} \rangle$, where each b_i is a *bytecode instruction* and pc_i is its address; and the method's exceptions table. When it is clear from the context, we omit bytecode addresses and refer to a *method signature* as *method*.

In this part we consider a subset of the JVM language which is able to handle operations on integers, object creation and manipulation (by accessing fields and calling methods) and exceptions (either generated by abnormal execution or explicitly thrown by the program). We omit interfaces, static fields and methods and primitive types different from integers. Methods are assumed to return an integer value. Thus, our bytecode instruction set (*bcInst*) is:

$$
\begin{array}{l}
bcInst ::= \mathsf{push}\ x \mid \mathsf{istore}\ v \mid \mathsf{astore}\ v \mid \mathsf{iload}\ v \mid \mathsf{aload}\ v \mid \mathsf{iconst}\ a \mid \mathsf{iadd} \mid \mathsf{isub} \mid \mathsf{imul} \\
\quad \mid \mathsf{idiv} \mid \mathsf{if}\diamond\ pc \mid \mathsf{goto}\ pc \mid \mathtt{new} Class\_Name \mid \mathsf{invokevirtual}\ Class\_NameMeth\_Sig \\
\quad \mid \mathsf{invokespecial}\ Class\_NameMeth\_Sig \mid \mathsf{athrow} \mid \mathsf{ireturn} \\
\quad \mid \mathsf{getfield}\ Class\_NameField\_Sig \mid \mathsf{putfield}\ Class\_NameField\_Sig
\end{array}
$$

where $\diamond$ is a comparison operator (ne,le,_icmpgt, etc.), v a local variable, a an integer, pc an instruction address, and x an integer or the special value `NULL`.

3.4.2 From Bytecode to Control Flow Graphs

This section describes the generation of a *control flow graph* (CFG) from the bytecode of a method. This will allow transforming the unstructured control flow of bytecode into recursion. The technique we use follows well-established ideas in compilers [**?**], already applied in Java bytecode analysis [166].

Given a method m, we denote by G_m its CFG, which is a directed graph whose nodes are referred to as *blocks*. Each block $Block_{id}$ is a tuple of the form $\langle Id, G, B, D \rangle$ where: Id is the block's unique identifier; G is the *guard* of the block which indicates under which conditions the block is executed; B is a sequence of contiguous bytecode instructions which are guaranteed to be executed unconditionally (i.e., if G succeeds then all instructions in B are executed before control moves to another block); and D is the *adjacency list* for $Block_{Id}$, i.e., D contains the identifiers of all blocks which are possible successors of $Block_{Id}$, i.e., $Id' \in D$ iff there is an arc from $Block_{Id}$ to $Block_{Id'}$. Guards originate from bytecodes where the execution might take different paths depending on the runtime values. This is the case of bytecodes for conditional jumps, method invocation, and exceptions manipulation. In the CFG this will be expressed by *branching* from the corresponding block. The successive blocks will have mutually exclusive guards since only one of them will

be executed. Guards take the form `guard(cond)`, where `cond` is a Boolean condition on the local variables and stack elements of the method. It is important to point out that guards in the successive blocks will not be taken into account when computing the cost of a program.

A large part of the bytecode instruction set has only one successor. However, there are three types of branching statements:

Conditional jumps: of the form "pc_i : if$\diamond$ pc_j". Depending on the truth value of the condition, the execution can jump to pc_j or continue, as usual, with pc_{i+1}. The graph describes this behavior by means of two arcs from the block containing the instruction of pc_i to those starting respectively with instructions of pc_j and pc_{i+1}. Each one of these new blocks begins by a `guard` expressing the condition under which such block is to be executed.

Dynamic dispatch: of the form "pc_i : invokevirtual cm". The type of the object o whose method is being invoked is not known statically (it could be c or any subclass of c); therefore, we cannot determine statically which method is going to be invoked. Hence, we need to make all possible choices explicit in the graph. We deal with dynamic dispatching by using the function `resolve_virtual`(c, m), which returns the set *ResolvedMethods* of pairs $\langle d, \{c_1, \ldots, c_k\}\rangle$, where d is a class that defines a method with signature m and each c_i is either c or a subclass of c which inherits that specific method from d. For each $\langle d, \{c_1, \ldots, c_k\}\rangle \in$ *ResolvedMethods*, a new block $Block_d^{pc_i}$ is generated with a unique instruction `invoke`$(d{:}m)$ which stands for the *non-virtual* invocation of the method m that is defined in the class d. In addition, the block has a guard of the form `instanceof`$(o, \{c_1, \ldots, c_k\})$ (o is a stack element) to indicate that the block is applicable only when o is an instance of one of the classes $c_1, \ldots, c_k$. An arc from the block containing pc_i to $Block_d^{pc_i}$ is added, together with an arc from $Block_d^{pc_i}$ to the block containing the next instruction at pc_{i+1} (which describes the rest of the execution after invoking m). Note that the `invokevirtual` is no longer needed in the CFG since it was split into several `invoke` instructions which cover all the possible runtime scenarios. Yet, in order to take into account the cost of dynamic dispatching, we replace the `invokevirtual` by a corresponding call to `resolve_virtual`. Fields are treated in a similar way.

Exceptions: As regards the structure of the CFG, exceptions are not dealt with in a special way. Instead, the possibility of an exception being raised while executing a bytecode statement b is simply treated as an additional branching after b. Let $Block_b$ be the block ending with b; arcs exiting from $Block_b$ are those originated by its *normal behavior* control flow, together with those reaching the sub-graphs which correspond to exception handlers.

Describing dynamic dispatching and exceptions as additional blocks simplifies program analysis. After building the CFG, we do not need to distinguish how and why blocks were generated. Instead, all blocks can be dealt with uniformly.

Example 5 (running example)**.** *The execution of the method* `add(n, o)` *shown in Fig. 3.12 computes:* $\Sigma_{i=0}^{n} i$ *if* `o` *is an instance of A;* $\Sigma_{i=0}^{\lfloor n/2 \rfloor} 2i$ *if* `o` *is an instance of B; and* $\Sigma_{i=0}^{\lfloor n/3 \rfloor} 3i$ *if* `o` *is an instance of C. The CFG of the method* `add` *is depicted at the bottom of the figure. The fact that the successor of* `6: if_icmpgt 16` *can be either the instruction at address* `7` *or* `16` *is expressed by means of two arcs from* Block_1*, one to* Block_2 *and another one to* Block_3*, and by adding the guards* icmpgt *and* icmple *to* Block_2 *and* Block_3*, respectively. The invocation* `13: invokevirtual A.incr : (I)I` *is split into 3 possible runtime scenarios described in blocks* Block_4*,* Block_5 *and* Block_6*. Depending on the type of the object* `o` *(the second stack element from top, denoted* `s(top(1))` *in the guards), only one of these blocks will be executed and hence one of the definitions for* `incr` *will be invoked. Note that the* `invokevirtual` *bytecode is replaced by* `resolve_virtual`*. The exception behavior when* `o` *is a* `NULL` *object is described in blocks* Block_7 *and* Block_{exc}*.* □

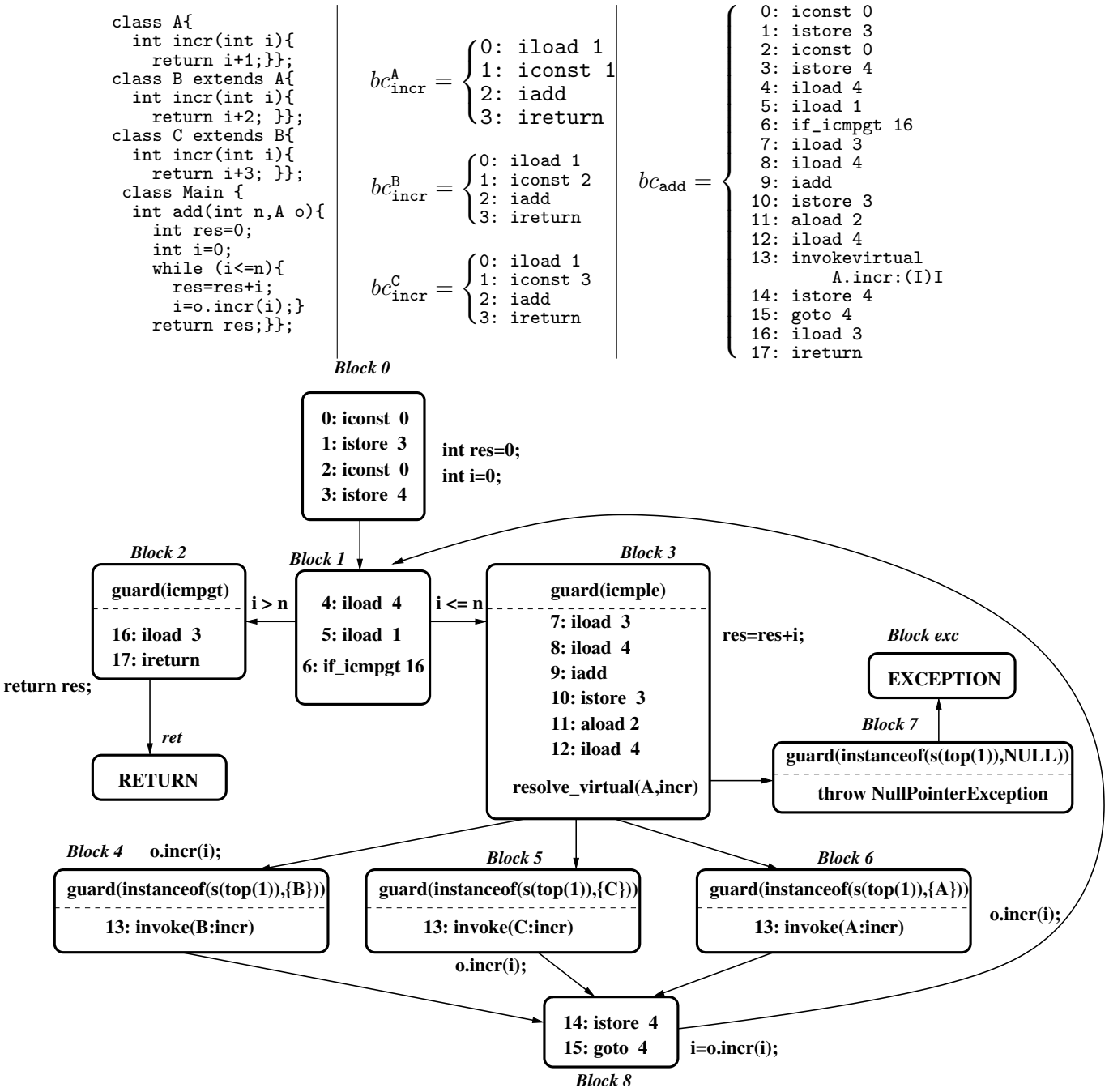

Figure 3.12: The running example in source code, bytecode, and control flow graph

3.4.3 Recursive Representation with Flattened Stack

In this section, we present a method for obtaining a representation of the code of a method where 1) iteration is transformed into recursion and 2) the operand stack is *flattened* in the sense that its contents are represented as a series of local variables. The latter is possible because in valid bytecode the maximum stack height t can always be statically decided. For the sake of simplicity, exceptions possibly occurring in a method will be ignored. Handling them introduces more branching in the CFG and also requires additional arguments in the recursive representation. This could influence the performance of the cost analysis.

Let m be a method defined in class c, with *local* variables $\overline{l_k} = l_0, \ldots, l_k$; of them, l_0 contains a reference to the *this* object, $l_1, \ldots, l_n$ are the n input arguments to the method, and $l_{n+1}, \ldots, l_k$ correspond to the

$\texttt{translate}(\texttt{m},\texttt{pc},\texttt{iadd},\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}) :=$ $\quad\texttt{let}\ \ \texttt{j} = \texttt{top\_stack\_index}(\texttt{pc},\texttt{m})\ \ \texttt{in}$ $\quad\quad\overline{\texttt{s}'_\texttt{t}} = \overline{\texttt{s}_\texttt{t}}[\texttt{j}{-}1 \mapsto \texttt{s}'_{\texttt{j}-1}]$ $\quad\quad\texttt{ret}\langle\texttt{iadd}(\texttt{s}_{\texttt{j}-1},\texttt{s}_\texttt{j},\texttt{s}'_{\texttt{j}-1}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}'_\texttt{t}}\rangle$	$\texttt{translate}(\texttt{m},\texttt{pc},\texttt{iload}(\texttt{v}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}) :=$ $\quad\texttt{let}\ \texttt{j} = \texttt{top\_stack\_index}(\texttt{pc},\texttt{m})\ \texttt{in}$ $\quad\quad\overline{\texttt{s}'_\texttt{t}} = \overline{\texttt{s}_\texttt{t}}[\texttt{j}{+}1 \mapsto \texttt{s}'_{\texttt{j}+1}]$ $\quad\quad\texttt{ret}\ \langle\texttt{iload}(\texttt{l}_\texttt{v},\texttt{s}'_{\texttt{j}+1}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}'_\texttt{t}}\rangle$
$\texttt{translate}(\texttt{m},\texttt{pc},\texttt{guard}(\texttt{icmpgt}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}) :=$ $\quad\texttt{let}\ \texttt{j} = \texttt{top\_stack\_index}(\texttt{pc},\texttt{m})\ \texttt{in}$ $\quad\texttt{ret}\ \langle\texttt{guard}(\texttt{icmpgt}(\texttt{s}_{\texttt{j}-1},\texttt{s}_\texttt{j})),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}\rangle$ $\texttt{translate}(\texttt{m},\texttt{pc},\texttt{ireturn}(\texttt{v}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}})\ :=$ $\quad\texttt{ret}\langle\texttt{ireturn}(\texttt{s}_0,\texttt{ret}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}\rangle$	$\texttt{translate}(\texttt{m},\texttt{pc},\texttt{invoke}(\texttt{b}{:}\texttt{m}'),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}) :=$ $\quad\texttt{let}\ \texttt{j} = \texttt{top\_stack\_index}(\texttt{pc},\texttt{m}),$ $\quad\quad\texttt{n} = \texttt{number\_of\_arguments}(\texttt{b},\texttt{m}')\ \texttt{in}$ $\quad\quad\overline{\texttt{s}'_\texttt{t}} = \overline{\texttt{s}_\texttt{t}}[\texttt{j}{-}\texttt{n} \mapsto \texttt{s}'_{\texttt{j}-\texttt{n}}]$ $\quad\quad\texttt{ret}\ \langle\texttt{b} : \texttt{m}'(\texttt{s}_{\texttt{j}-\texttt{n}},\ldots,\texttt{s}_\texttt{j},\texttt{s}'_{\texttt{j}-\texttt{n}}),\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}'_\texttt{t}}\rangle$

Figure 3.13: Translation of selected bytecode instructions

$k-n$ local variables declared in m. In addition to these arguments, we add the variables $\overline{s_t} = s_0,\ldots,s_{t-1}$, which correspond to the stack elements, with s_0 and s_{t-1} being the bottom-most and top-most positions respectively. Moreover, let h_{id} be the height of the stack at the entry of $Block_{id}$, and $\overline{s_t}|_{h_{id}}$ be the restriction of $\overline{s_t}$ to the corresponding stack variables. The *recursive representation* of m is defined as a set of rules *head* $\leftarrow$ *body* obtained from its control flow graph G_m as follows:

(1) the *method entry* rule is $\texttt{c:m}(\overline{\texttt{l}_\texttt{n}},\texttt{ret}) \leftarrow \texttt{c:m}^0(\overline{\texttt{l}_\texttt{k}},\texttt{ret})$, where ret is a variable for storing the return value,

(2) for each $Block_{id} = \langle id, G, \overline{B}_p, \{id_1,\ldots,id_j\}\rangle \in G_m$, there is a rule:

$$\texttt{c:m}^{id}(\overline{\texttt{l}_\texttt{k}},\overline{\texttt{s}_\texttt{t}}|_{\texttt{h}_\texttt{id}},\texttt{ret}) \leftarrow \texttt{G}',\overline{\texttt{B}}'_\texttt{p}(\texttt{call}_{\texttt{id}_1}\ ;\ \ldots\ ;\ \texttt{call}_{\texttt{id}_\texttt{j}})$$

where $\{\texttt{G}'\}\cup\overline{\texttt{B}}'_\texttt{p}$ is obtained from $\{\texttt{G}\}\cup\overline{\texttt{B}}_\texttt{p}$, and $\texttt{call}_{\texttt{id}_1}\ ;\ldots;\ \texttt{call}_{\texttt{id}_\texttt{j}}$ are possible calls to blocks (";" means disjunction), as explained below.

Each $b_i \in \{\texttt{G}\}\cup\overline{\texttt{B}}_\texttt{p}$ is *translated* into b'_i by explicitly adding the variables (local variables or stack variables) used by b_i as arguments. For example, iadd is translated to $\texttt{iadd}(s_{j-1},s_j,s'_{j-1})$, where j is the index of the top of the stack just before executing iadd. Here, we refer to the $j{-}1^{th}$ stack variable twice by different names: s_{j-1} refers to the input value and s'_{j-1} refers to the output value. The use of new names for output variables, in the spirit of *Static Single Assignment (SSA)* (see [61] and its references), is crucial in order to obtain simple, yet efficient, denotational program analyses. In Fig. 3.13 we give the translation function for selected bytecodes; among them, the one for iadd works as follows. Function translate takes as input the name of the current method m, the program counter pc of the bytecode, the bytecode (in this case iadd), the current local variable *names* $\overline{\texttt{l}_\texttt{k}}$, and the current stack variable *names* $\overline{\texttt{s}_\texttt{t}}$. In line 1, we retrieve the index of the top stack element before executing the current bytecode. In line 2, we generate new stack variable *names* $\overline{\texttt{s}'_\texttt{t}}$ by renaming the output variable of iadd in $\overline{\texttt{s}_\texttt{t}}$. As notation, given a sequence $\overline{a}_n$ of elements, $\overline{a}_n[i \mapsto b]$ denotes the replacement in $\overline{a}_n$ of the element a_i by b. In line 3, we return ($\texttt{ret}\langle\_\rangle$) the translated bytecode together with the new stack variable *names*. Assume that $\texttt{G}{=}pc_0{:}b_0$ and $\overline{\texttt{B}}_\texttt{p}{=}\langle pc_1{:}b_1,\ldots,pc_\texttt{p}{:}b_\texttt{p}\rangle$. The translation of all bytecodes is done iteratively as follows:

$$\texttt{for i} = 0\ \texttt{to p}\ \ \{\langle b'_\texttt{i},\overline{\texttt{l}}_\texttt{k}^{\texttt{i}+1},\overline{\texttt{s}}_\texttt{t}^{\texttt{i}+1}\rangle = \texttt{translate}(\texttt{m},pc_\texttt{i},b_\texttt{i},\overline{\texttt{l}}_\texttt{k}^{\texttt{i}},\overline{\texttt{s}}_\texttt{t}^{\texttt{i}})\}$$

We start from an initial set of local and stack variables, $\overline{\texttt{l}}_\texttt{k}^{0}{=}\overline{\texttt{l}}_\texttt{k}$ and $\overline{\texttt{s}}_\texttt{t}^{0}{=}\overline{\texttt{s}}_\texttt{t}$; in each step, translate takes as input the local and stack variable names which were generated by translating the previous bytecode. At the end of this loop, we can define each $\texttt{call}_{\texttt{id}_\texttt{i}}$, $1 \le i \le j$, as $\texttt{c:m}^{\texttt{id}_\texttt{i}}(\overline{\texttt{l}}_\texttt{k}^{\texttt{p}+1},\overline{\texttt{s}}_\texttt{t}^{\texttt{p}+1}|_{\texttt{h}_{\texttt{id}_\texttt{i}}},\texttt{ret})$, meaning that we call the next block with the last local and (restricted) stack variable *names*.

Example 6. *Consider the CFG in Fig. 3.12. The translation of* Block_3 *and* Block_4 *works as shown below. For clarity, in the block identifiers we have not included the class name for the* add *method. Also, we ignore the exception branch from* Block_3 *to* Block_7.

$\texttt{add}^3(\overline{\texttt{l}}_4, \texttt{s}_0, \texttt{s}_1, \texttt{ret}) \leftarrow$ $\quad \mathbf{guard}(\mathbf{icmple}(\mathbf{s_0}, \mathbf{s_1}))$, $\quad \texttt{iload}(\texttt{l}_3, \texttt{s}'_0), \quad \texttt{iload}(\texttt{l}_4, \texttt{s}'_1), \texttt{iadd}(\texttt{s}'_0, \texttt{s}'_1, \texttt{s}''_0)$, $\quad \texttt{istore}(\texttt{s}''_0, \texttt{l}'_3), \texttt{aload}(\texttt{l}_2, \texttt{s}'''_0), \texttt{iload}(\texttt{l}_4, \texttt{s}''_1)$, $\quad \texttt{resolve\_virtual}(\texttt{A}, \texttt{incr})$, $\quad (\ \texttt{add}^4(\texttt{l}_0, \texttt{l}_1, \texttt{l}_2, \texttt{l}'_3, \texttt{l}_4, \texttt{s}'''_0, \texttt{s}''_1, \texttt{ret})$; $\quad\ \ \texttt{add}^5(\texttt{l}_0, \texttt{l}_1, \texttt{l}_2, \texttt{l}'_3, \texttt{l}_4, \texttt{s}'''_0, \texttt{s}''_1, \texttt{ret})$; $\quad\ \ \texttt{add}^6(\texttt{l}_0, \texttt{l}_1, \texttt{l}_2, \texttt{l}'_3, \texttt{l}_4, \texttt{s}'''_0, \texttt{s}''_1, \texttt{ret})\ )$	$\texttt{add}^4(\overline{\texttt{l}}_4, \texttt{s}_0, \texttt{s}_1, \texttt{ret}) \leftarrow$ $\quad \mathbf{guard}(\mathbf{instanceof}(\mathbf{s_0}, \{\mathbf{B}\}))$, $\quad \texttt{B:incr}(\texttt{s}_0, \texttt{s}_1, \texttt{s}'_0)$, $\quad \texttt{add}^8(\overline{\texttt{l}}_4, \texttt{s}'_0, \texttt{ret})$.

In the $\texttt{add}^3$ *rule, dynamic dispatch is represented as a disjunction of calls to* $\texttt{add}^4$, $\texttt{add}^5$ *or* $\texttt{add}^6$. *Thus, in the rule for* $\texttt{add}^4$, *we find a call to (the translation of)* `incr` *from class* `B` *which corresponds to the translation of* `invoke(B:incr)`; *arguments passed to* `incr` *are the two top-most stack elements; the return value (the last argument) goes also to the stack. Note the change in the superscript when a variable is updated.* □

Several optimizations are applied to the above translation. An important one is to replace (redundant) stack variables corresponding to intermediate states by local variables whenever possible. This can be done by tracking dependencies between variables, which stem from instructions like `iload` and `istore`. The fact that the program is in SSA form makes this transformation relatively straightforward. However, note that, in order to eliminate stack variables from the head of a block, we need to consider all calling patterns to the block.

Example 7. *After eliminating redundant variables, the optimized version of rules 3 and 4 from Ex. 6 is as follows:*

$\texttt{add}^3(\overline{\texttt{l}}_4, \texttt{ret}) \leftarrow$ $\quad \mathbf{guard}(\mathbf{icmple}(\mathbf{l_4}, \mathbf{l_1}))$, $\quad \underline{\texttt{iload}(\texttt{l}_3, \texttt{s}'_0), \texttt{iload}(\texttt{l}_4, \texttt{s}'_1), \texttt{iadd}(\texttt{l}_3, \texttt{l}_4, \texttt{l}'_3)}$, $\quad \underline{\texttt{istore}(\texttt{s}''_0, \texttt{l}'_3), \texttt{aload}(\texttt{l}_2, \texttt{s}'''_0), \texttt{iload}(\texttt{l}_4, \texttt{s}''_1)}$, $\quad \texttt{resolve\_virtual}(\texttt{A}, \texttt{incr})$, $\quad (\ \texttt{add}^4(\texttt{l}_0, \texttt{l}_1, \texttt{l}_2, \texttt{l}'_3, \texttt{l}_4, \texttt{ret})$; $\quad\ \ \texttt{add}^5(\texttt{l}_0, \texttt{l}_1, \texttt{l}_2, \texttt{l}'_3, \texttt{l}_4, \texttt{ret})$; $\quad\ \ \texttt{add}^6(\texttt{l}_0, \texttt{l}_1, \texttt{l}_2, \texttt{l}'_3, \texttt{l}_4, \texttt{ret})\ )$	$\texttt{add}^4(\overline{\texttt{l}}_4, \texttt{ret}) \leftarrow$ $\quad \mathbf{guard}(\mathbf{instanceof}(\mathbf{l_2}, \{\mathbf{B}\}))$, $\quad \texttt{B:incr}(\texttt{l}_2, \texttt{l}_4, \texttt{s}'_0)$, $\quad \texttt{add}^8(\overline{\texttt{l}}_4, \texttt{s}'_0, \texttt{ret})$.

The underlined instructions have been used to discover equivalences among stack elements and local variables. For example, all the arguments of `iadd` *have been replaced by local variables. However, eliminating stack variables is not always possible. This is the case of* $\texttt{s}'_0$ *in the rule* $\texttt{add}^4$, *as it corresponds to the return value of* `B:incr`. *After these optimizations, the underlined instructions become redundant and could be removed. However, we do not remove them in order to take their cost into account in the next sections.* □

3.4.4 Size Relations for Cost Analysis

Obtaining *size-relations* between the states at different program points is indispensable for setting up cost relations. In particular, they are essential for defining the cost of one block in terms of the cost of its successors. In general, various *measures* can be used to determine the *size* of an input. For instance, in symbolic languages (see, e.g., [**?**]), term-depth, list-length, etc. are used as term sizes. In Java bytecode, we consider two cases: for integer variables, *size-relations* are constraints on the possible values of variables; for reference variables, they are constraints on the length of the longest reachable paths.

Example 8. *Consider the two loops below, written in Java for simplicity:*

```
while( i>0 ) { i--; }        while( l != null ) { l = l.next; }
```

A useful size-relation *for cost analysis is that the value of* `i` *is always greater than* `0` *and decreases by* `1` *in each iteration, and that the longest path reachable from* `l` *is decreasing by* `1` *in each iteration.* □

Inferring *size-relations* is not straightforward: such relations might be the result of executing several statements, calling methods or loops. For instance, in our running example, the size relation for variable `i` is the result of executing the method `incr` and is propagated through the loop in the procedure `add`. Fixpoint computation is often required. Fortunately, there are several abstract interpretation based approaches for inferring *size-relations* between integer variables [**?**], as well as between reference variables (in terms of longest path length) [167].

The notion of *Size Relation*

In order to set up cost relations, we need, for each rule in the recursive representation, the *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) which occur in the body. Note that, given a rule $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G}, \overline{\mathtt{B}}_\mathtt{k}, (\mathtt{q}_1; \ldots; \mathtt{q}_\mathtt{n})$, each $\mathtt{b}_\mathtt{i} \in \overline{\mathtt{B}}_\mathtt{k}$ is either a bytecode or a call to another rule (which stems from the translation of a method invocation). We denote by $\mathtt{calls}(\overline{\mathtt{B}}_\mathtt{k})$ the set of all $\mathtt{b}_\mathtt{i}$ corresponding to a method call, and by $\mathtt{bytecode}(\overline{\mathtt{B}}_\mathtt{k})$ the set of all $\mathtt{b}_\mathtt{i}$ corresponding to other bytecodes.

Definition 3.4.1 (calls-to size-relations)**.** *Let $\mathcal{R}_m$ be the recursive representation of a method m, where each rule takes the form $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G}, \overline{\mathtt{B}}_\mathtt{k}, (\mathtt{q}_1(\bar{\mathtt{y}}); \cdots; \mathtt{q}_\mathtt{n}(\bar{\mathtt{y}}))$. The calls-to* size-relations *of $\mathcal{R}_m$ are triples of the form*

$$\langle \mathtt{p}(\bar{\mathtt{x}}), \mathtt{p}'(\bar{\mathtt{z}}), \varphi \rangle \quad \textit{where} \quad \mathtt{p}'(\bar{\mathtt{z}}) \in \mathtt{calls}(\overline{\mathtt{B}}_\mathtt{k}) \cup \{\mathtt{p\_cont}(\bar{\mathtt{y}})\}$$

describing, for all rules, the size-relation *between $\bar{\mathtt{x}}$ and $\bar{\mathtt{z}}$ when $\mathtt{p}'(\bar{\mathtt{z}})$ is called, where $\mathtt{p\_cont}(\bar{\mathtt{y}})$ refers to the program point immediately after $\overline{\mathtt{B}}_\mathtt{k}$. The* size-relation *$\varphi$ is given as a conjunction of linear constraints $\mathtt{a}_0+\mathtt{a}_1\mathtt{v}_1+\cdots+\mathtt{a}_\mathtt{n}\mathtt{v}_\mathtt{n}\ \mathtt{op}\ 0$, where $\mathtt{op} \in \{=, \leq, <\}$, each $\mathtt{a}_\mathtt{i}$ is a constant and $\mathtt{v}_\mathtt{k} \in \bar{\mathtt{x}} \cup \bar{\mathtt{z}}$ for each $\mathtt{k}$.*

Note that in the definition above there is no need to have separate relations for each $\mathtt{q}_\mathtt{i}(\bar{\mathtt{y}})$ as, in the absence of exceptions, size relations are exactly the same for all of them, since they correspond to the same program point.

Inferring Size Relations

A simple, yet quite precise and efficient, *size-relation* analysis for the recursive representation of methods can be done in two steps: 1) compiling the bytecodes into the linear constraints they impose on variables; and 2) computing a bottom-up fixpoint on the compiled rules using standard bottom-up fixpoint algorithms. Compilation into linear constraints is done by an abstraction function α_{size} which basically replaces guards and bytecodes by the constraints they impose on the corresponding variables. In general, each bytecode performing (linear) arithmetic operations is replaced by a corresponding linear constraint, and each bytecode which manipulates objects is compiled to linear constraints on the length of the longest reachable path from the corresponding variable [167]. Here are some examples of abstracting guards and bytecodes into linear constraints:

$\dfrac{\alpha_{\mathtt{size}}(\mathtt{iload}(\mathtt{l}_1, \mathtt{s}_0)) := (\mathtt{l}_1 = \mathtt{s}_0)}{\alpha_{\mathtt{size}}(\mathtt{iadd}(\mathtt{s}_1, \mathtt{s}_0, \mathtt{s}'_0)) := (\mathtt{s}'_0 = \mathtt{s}_0 + \mathtt{s}_1)}$	$\dfrac{\alpha_{\mathtt{size}}(\mathtt{guard}(\mathtt{icmpgt}(\mathtt{s}_1, \mathtt{s}_0))) := (\mathtt{s}_1 > \mathtt{s}_0)}{\alpha_{\mathtt{size}}(\mathtt{getfield}(\mathtt{s}_1, \mathtt{f}, \mathtt{s}'_1)) := (\mathtt{s}'_1 < \mathtt{s}_1)}$

It is important to note that $\alpha_{\mathtt{size}}$ uses the same name for the original variables in order to refer to their sizes. Compiling the rules of Ex. 7 results in:

$\mathtt{add}^3(\overline{\mathtt{l}}_4, \mathtt{ret}) \leftarrow \mathtt{l}_4 \leq \mathtt{l}_1, \mathtt{l}'_3 = \mathtt{l}_3 + \mathtt{l}_4,$ $\mathtt{resolve\_virtual}(\mathtt{A}, \mathtt{incr}),$ $(\mathtt{add}^4(\overline{\mathtt{l}}_2, \mathtt{l}'_3, \mathtt{l}_4, \mathtt{ret}); \mathtt{add}^5(\overline{\mathtt{l}}_2, \mathtt{l}'_3, \mathtt{l}_4, \mathtt{ret}); \mathtt{add}^6(\overline{\mathtt{l}}_2, \mathtt{l}'_3, \mathtt{l}_4, \mathtt{ret}))$	$\mathtt{add}^4(\overline{\mathtt{l}}_4, \mathtt{ret}) \leftarrow$ $\mathtt{B{:}incr}(\mathtt{l}_2, \mathtt{l}_4, \mathtt{s}'_0),$ $\mathtt{add}^8(\overline{\mathtt{l}}_4, \mathtt{s}'_0, \mathtt{ret}).$

Example 9. *Compiling all the rules corresponding to the program in Fig. 3.12 and computing a bottom-up fixpoint over an appropriate abstract domain [**?**] would result in the following calls-to* size-relations *for rules from Ex. 6:*

$\langle \mathtt{add}^3(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{ret}), \mathtt{add}^3\mathtt{\_cont}(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}'_3, \mathtt{l}_4, \mathtt{ret}), \{\mathtt{l}_4 \leq \mathtt{l}_1, \mathtt{l}'_3 = \mathtt{l}_3 + \mathtt{l}_4\} \rangle$
$\langle \mathtt{add}^4(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{ret}), \mathtt{B{:}incr}(\mathtt{l}_2, \mathtt{l}_4, \mathtt{ret}), \{\} \rangle$
$\langle \mathtt{add}^4(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{ret}), \mathtt{add}^4\mathtt{\_cont}(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{s}'_0, \mathtt{ret}), \{\mathtt{s}'_0 = \mathtt{l}_4 + 2\} \rangle$

□

3.4.5 Cost Relations for Java Bytecode

We now present our approach to the automatic generation of *cost relations* which define the computational cost of the execution of a bytecode method. They are generated from the recursive representation of the method (Sec. 3.4.3) and by using the information inferred by the size analysis (Sec. 3.4.4). An important issue in order to obtain optimal cost relations is to find out the arguments which can be safely ignored in cost relations.

Restricting Cost Relations to (Subsets of) Input Arguments

Let us consider *Block*$_{id}$ in a CFG, represented by the rule $\mathtt{c{:}m}^{id}(\overline{\mathtt{l_k}}, \mathtt{ret}) \leftarrow \mathtt{G}, \overline{\mathtt{B_h}}, (\mathtt{call_{id_1}}\ ;\ \ldots\ ;\ \mathtt{call_{id_j}})$ in which local and stack variables are no longer distinguishable. The cost function for *Block*$_{id}$ takes the form $C_{id} : (\mathbb{Z})^n \rightarrow \mathbb{N}_\infty$, with $n \leq k$ argument positions, and where $\mathbb{Z}$ is the set of integers and $\mathbb{N}_\infty$ is the set of natural numbers augmented with a special symbol ∞, denoting *unbounded*.

Our aim here is to minimize the number n of arguments which need to be taken into account in cost functions. As usual in cost analysis, we consider that the output argument `ret` cannot influence the cost of any block, so that it can be ignored in cost functions. Furthermore, it is sometimes possible to disregard some input arguments. For instance, in our running example, $\mathtt{l_3}$ is an *accumulating* parameter whose value does not affect the control flow nor the cost of the program: it merely keeps the value of the temporary result.

Given a rule, the arguments which can have an impact on the cost of the program are those which may affect directly or indirectly the program guards (i.e., they can affect the control flow of the program), or are used as input arguments to external methods whose cost, in turn, may depend on the input size. Computing a safe approximation of the set of variables affecting a series of statements is a well studied problem in static analysis. To do this, we need to follow data dependencies against the control flow, and this involves computing a fixpoint. Our problem is slightly simpler than *program slicing* [170], since we do not need to delete redundant program statements; instead, we only need to detect relevant arguments. Given a rule $p(\overline{\mathtt{x}}) \leftarrow$ *body* (p for short), $\hat{\mathtt{l}}_p \subseteq \overline{\mathtt{x}}$ is the sub-sequence of *relevant variables* for p. The sequence $\hat{\mathtt{l}}_P$, obtained by union of sequences $\{\hat{\mathtt{l}}_p\}_{p \in P}$ for a set P of rules, keeps the ordering on variables.

Example 10. *Given p_i, corresponding to* Block$_i$ *in the graph of the running example, we are interested in computing which variables in this rule are relevant to program guards or external methods. For example, 1) when the execution flow reaches p_2, we execute the unconditional bytecode instructions in p_2 and move to the final block. As a result, there are no relevant variables for p_2, since none can have any impact on its cost, and p_2 does not* reach *any guards nor methods. 2) On the other hand, p_3 can reach the guards in p_4, p_5 and p_6, which take the form* `instanceof(_)` *and involve* $\mathtt{l_2}$. *Also, the guard in p_3 itself, involving* $\mathtt{l_1}$ *and* $\mathtt{l_4}$, *can be recursively reached via the loop. Moreover, the call to the external method* ***incr*** *involves* $\mathtt{l_2}$ *and* $\mathtt{l_4}$. *After computing a fixpoint, we conclude that* $\hat{\mathtt{l}}_{p_3} = \{\mathtt{l_1}, \mathtt{l_2}, \mathtt{l_4}\}$. *3) We have* $\hat{\mathtt{l}}_{p_8} = \{\mathtt{l_1}, \mathtt{l_2}, \mathtt{s_0}\}$*; here,* $\mathtt{s_0}$ *is also relevant since it affects* $\mathtt{l_4}$ *(which in turn is involved in the guard of p_3, reachable from p_8).* □

The Cost Relation

Herein, we define the cost function $C_{id} : (\mathbb{Z})^n \rightarrow \mathbb{N}_\infty$ for a *Block*$_{id}$ by means of a *cost relation* which consists of a set of *cost equations*. It will allow us to reason about the computational cost of the execution of the block *id*. The intuitive idea is that, given the rule $\mathtt{p}(\overline{\mathtt{x}}) \leftarrow \mathtt{G}, \mathtt{B}, (\mathtt{q_1}; \ldots; \mathtt{q_n})$ associated to *Block*$_{id}$, we generate:

- one cost equation which defines the cost of `p` as the cost of the statements in `B`, plus the cost of its *continuation*, denoted `p_cont`;
- another cost equation which defines the cost of `p_cont` as either the cost of $\mathtt{q_1}$ (if its guard is satisfied), ..., or the cost of $\mathtt{q_n}$ (if its guard is satisfied).

We specify the cost of the *continuation* in a separate equation because the conditions for determining the alternative path $\mathtt{q_i}$ that the execution will take (with $i = 1, \ldots, n$) are only known at the end of the

execution of B; thus, they cannot be evaluated before B is executed. In the definition below, we use the function α_{guard} to replace those guards which indicate the type of an object by the appropriate test (e.g., $\alpha_{guard}(\texttt{guard}(\texttt{instanceof}(\texttt{s}_0, \{\texttt{B}\}))) := \texttt{s}_0 \in \texttt{B}$). For guards on size relations, it is equivalent to α_{size}.

Definition 3.4.2 (cost relation)**.** *Let $\mathcal{R}_m$ be the recursive representation of a method m where each block takes the form $\texttt{p}(\bar{\texttt{x}}) \leftarrow \texttt{G}_\texttt{p}, \texttt{B}, (\texttt{q}_1(\bar{\texttt{y}}); \cdots; \texttt{q}_\texttt{n}(\bar{\texttt{y}}))$ and $\hat{\texttt{l}}_\texttt{p}$ be its sequence of relevant variables. Let φ be the calls-to size relation for $\mathcal{R}_m$ where each size relation is of the form $\langle \texttt{p}(\bar{\texttt{x}}), \texttt{p}'(\bar{\texttt{z}}), \varphi^{\texttt{p}(\bar{\texttt{x}})}_{\texttt{p}'(\bar{\texttt{z}})} \rangle$ for all $\texttt{p}'(\bar{\texttt{z}}) \in \texttt{calls}(\texttt{B}) \cup \{\texttt{q}(\bar{\texttt{y}})\}$ such that $\texttt{q}(\bar{\texttt{y}})$ refers to the program point immediately after B. Then, we generate the cost equations for each block of the above form in $\mathcal{R}_m$ as follows:*

$$C_p(\hat{\texttt{l}}_p) = \sum_{b \in bytecode(B)} T_b + \sum_{r(\bar{z}) \in calls(B)} C_r(\hat{\texttt{l}}_r) + C_{p\_cont}(\cup_{i=1}^n \hat{\texttt{l}}_{q_i}) \qquad \bigwedge_{r(\bar{z}) \in calls(B)} (\varphi^{p(\overline{x})}_{r(\bar{z})}) \wedge \varphi^{p(\overline{x})}_{q(\bar{y})}$$

$$C_{p\_cont}(\cup_{i=1}^n \hat{\texttt{l}}_{q_i}) = \begin{cases} C_{q_1}(\hat{\texttt{l}}_{q_1}) & \alpha_{guard}(G_{q_1}) \\ \cdots & \\ C_{q_n}(\hat{\texttt{l}}_{q_n}) & \alpha_{guard}(G_{q_n}) \end{cases}$$

where T_b is the cost unit associated to the bytecode b. The cost relation associated to $\mathcal{R}_m$ and φ is defined as the set of cost equations of its blocks.

Let us notice four points about the above definition. 1) The size relationships between the input variables provided by the size analysis are *attached* to the cost equation for p. 2) Guards do not affect the cost: they are simply used to define the applicability conditions of the equations. 3) Arguments of the cost equations are only the relevant arguments to the block. In the equation for the continuation, we need to include the union of all relevant arguments to each of the subsequent blocks $\texttt{q}_\texttt{i}$.

The cost T_b of an instruction b depends on the chosen *cost model*. If our interest is merely on finding out the complexity or on approximating the number of bytecode statements which will be executed, then T_b can be the same for all instructions. On the other hand, we may use more refined cost models in order to estimate the execution time of methods. Such models may assign different costs to different instructions. One approach might be based on the use of a *profiling* tool which estimates the value of each T_b on a particular platform. (see, e.g., an application [120] for Prolog). It should be noted that, since we are not dealing with the problem of choosing a realistic cost model, a direct comparison between the result of our analysis and the actual measured run time (e.g., in milliseconds) cannot be done; instead, in this paper we focus only on the number of instructions to be executed.

Example 11. *Consider the recursive representation in Ex. 6 (without irrelevant variables, as explained in Ex. 10). Consider the size relations derived in Ex. 9; by applying Def. 3.4.2, we obtain the following cost relations:*

$$\begin{array}{lcll}
C_{\texttt{add}}(\texttt{l}_1, \texttt{l}_2) & = & C_{\texttt{add}^0}(\texttt{l}_1, \texttt{l}_2) & \\
C_{\texttt{add}^0}(\texttt{l}_1, \texttt{l}_2) & = & \texttt{T}_0 + C_{\texttt{add}^1}(\texttt{l}_1, \texttt{l}_2, \texttt{l}'_4) & \texttt{l}'_4 = \texttt{0} \\
C_{\texttt{add}^1}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \texttt{T}_1 + C_{\texttt{add}^1_\texttt{cont}}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & \\
C_{\texttt{add}^1_\texttt{cont}}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \begin{cases} C_{\texttt{add}^2}() & \texttt{l}_4 > \texttt{l}_1 \\ C_{\texttt{add}^3}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & \texttt{l}_4 \leq \texttt{l}_1 \end{cases} & \\
C_{\texttt{add}^2}() & = & \texttt{T}_2 & \\
C_{\texttt{add}^3}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \texttt{T}_3 + C_{\texttt{add}^3_\texttt{cont}}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & \\
C_{\texttt{add}^3_\texttt{cont}}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \begin{cases} C_{\texttt{add}^4}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & \texttt{l}_2 \in \texttt{B} \\ C_{\texttt{add}^5}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & \texttt{l}_2 \in \texttt{C} \\ C_{\texttt{add}^6}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & \texttt{l}_2 \in \texttt{A} \end{cases} & \\
C_{\texttt{add}^4}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \texttt{T}_4 + C_{\texttt{B:incr}}(\texttt{l}_2, \texttt{l}_4) + C_{\texttt{add}^8}(\texttt{l}_1, \texttt{l}_2, \texttt{s}_0) & \texttt{s}_0 = \texttt{l}_4 + \texttt{2} \\
C_{\texttt{add}^5}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \texttt{T}_5 + C_{\texttt{C:incr}}(\texttt{l}_2, \texttt{l}_4) + C_{\texttt{add}^8}(\texttt{l}_1, \texttt{l}_2, \texttt{s}_0) & \texttt{s}_0 = \texttt{l}_4 + \texttt{3} \\
C_{\texttt{add}^6}(\texttt{l}_1, \texttt{l}_2, \texttt{l}_4) & = & \texttt{T}_6 + C_{\texttt{A:incr}}(\texttt{l}_2, \texttt{l}_4) + C_{\texttt{add}^8}(\texttt{l}_1, \texttt{l}_2, \texttt{s}_0) & \texttt{s}_0 = \texttt{l}_4 + \texttt{1} \\
C_{\texttt{add}^8}(\texttt{l}_1, \texttt{l}_2, \texttt{s}_0) & = & \texttt{T}_8 + C_{\texttt{add}^1}(\texttt{l}_1, \texttt{l}_2, \texttt{s}_0) &
\end{array}$$

$\texttt{T}_{\texttt{B}_\texttt{i}}$ denotes the sum of the costs of all bytecode instructions contained in Block_i*. For brevity, as the blocks* 0, 2, 4, 5, 6, *and* 8 *have a single-branched continuation, we merge their two equations. Note that the cost relation for the external method* **incr** *does not include the third argument since it is an output argument.*□

Demonstrating the correctness of our approach to cost analysis requires: (1) Defining the meaning of cost in terms of the Java bytecode operational semantics; (2) Inheriting that definition to a corresponding (equivalent) operational a semantics of the recursive representation. (3) Demonstrating that the cost relations describe the cost as defined in step 2. The first two steps are straightforward as the CFG and the recursive representation describe the behavior of the original program, in particular at each branching point we have several possibilities from which *only* one will be executed. The correctness of the third step stems from the facts that the cost relations are obtained from the recursive representation by replacing each bytecode by its cost, and that the size analysis provides us with information that can be used to compute (or approximate) the number of times we visit in each program point during the execution.

3.4.6 Experiments in Cost Analysis of Java Bytecode

The purpose of this section is to assess the practicality of such cost analysis by experimentally evaluating a prototype analyzer implemented in `Ciao`. With this aim, we approximate the computational complexity of a set of selected benchmarks, including both well-known algorithms which have been used to evaluate existing cost analyzers in other programming paradigms, and other benchmarks which illustrate object-oriented features. In our evaluation, we first study whether the generated cost relations can be automatically solved. Our experiments show that in some cases the inferred cost relations can be automatically solved by using the Mathematica system, whereas, in other cases, some prior manipulation is required for the equations to be solvable. Moreover, we experimentally evaluated the running time of the different phases of the analysis process. Overall, we believe our experiments show that the efficiency of our cost analysis is acceptable, and that the obtained cost relations are useful in practice since, at least in our experiments, it is possible to get a closed form solution.

Cost Analysis for Recursive procedures

In this section, we infer the cost of two classical recursive procedures. In both cases, and in general for recursive procedures whose base case depends on constant values, the cost relations obtained by our analysis are directly solvable by Mathematica. For simplicity, in the following the cost of all bytecode instructions is assumed to be 1; using a more refined cost model which assigns different costs to different bytecodes would not introduce further complications. For readability, we present only the original Java code, instead of the bytecode.

The Classical Hanoi Towers The first example corresponds to the classical algorithm of the Hanoi Towers, which is depicted in the table below; the call $\mathsf{hanoi}(7,1,2,3)$ moves 7 disks from tower 1 to tower 3 using the auxiliary tower 2. The recurrence equations obtained by the analyzer are depicted in the same table. The equation $\mathsf{hanoi}[\mathsf{n}]$ corresponds to the total cost of a call to hanoi, where n is the first argument of the method. The other equations correspond to the cost of the different blocks in the control flow graph; they are obtained directly from the corresponding recursive representation. For example, the equation $\mathsf{m0}[\mathsf{n}]$ corresponds to verifying the condition $\mathsf{n}>0$; here, 2 is the cost of the corresponding bytecodes used in the comparison. The equation $\mathsf{m3}[0]$ corresponds to the base-case (when $\mathsf{n}\leq 0$), and $\mathsf{m3}[\mathsf{n}]$ corresponds to executing the *then* branch; the constant 15 is the cost of the corresponding bytecodes, and the two occurrences of $\mathsf{hanoi}[\mathsf{n}-1]$ are the cost of the recursive calls. The fact that n decreases by 1 in the recursive calls was detected by size analysis of the bytecode program. Note that the local variables, and stack elements, which do not appear in the equations were removed by the slicing algorithm, since they do not affect the base-case condition; therefore, they are not relevant for the cost.

Once the equations have been generated, we solve them in Mathematica by calling its recurrence equation solver RSolve. The query $\mathsf{RSolve}[\{\mathsf{eqns}\},\{\mathsf{a}[\mathsf{n}],\ldots,\mathsf{z}[\mathsf{y}]\},\{\mathsf{n},\ldots,\mathsf{y}\}]$ solves a set of recurrence equations {eqns} for $\mathsf{a}[\mathsf{n}],\ldots,\mathsf{z}[\mathsf{y}]$, where $\mathsf{n},\ldots,\mathsf{y}$ are the only variables, by giving solutions for $\mathsf{a},\ldots,\mathsf{z}$ as pure functions. The full Mathematica query is shown in the table. We are able to solve the above equations without any preprocessing, and, as expected, the obtained answer predicts an exponential complexity for $\mathsf{hanoi}[\mathsf{n}]$.

Cost relations and Mathematica solution for Hanoi

```
static void hanoi(int n,int s,int a,int t) {
 if (n > 0) {
   hanoi(n-1, s, t, a);
   System.out.println(n+":"+s+"→"+t);
   hanoi(n-1, a, t, s); } }
```

$$\mathcal{E}_{hanoi} = \begin{cases} \text{hanoi[n]} == \text{m0[n]}, \\ \text{m0[n]} == 2 + \text{m3[n]}, \\ \text{m3[0]} == 1, \\ \text{m3[n]} == \text{m4[n]}, \\ \text{m4[n]} == 15 + \text{hanoi[n-1]} + \text{hanoi[n-1]} \end{cases}$$

Mathematica **query:** RSolve[$\{\mathcal{E}_{hanoi}\}$, {hanoi[n],m0[n],m3[n],m4[n]},n]

Mathematica **answer (complexity):** hanoi[n] $\rightarrow$ (-17) + 5 2^{2+n}

Figure 3.14: The Hanoi Problem

Cost relations and Mathematica solution for Fibonacci

```
static int fib(int n){
  if ((n==0) || (n==1))   return 1;
   else return (fib(n-1)+fib(n-2));
}
```

$$\mathcal{E}_{fib} = \begin{cases} \text{fib[n]} == \text{m0[n]}, \\ \text{m0[n]} == 2 + \text{m4[n]}, \\ \text{m4[0]} == 2, \\ \text{m4[n]} == \text{m5[n]}, \\ \text{m5[n]} == 3 + \text{m6[n]}, \\ \text{m6[1]} == 2, \\ \text{m6[n]} == \text{m7[n]}, \\ \text{m7[n]} == 10 + \text{fib[n-1]} + \text{fib[n-2]} \end{cases}$$

Mathematica **query:** RSolve[$\{\mathcal{E}_{fib}\}$, {fib[n],m0[n],m4[n],m6[n],m7[n]},n]

Mathematica **answer (complexity):** fib[n] $\rightarrow$-(2^{3-n} (15 2^{1+n} - 19 (1 - $\sqrt{5}$)^n + 5 $\sqrt{5}$ (1 - $\sqrt{5}$)^n - 19 (1 + $\sqrt{5}$)^n - 5 $\sqrt{5}$ (1 + $\sqrt{5}$)^n)) / ((-1 + $\sqrt{5}$)^2 (1 + $\sqrt{5}$)^2)

Figure 3.15: The Fibonacci Problem

Recursive Fibonacci The next example (Fig. 3.15) is a recursive implementation of the Fibonacci number series. The recurrence equations obtained by the analyzer are depicted in the same table. The equation fib[n] corresponds to the total cost of a call fib(n). The other equations correspond to the different blocks in the control flow graph. For example, m4[0] and m4[n] correspond to the success and failure of the condition n == 0, respectively. Similarly, m6[1] and m6[n] corresponds to n == 1. The equation m7[n] corresponds to the cost of the recursive calls and their corresponding bytecodes; the decreasing by 1 and 2 in the calls was detected by size analysis on the bytecode. Moreover, irrelevant stack elements were removed from the equations by means of slicing. Solving the above equations in Mathematica gives the expected exponential complexity.

Analyzing Programs with Arrays and (Nested) Loops

In this section, we assess the practicality of the cost analysis for several procedures dealing with arrays and loops. We start by an example for array reversal, whose cost relations are solvable in Mathematica. Then, we study array concatenation, which requires some transformations over the cost relation in order to make it solvable. Finally, we analyze a method for matrix multiplication with several nested loops, which can be solved by means of a different query for each loop.

Reverse of an Array We want to infer the cost of a simple reverse method which reverses the elements of an array. The recursive representation of reverse in our system takes the form reverse(a, i, r), where a represents the input array, i is the local variable and r is the resulting array. Basically, the execution time depends on the number of loop iterations; therefore, relevant variables are those appearing in the guard of the recurrence relation for m2 (which denotes the termination condition of the loop). Only a and i appear in the cost relation yielded by our system, while r is removed. The size analysis abstracts the array a to its *length* and infers that the variable i decreases by one unit in each iteration.

In order to solve the recurrence equations in Mathematica, we need to use the same variable name in all

Cost relations and Mathematica solution for Array Reversal

```
static int[ ] reverse(int[ ] a){
  int la = a.length;
  int[ ] r = new int[la];
  for (int i=la ; i > 0 ; i--) r[la-i]=a[i-1];
  return r;
}
```

```
reverse[a] == m0[a],
m0[a]      == 8 + m1[a],
m1[i]      == 2 + m2[i],
m2[0]      == 2,
m2[i]      == m4[i],
m4[i]      == 12 + m1[i-1]
```

Mathematica **Query:** RSolve[{ rev[a] == m0[a], m0[a] == 8 + m1[a-1], m1[a] == 2 + m2[a], m2[0] == 2, m2[a] == m4[a], m4[a] == 12 + m1[a-1]}, {rev[a],m0[a],m1[a],m2[a],m4[a]}]

Mathematica **Answer:** reverse[a] − > 12 (1 + 2 a)

Figure 3.16: Array Reversal

Cost relations and Mathematica solution for Array Concatenation

```
static int[ ] concat(int a[ ], int b[ ]) {
  int l1 = a.length;
  int l2 = b.length;
  int[ ] r = new int[l1+l2];
  int i = 0;
  for (i=0;i<l1;i++) r[i]=a[i];
  for (i=l1;i<l1+l2;i++) r[i]=b[i];
  return r;
}
```

```
concat[a,b] == m0[a,b],
m0[a,b] == 15 + m1[a,b,0],
m1[a,b,i] == 3 + m2[a,b,i],
m2[a,b,i] == m3[a,b,i],          i ≥ a
m2[a,b,i] == m4[a,b,i],          i < a
m3[a,b,i] == 2 + m5[a,b,b],
m4[a,b,i] == 8 + m1[a,b,i+1],
m5[a,b,i] == 5 + m7[a,b,i],
m7[a,b,i] == 2,                  i ≥ a+b
m7[a,b,i] == m8[a,b,i],          i < a+b
m8[a,b,i] == 8 + m5[a,b,i+1],
```

Mathematica **queries:**

```
RSolve[{ m1[i] == 3 + m2[i], m2[a] == 2 + k, m2[i] == m4[i],
         m4[i] == 8 + m1[i+1] }, {m1[i],m2[i],m4[i]},i]
RSolve[{ m5[i] == 5 + m7[i], m7[a+b] == 2, m7[i] == m8[i],
         m8[i] == 8 + m5[i+1]}, {m5[i],m7[i],m8[i]},i]
```

Mathematica **answers:** m1[i] -> 5 + 11 a - 11 i + k (k ->m5[b]) m5[i] -> 7 + 13 a + 13 b - 13 i

Solution (composition of the answers):

concat[a,b] -> 15 + m1[0] ≡ 15 + 5 + 11 a + m5[b] ≡ 27 + 24 a

Figure 3.17: Array Concatenation

equations, i.e., we cannot have both a and i. This is because, otherwise, Mathematica requires all variables to be passed from the initial equation on (see also Sec. 3.4.6). Note that this renaming can be easily done in an automatic way (the result can be seen in the RSolve query).

Concatenation of Two Arrays Consider the method concat in Fig. 3.17: it concatenates two input arrays a and b and returns the result in c. The equation concat[a, b] corresponds to the cost of calling concat with two arrays with length a and b, and m0[a, b] corresponds to the initialization of the local variables. The loops correspond respectively to the equations: (1) m1[a, b, i], m2[a, b, i] and m4[a, b, i]; and (2) m3[a, b, i] m5[a, b, i], m7[a, b, i] and m8[a, b, i].

The size analysis was able to infer the increase in the loops' counters and their corresponding initial values; slicing removed the variable r, which is irrelevant to the cost. The major limitations we found in Mathematica are:

1) it is impossible to include guards in the recurrence equations;

2) variables cannot be repeated in the equation head;

3) all equations must have at least one variable argument;

Cost relations and Mathematica solution for Matrix Multiplication

```
static int[ ][ ] mult(int[ ][ ] a,int[ ][ ] b,
                      int r, int c) {
  int[ ][ ] c1 = new int[r][c];
  for(int i=0; i < r;i++)
    for( int j =0; j < c; j++)
      for (int k=0; k < c; k++)
        c1[i][j] = c1[i][j] + (a[i][k] *a[k][j]);
  return c1;
}
```

```
mult[r,c] == 16 + m0[r,c,0],
m0[r,c,i] == 3 + m1[r,c,i],
m1[r,c,i] == 0                  i ≥ r
m1[r,c,i] == m2[r,c,i]          i < r
m2[r,c,i] == 4 + m3[r,c,0] + m0[r,c,i+1]
m3[r,c,j] == 3 + m4[r,c,j],
m4[r,c,j] == 0,                 j ≥ c
m4[r,c,j] == m5[r,c,j],         j < c
m5[r,c,j] == 4 + m6[r,c,0] + m3[r,c,j+1]
m6[r,c,k] == 3 + m7[r,c,k],
m7[r,c,k] == 0,                 k ≥ c
m7[r,c,k] == m8[r,c,k],         k < c
m8[r,c,k] == 24 + m6[r,c,k+1]
```

Mathematica **queries:**

```
RSolve[{m0[i] == 3 + m1[i], m1[r] == 0, m1[i] == m2[i],
        m2[i] == 4 + k + m0[i+1]}, {m0[i],m1[i],m2[i]},i]

RSolve[{m3[j] == 3 + m4[j], m4[c] == 0, m4[j] == m5[j],
        m5[j] == 4 + z + m3[j+1]},{m3[j],m4[j],m5[j]},j]

RSolve[{m6[k] == 3 + m7[k], m7[c] == 0, m7[k] == m8[k],
        m8[k] == 24 + m6[k+1]}{m6[k],m7[k],m8[k]},k]
```

Mathematica **answers:**

```
m1[i] -> 3 - 7 i - i k + 7 r + k r        (k = m3[0])
m3[j] -> 3 + 7 c - 7 j + c z - j z        (z = m6[0])
m6[k] -> 3 (1 + 9 c - 9 k)
```

Solution: mul-> 16+m1[0]≡19+7r+rm3[0] ≡19+7r+r(3+7c+cm6[0]) ≡ 19+10r+10rc+27c^{2}r

Figure 3.18: Matrix multiplication

4) variables in the equation head must appear in the body.

Regarding limitation 1), we can notice in the equations for m2 that recursion ends when i = a. Therefore, we could write the two equations for m2 as follows: m2[a, b, a] == m3[a, b, a], m2[a, b, i] == m3[a, b, i]. The same process can be applied to the equations for m7, which can be transformed to m7[a, b, a + b] == 2, m7[a, b, i] == m8[a, b, i]. This reformulation is still not acceptable by Mathematica, because there are repeated variables in the head of the rules (point 2). Yet, we observe that the first two arguments of the relation, a and b (i.e., the array lengths), remain constant through the relation. Therefore, we can safely (and automatically) remove them from *all* the equations. However, this transformation incurs problems 3) and 4). Problem 3 appears because the first two equations do not have variables anymore; this prevents us from including them in the Mathematica query (rather, we can use them only at the end, to compose the final solution). Furthermore, when i is initialized to the length of the array b in the equation m3, i.e., we have m3[i] == m5[b], problem 4) occurs. In order to overcome problem 4) (which will indeed appear frequently), we treat m5[b] as a constant (k is used in the table) and replace it in all the equations. This involves the execution of two different queries in Mathematica, as it can be seen above: one for m1[i], and one for m5[i]. The final complexity is obtained by composing the results (taking into account that k = m5[b]) with the initial equations, which have no variables.

We want to point out that, although the above transformations could be done automatically (and we could produce recurrence relations which are directly solvable in Mathematica), we have not implemented them in our system because we are still studying which solver is more appropriate for our needs. Indeed, Mathematica is a rather complex software which offers much more than is needed in order to solve recurrence equations; therefore, we might want to process the output of our system with a simpler software, like PURRS [**?**], which is indeed dedicated to solve recurrence equations.

Matrix Multiplication Consider the method mult in Fig. 3.18, which implements the multiplication of (a subset of) two matrices. The first two arguments are the matrices to be multiplied, and r and c are the number of rows and columns to be taken into account. As a novel feature, mult presents nested loops. This requires a special processing of the CFG which detects and extracts loops.

The equations m0[r, c, i], m1[r, c, i] and m2[r, c, i] correspond to the outermost loop; m3[r, c, j], m4[r, c, j] and m5[r, c, j] corresponds to the middle loop; and m6[r, c, k], m7[r, c, k] and m8[r, c, k] correspond to the innermost loop. Note that size analysis was able to infer the increase of the loops' counters, and that slicing was able to remove variables which are irrelevant to the cost.

The inferred recurrence equations are not solvable by Mathematica. We basically need to apply the same transformations explained in Sect. 3.4.6 to make the equations solvable (and overcome the previously mentioned limitations). Very briefly, we first simplify all guards by applying them to the equation heads. Then, we remove parameters f and c from the equations, since they are constant in all of them. Finally, we input three separate queries to Mathematica, one for each loop. In the end, the results obtained for the three loops are composed in the initial equation (we could not include it in the query as it has no arguments).

Dealing with Object-Oriented Features

In this section, we study several object-oriented features. First, we see how we deal with dynamic dispatching in the context of cost analysis. Then, we analyze the cost of reversing a list implemented as a class with field attributes. Finally, we infer the cost of a linear search algorithm over the list. To the best of our knowledge, these examples illustrate novel object-oriented features which are not studied in existing cost analyses for other languages and paradigms.

Dynamic dispatching The Incr example in Fig. 3.19 above presents interesting object-oriented features, such as the use of objects and the invocation of methods with dynamic dispatching. In particular, as it is not known at compile time which of the three methods (A.inc, B.inc or C.inc) will be executed, we need to consider the different costs obtained for each case. Therefore, the *object* o which determines which method will be executed becomes part of the guards in the cost relation. It can be seen in the equation for m4 that, depending on whether the object o belongs to class A, B, or C, we have a different cost. We can apply all the transformations discussed in Sect. 3.4.6 in order to make the equations solvable in Mathematica (i.e., apply the guards for i, eliminate variable n from all equations, etc). However, we cannot apply the guards which distinguish the type of the object to the equation head. Our proposal consists of generating three different sets of recurrence equations (one corresponding to each method invocation). We can now get rid of variable o in all sets of equations. This leads to the three Mathematica queries written in the table. We named the result for each one as add_X, where X is the type of object for which the cost was computed. As the Mathematica answer is rather large for add_B and add_C, we did not written the constant parts in the table. Then, depending on whether one is interested in upper or lower bounds of the computational cost, we compute the maximum or the minimum of the three solutions: clearly, add_A provides an upper bound and add_C a lower bound of the computational cost.

List Processing Algorithms The class List (Fig. 3.20) contains a procedure which computes the reverse of a list implemented as a class with two fields: next, which points to the next element in the list, and data, which contains the information stored in the list. The equations inferred by the analyzer are depicted in the table. Recall that, in the recurrence equations, x stands for the length of paths reachable from x. The size analysis was able to infer that the path length of x is decreasing by one in every two consecutive visits of the loop, and that slicing was able to remove all variables that do not affect the loop condition. The output recurrence equations can be directly solved in Mathematica. We obtained linear complexity as it is shown in the table.

Finally, the last example Search (Fig. 3.21) implements the linear search of an element e in an input list x. It uses the List class, and returns the element of x whose data field is equal to e. The novel feature of this example is that we have two conditions on the loop, and the second one depends on the content of

Cost relations and Mathematica solution for Dynamic Dispatching

```
class A {
  int incr(int i) {return i+1; }};
class B extends A {
  int incr(int i) {return i+2; }};
class C extends B {
  int incr(int i) {return i+3; }};
class Incr {
  int add(int n, A o) {
    int res=0;
    int i=0;
    while (i <=n) {
     res = res + i;
     i = o.incr(i);}
    return res; }};
```

```
add[n,o]   ==  m0[n,o],
m0[n,o]    ==  4 + m1[n,o,0],
m1[n,o,i]  ==  3 + m2[n,o,i],
m2[n,o,i]  ==  2,                    i > n
m2[n,o,i]  ==  m3[n,o,i],            i ≤ n
m3[n,o,i]  ==  7 + m4[n,o,i],
m4[n,o,i]  ==  A:incr[i] + m5[n,o,i+1],   o ∈ A
m4[n,o,i]  ==  B:incr[i] + m5[n,o,i+2],   o ∈ B
m4[n,o,i]  ==  C:incr[i] + m5[n,o,i+3],   o ∈ C
m5[n,o,i]  ==  2 + m1[n,o,i],
A:incr[i]  ==  3,
B:incr[i]  ==  3,
C:incr[i]  ==  3,
```

Mathematica **query:**

```
RSolve[{m1[i] == 3 + m2[i], m2[n] == 2, m2[i] == m3[i], m3[i] == 7 + m4[i],
        m4[i] == A + m5[i+1], m5[i] == 2 + m1[i], A[i] == 3 },
        {m1[i],m2[i],m3[i],m4[i],m5[i],A[i]},i]

RSolve[{m1[i] == 3 + m2[i], m2[n] == 2, m2[i] == m3[i], m3[i] == 7 + m4[i],
        m4[i] == B[i] + m5[i+2], m5[i] == 2 + m1[i], B[i] == 3},
        {m1[i],m2[i],m3[i],m4[i],m5[i],B[i]},i]

RSolve[{ m1[i] == 3 + m2[i], m2[n] == 2, m2[i] == m3[i], m3[i] == 7 + m4[i],
        m4[i] == C[i] + m5[i+3], m5[i] == 2 + m1[i], C[i] == 3},
        {m1[i],m2[i],m3[i],m4[i],m5[i],C[i]},i]
```

Appr. of Mathematica **answers:** $\text{add}_A \approx 15n + K$ $\text{add}_B \approx 7.5n + K$ $\text{add}_C \approx 5n + K$

Figure 3.19: The `Incr` program

Cost relations and Mathematica solution for List Reversal

```
class List {
  List next;  int data;
  public List reverse(List x) {
   List result = null;  List tmp = null;
   while ( x != null ) {
    tmp = x.next;  x.next = result;
    result = x;  x = tmp;
  }
  return result; }}
```

```
reverse[x]  ==  m0[x],
m0[x]       ==  4 + m1[x],
m1[x]       ==  2 + m2[x],
m2[0]       ==  2,
m2[x]       ==  m4[x],
m4[x]       ==  11 + m1[x-1],
```

Mathematica **query:**

```
RSolve[{rev[x] == m0[x], m0[x] == 4 + m1[x], m1[x] == 2 + m2[x],
        m2[0] == 2, m2[x] == m4[x], m4[x] == 11 + m1[x-1]},
        {rev[x],m0[x],m1[x],m2[x],m4[x]},x]
```

Mathematica **answer (complexity):** rev[x] -> 8 + 19 x

Figure 3.20: List reversal

the list. From the recurrence equations, we observe that the equations m8 correspond to the first guard in the loop condition. In particular, the first one is the exit condition of the loop when the list is null, i.e., $x = 0$. The second one, $x \neq 0$, leads to the equations n1, where the second condition is evaluated. Variable d in this guard represents x.data. Exiting form the loop depends on whether d is equal to e. Mathematica cannot handle these recurrence equations, due to the fact that they involve two guards (and one should consider the best and the worst case). Besides, it is not possible to express the second guard in a way which is understandable to the solver. The approach we propose consists of *approximating* the solution by disregarding the second guard of the loop. This implies that we delete the first equation for n1 from the set of equations, and the remaining guard $d \neq e$. As a consequence, variables e and d become now irrelevant

Cost relations and Mathematica solution for List Manipulation

```
class Search {
  public List search(List x, int e) {
  int index=1;
   while ( x != null && x.data != e ) {
      index++;
      x = x.next;
   }
    return x;
}
```

```
search[x,e] == m5[x,e],
m5[x,e] == 7 + m6[x,e] + m7[c],   c ≤ x
m6[x,e] == 2 + m8[x,e],
m8[0,e] == 0,
m8[x,e] == m9[x,e],
m9[x,e] == 4 + n1[x,e,d],
n1[x,e,d] == 0,                   d = e
n1[x,e,d] == n0[x,e,d],           d ≠ e
n0[x,e,d] == 5 + m6[x-1,e],
m7[c] == 2
```

Mathematica **query:**

```
RSolve[{ search[x] == m5[x], m5[x] == 9 + m6[x], m6[x] == 2 + m8[x], m8[0] == 0,
    m8[x] == m9[x], m9[x] == 4 + n1[x], n1[x] == n0[x], n0[x] == 5 + m6[x-1]},
        {search[x],m5[x],m6[x],m8[x],m9[x],n0[x],n1[x]},{x}]
```

Mathematica **answer (upper bound complexity):** search[x] -> 11 (1+x)

Figure 3.21: List Manipulation

Benchmark	BC	CFG	RR	Size An.	Slicing	Cost	Total
Hanoi	289	15	5	150	15	3	187
Fibonacci	298	19	6	265	39	2	331
Reverse	296	21	5	207	21	2	256
Concat	351	64	7	648	43	4	766
MatMult	388	182	12	2152	115	5	2465
Incr	320	38	13	956	371	7	1383
List	355	27	4	123	58	3	216
Search	351	51	12	462	220	4	750
Diff	377	167	14	3804	595	10	4590
Intersec	390	181	18	4575	869	15	5657
Sum	295	62	8	1415	287	5	1776

Table 3.1: Measured time (in ms) of the different phases of cost analysis

and are sliced away. Note that we will obtain an upper bound solution for the computational cost, rather than the exact solution. This reasoning is not easy to automate, and our system still cannot deal with it automatically. Besides, it should be noted that, in order to solve the equations in Mathematica, we need to unfold m7 in order to eliminate the guard of m5. After all these (non trivial) simplifications, Mathematica provides a linear complexity as the upper bound.

Experiments and Discussion

In order to assess the practicality of our cost analysis framework, we have implemented a prototype analyzer in Ciao [89]. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9.

Table 1 shows the run-times of the different phases of the cost analysis process. The first column, **Benchmark**, indicates the name of the class and method of the benchmark to be analyzed. The second column, **BC**, contains the size in bytes of the corresponding .class. All other columns show execution times in milliseconds and have been obtained using the statistics/2 procedure of Ciao with the parameter runtime. They are computed as the arithmetic mean of five runs. For each benchmark, **CFG** represents the time taken to build the control flow graph of the corresponding method; **RR** is the time taken for obtaining the recursive representation from the CFG (this includes translating bytecode operations for converting stack positions into local variables and removing irrelevant variables by means of slicing); **Size An.** is the time

taken by the abstract-interpretation based size analysis for computing size relations; **Slicing** shows the time required for detecting the set of variables which are relevant in each block of the CFG; finally, **Cost** stands for the time taken to build the cost relations for the different blocks.

The benchmarks are divided into four categories, as it can be seen from the structure of the table: (i) recursive procedures (Sec. 3.4.6) solving Hanoi and Fibonacci problems; (ii) methods involving (possibly nested) loops, as array reverse and concatenation, and matrix multiplication (Sec. 3.4.6); (iii) procedures manipulating objects and fields (Sec. 3.4.6), as the add method involving dynamic dispatching, and list reversal and search; (iv) further examples: computing the difference (diff) and the intersection (intersec) of two arrays, and the function sum computing $\Sigma_{i=1}^{n}\ \Sigma_{j=1}^{i}\ i+j$.

As the figure shows, the total times obtained using our prototype implementation range from 187 ms in the case of Hanoi, to 5657 ms in the case of Intersec. As it can be seen, computing size relations is the most expensive step. This comes from the fact that this step requires a global analysis of the program, whereas **CFG**, **RR**, and **Cost** basically involve a single pass on the code. **Slicing** also requires a global, though much simpler, analysis. Thus, the time it requires is the biggest after the size analysis.

Our experimental results are very preliminary, and there is still plenty of room for optimization (mainly in the size analysis phase). The main planned optimization is the use of abstract compilation techniques in order to avoid re-computation of abstract operations which are related to the bytecodes. This can be done since the analysis is denotational, so that those bytecodes will always have the same abstract approximations.

As regards the accuracy of the analysis, our approach was able to obtain accurate cost relations for all the considered benchmarks. Note that this is an important observation, since we are confident that, by further transformations on the cost relations, or by using a more powerful system for solving recurrence equations, we will be able to obtain closed form solutions for a broader class of programs.

3.4.7 Conclusions and Future Work

We have presented an automatic approach to the cost analysis of Java bytecode, based on generating at compile-time cost relations for an input bytecode program. Such relations are functions of input data which are informative by themselves about the computational cost, provided an accurate size analysis is used to establish relationships between the input arguments. Essentially, the sources of inaccuracy in size analysis are: 1) guards depending (directly of indirectly) on values which are not handled in the abstraction, e.g., non-integer values, numeric fields or multidimensional arrays, cyclic data-structures; 2) loss of precision due to the abstraction of (non-linear) arithmetic instructions and domain operations like widening. In such cases, we can still set up cost relations; however, they might not be useful if the size relationships are not precise enough.

To the best of our knowledge, our work presents the first approach to the automatic cost analysis of Java bytecode. Related work in the context of Java bytecode includes the work in the MRG project [20], which can be considered complementary to ours. MRG focuses on building a proof-carrying code [138] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. Also, the resource which has been studied in more depth is heap consumption, since applications to be deployed on devices with a limited amount of memory, such as smartcards, must be rejected if they require more memory than that available. Another related work is [53], where a resource usage analysis is presented. Again, this work focuses on memory consumption and it aims at verifying that the program executes in bounded memory by making sure that the program does not create new objects inside loops. The analysis has been certified by proving its correctness using the Coq proof assistant.

As future work, we plan to apply this analysis to estimate heap consumption in programs which make intensive use of OO features. Also, as seen in Section, existing computer algebra systems are, in many cases, not directly applicable to obtain upper bounds of cost relations and a high-degree of human intervention is required. Therefore, we plan to work on fully automating the process of obtaining upper bounds for cost relations.

Chapter 4

Alias Control Types

This chapter presents the work done in the development and formalisation of alias control type systems for object-oriented programming languages. These type systems serve to characterize which reference types may be aliases of other reference types, and thus help to characterize the effects of expression evaluation, and to restrict the spaces necessary for program verification. Universes are being used, *e.g.,* in ESC/Java2 and JML2. In Task 2.5 we developed powerful but lightweight alias type systems for the source and byte code language, tailored for local reasoning, for use in WP3.

Section 4.1 builds on previous work done in [127, 71] and formalises UJ, an adaptation of the Universe type system for a Java-like language. In particular this work separates the topological aspect of Universes from the encapsulation aspect into two distinct type systems. The encapsulation aspect is crucial in PCC certification; for midlet certification it is important to be able to guarantee that no references to internal data structures are exposed.

Section 4.2 outlines an extension of the type system of UJ, which can be used to certify the absence of race conditions in a program. The approach is similar to that from [79, 50] with two significant advantages: the type system is based on universes, and thus simpler; the use of anyin some cases allows fewer locks to need to be taken, and more operations to be atomic. The work has been published at [60].

Section 4.3 extends UJ to generic class definitions; the resulting type system is called GUT. GUT is an essential development, given that generics are an integral part of Java, or indeed any mainstream strongly-typed language. The resulting system is lightweight, and supports parameterization both with respect to the classes of the entities in the container, and also with respect to the topology of these entities. Thus, GUT can express finer grained topological properties than UJ could. The work has been published at [68].

Finally, Section 4.4 explores a novel extension to ownership type systems (which generally organise heaps as trees), to allow for multiple owners (thus organising heaps as directed acyclic graphs), and which extends the domain where such type systems can be applied to sensibly describe the effects of computations. This work will appear at [54].

4.1 UJ: Type Soundness for Universe Types

Aliasing is an integral part of imperative object-oriented programming, as e.g. in Smalltalk, Java and C++. It allows a natural presentation of real situations such as the sharing of state. However, it makes several other programming aspects more difficult, such as reasoning about programs, garbage collection and memory management, code migration, parallelism, and the analysis of atomicity.

To address these issues, type systems such as Universes [127, 71], ownership types [58, 57, 51] and similar others [79, 16] have been introduced. They come in various flavours, but all have in common that they organize the heap *topology* as a tree (or forest) where each object *is owned* by at most a *single* object and the ownership relationship is acyclic. It is common practice to depict ownership through a box in the object graph. For example, in Figure 4.1 the dotted box around object 1 of class Bag indicates that it owns object 2, while the object 2 of class Stack owns objects 3, 4 and 5. Objects that are not owned by any

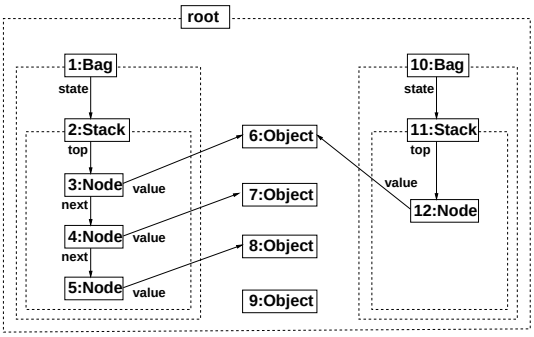

Figure 4.1: Depicting object ownership and references in a heap

```
1  class Bag {
2      rep Stack state;
3      public impure void add(any Object o) { this.state.push(o); }
4      public pure Bool isEmpty() { return (this.state.top == null); }
5  }
6
7  class Stack {
8      rep Node top;
9      public impure void push(any Object o) { this.top := new rep Node(o, this.top); }
10     public impure any Object pop() {
11         any Object o := null;
12         if (this.top != null) { o := this.top.value; this.top := this.top.next; }
13         return o;
14     }
15 }
16
17 class Node{
18     any Object value;
19     peer Node Next;
20 }
```

Figure 4.2: Code augmented with universe annotations

other object, such as 1, 6 and 9 are contained within the dotted box labeled root, which gives us a tree (as opposed to a forest).

This hierarchic heap topology can be exploited in several ways: Vitek et al. [16] speed up garbage collection, because as soon as an owner is unreachable, all owned objects are unreachable too. Flanagan et al. [79], Boyapati et al. [50], Cunningham et al. [60] can guarantee that a program will not have races, because locking an object implicitly locks all owned objects. Clarke et al. [57] use ownership types to calculate the effects of computation, and thus determine when they do not affect each other. Clarke et al. [58] introduce deep ownership, whereby any path to an object from the root must go through its owner; thus the owned objects are its *representation*, and are encapsulated within the object. Leavens et al. [130] use the hierarchic topology for defining modular verification techniques of object invariants.

Universe types were developed by Müller and Poetzsch-Heffter [129, 127] to support modular reasoning. The type system is one of the simplest possible in the family of ownership related systems. There are three universe annotations: rep, peer and any, which denote the relative placement of objects in the ownership hierarchy.

An example appears in Figure 4.2. The annotation rep (short for representation) expresses that the

object is owned by the currently active object, while peer expresses that the object has the same owner as the currently active object. In terms of our example code, the Stack object state in class Bag is declared as rep and indeed, we see that in Figure 4.1 the Stack objects 2 and 11 are owned by Bag objects 1 and 10 respectively; similarly, in class Node the field next is declared as peer, and indeed the Node objects 3, 4 and 5 have the same owner. The annotation any abstracts over the object's position in the hierarchy: the field value in class Node is declared as any and can therefore refer to any object in the hierarchy. The code in Figure 4.2 also augments methods with the keywords pure and impure, qualifying read-only methods from methods that affect the state of the receiver; though not integral to Universes, these purity annotations are used to guarantee encapsulation, as we discuss later.

Thus, Universe types offer a very simple system which describes the topology of the heap. In contrast with several other systems [58], Universe types do not restrict access into the boxes, as long as they are carried out through any references. Thus, a reference from 3 to 12 would be legal through the field value, because this field has the type any Object.

Nevertheless, Universe types impose *encapsulation*, in the sense that they guarantee that the state of an object can only be modified when one of the object's owners is the currently active object. This is the *owners-as-modifiers* property [127], and it is guaranteed by the type system by requiring that only pure methods can be called when the receiver has the universe annotation any. Note however, that Universe types can be used to guarantee race free programs, without requiring the owners-as-modifiers discipline [60].

Thus, the owners-as-modifiers discipline supports modular reasoning, because it guarantees that the invariant of an object is preserved by an execution, whose receiver's owner is not an transitive owner of the former object. Modular reasoning using Universes has been adopted in Spec# [23, 24]. However, there they also consider a setting where the owners may change dynamically. The use of Universe types in modular reasoning is described in various literature, amongst which [106, 71, 130].

We give a formal, type theoretical description of Universe types. We distinguish the topological type system from the encapsulation type system, and describe the latter on top of the topological system. The reasons for this distinction are:

- we believe the distinction clarifies the rationale for the type systems,
- some systems, such as those required for races detection, atomicity and deadlock detection, only require the topological properties,
- it is conceivable that one might want to add an encapsulation part onto a different topological type system.

Universe types were already introduced and proven sound in [127], but the description was in terms of proof theory, rather than the type theoretic machinery we adopt in this report; they were further described in [71]. Extensions to Universe types, such as those described in [68] (adding generics to Universe types), already use the type theoretic approach.

This section thus aims to fill a gap in the Universe types literature, by giving a full type theoretical account of the basic type system, proofs, sufficient examples, explanations, and elucidate the distinction between the topological and the encapsulation system. We describe Universe types for UJ (Universe Java), a Java-like language, which contains classes, inheritance, fields, methods, dynamic method binding, and casts.

Outline Sec. 4.1 is organized as follows: we introduce our base language in Section 4.1.1, followed by a discussion on Universes and owners in Section 4.1.2. In Section 4.1.3 we give the operational semantics for our language. Section 4.1.4 presents the topological type system and states subject reduction. Section 4.1.5 covers the encapsulation type system and its relation to modular verification. Section 4.1.6 concludes.

4.1.1 UJ Source Language

UJ models a subset of Java supporting class definitions, inheritance, field lookup and update, method invocations and down-casting. On top of this core subset, we augment types with universe annotations and

method signatures with purity annotations.

Source Program

The syntax of our source language is given in Figure 4.3. We assume three infinite but distinct sets of names: one for classes, $c \in Id_c$, one for fields, $f \in Id_f$, and another for methods, $m \in Id_m$. A program is defined in terms of three partial functions, $\mathcal{F}$, $\mathcal{M}$ and $\mathcal{MBody}$, and a transitive closure relation over class names $\leq_c$; these functions and relations are global. $\mathcal{F}$ associates field names in a class to types, $\mathcal{M}$ associates method names in a class to method signatures and $\mathcal{MBody}$ associates method names in a class to method bodies. The reflexive transitive closure relation $\leq_c$ denotes class inheritance.

Source expressions, denoted by e, are a standard subset of Java. They consist of the basic value, null, the self reference this, parameter identifier, x, new object creation, new t, down-casting , $(t)\ e$, field access, $e.f$, field update, $e.f := e$, and method invocation, $e.m(e)$. Types, denoted by t, constitute a departure from standard Java. They consist of a pair, $w\,c$, where class names c are preceded by universe annotations, denoted as w. Method signatures also deviate slightly from standard Java. They consist of a triple, denoted as $p : t_1\ (t_2)$ where t_2 is the type of the method parameter, t_1 is the return type of a method, and p is a purity tag, ranging over the set $\{\mathsf{pure}, \mathsf{impure}\}$.

Universes, as defined in [127], are denoted by w and range over the set $\{\mathsf{rep}, \mathsf{peer}, \mathsf{any}\}$. They provide *relative information* with respect to the current object. More specifically,

- rep states that a reference constitutes part of the current object's *direct representation*. Stated otherwise, the current object *owns* the object pointed to by the reference.
- peer states that the reference constitutes part of the representation the current object belongs to. Stated otherwise, the current object and the object pointed to by the reference are *owned* by the same object (owner).
- any is a form of existential quantification over such information.

For our formalisation we need to extend Universes by another value, self, and refer to the extended set as *Extended Universes*, denoted by u.

- self is a specialisation of peer, referring to the object itself and not any other peer object in the direct representation the current object belongs to.

We find it convenient to identify a universe subset called *Concrete Universes*, z, ranging over $\{\mathsf{self}, \mathsf{rep}, \mathsf{peer}\}$ but not any, which give us concrete representation information about references with respect to the current object. When writing source programs, we only make use of universes w; extended universes u are used extensively in the topological type system (Section 4.1.4); concrete universes are used extensively in the encapsulation type system (Section 4.1.5).

4.1.2 Universes and Owners

Universes characterise object aliasing in a heap because they structure the heap as an acyclic ownership tree. Every object a in a heap is owned by a single owner, o, which is either another object in the heap or the root of the ownership tree, root. The *representation* of an object in a heap h is defined as all the objects it transitively owns (all the objects below it). Ownership is acyclic, that is any two distinct objects in h cannot belong to one another's representation (they cannot transitively own one another). When we do not want to refer to a particular heap, we find it convenient to refer to objects as pairs with their owners a, o. (*e.g.* 1,root and 2,1, meaning 1 owned by root and 2 owned by 1 respectively).

We recall that (concrete) universes are *relative* with respect to a viewpoint and do not mean anything without such a viewpoint. In class definitions, this viewpoint of universe annotations is implicitly assumed to be the current this object (see the code in Figure 4.2). An object a,o can be assigned to a universe with respect to another object a',o' using the judgement

$$a', o' \vdash a, o : u \tag{4.1}$$

$$
\begin{array}{rcl}
\mathcal{F} & : & Id_c \times Id_f \rightharpoonup Type \\
\mathcal{M} & : & Id_c \times Id_m \rightharpoonup TypeSig \\
\mathcal{MBody} & : & Id_c \times Id_m \rightharpoonup SrcExpr \\
\leq_c & : & Id_c \times Id_c \to Bool
\end{array}
$$

$$
\begin{array}{rcl}
e \in SrcExpr & ::= & \mathsf{this} \mid \mathsf{x} \mid \mathsf{null} \mid \mathsf{new}\ t \mid (t)\ e \\
 & & \mid e.f \mid e.f := e \mid e.m(e) \\
t \in Type & ::= & u\ c \\
w \in Universe & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{any} \\
u \in Extended\ Universes & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{any} \mid \mathsf{self} \\
z \in Concrete\ Universes & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{self} \\
TypeSig & ::= & p : t\ (t) \\
p \in Purity\ Tag & ::= & \mathsf{pure} \mid \mathsf{impure}
\end{array}
$$

Figure 4.3: Source program definition

$$\frac{}{a, o \vdash a, o : \mathsf{self}}(\textsc{Self}) \qquad \frac{}{\_, o \vdash \_, o : \mathsf{peer}}(\textsc{Peer})$$

$$\frac{}{a, \_ \vdash \_, a : \mathsf{rep}}(\textsc{Rep}) \qquad \frac{}{\_, \_ \vdash \_, \_ : \mathsf{any}}(\textsc{Any})$$

Figure 4.4: Assigning universes to objects

which is defined as the least relation satisfying the rules in Figure 4.4[1]. It states that, from the point of view of a' (owned by o'), a (owned by o) has universe u.

Example 4.1.2.1 (Universes and addresses)**.** *In the heap depicted in Figure 4.1, from the point of view of 2 (owned by 1) object 3 (owned by 2) has universe* rep *that is*

$$2, 1 \vdash 3, 2 : \mathit{rep} \tag{4.2}$$

Similarly, we can derive

$$3, 2 \vdash 4, 2 : \mathit{peer} \tag{4.3}$$

Also, we can assign any *to any address from any viewpoint using rule (*ANY*). Thus,*

$$3, 2 \vdash 6, \mathit{root} : \mathit{any} \qquad 2, 1 \vdash 3, 2 : \mathit{any} \qquad 3, 2 \vdash 4, 2 : \mathit{any} \tag{4.4}$$

Universe Ordering

We define the following ordering for universes $u \leq_u u'$:

$$\mathsf{self} \leq_u \mathsf{peer} \leq_u \mathsf{any} \qquad \mathsf{rep} \leq_u \mathsf{any}$$

It states that both peer and rep are sub-universes of any, but they are not directly related, and also that self is a sub-universe of peer. In Lemma 4.1.2.2 we state that the universe ordering relation ($\leq_u$) is consistent with judgement (4.1). Thus any address that is assigned rep, peer and self can also be assigned universe any and any address that is assigned self can be assigned universe peer, as we have already seen in Example 4.1.2.1.

[1]The notation _ indicates any value

u \ $u \oplus w$	w = peer	w = rep	w = any
self	peer	rep	any
peer	peer	any	any
rep	rep	any	any
any	any	any	any

u \ $u \ominus w$	w = peer	w = rep	w = any
self	peer	rep	any
peer	peer	any	any
rep	any	peer	any
any	any	any	any

Figure 4.5: Universe composition and decomposition

Lemma 4.1.2.2 (Universe Address Judgements respect Universe Ordering)**.**

$$\left.\begin{array}{l} a, o \vdash a', o' : u \\ u \leq_u u' \end{array}\right\} \implies a, o \vdash a', o' : u'$$

Example 4.1.2.3 (Subtyping and Universes)**.** *We can see how the field universe annotations in the classes of Figure 4.2 restrict the field references in Figure 4.1. For instance, 2 has the* rep top *field correctly assigned to 3, since from (4.2) above we know that 3 has universe* rep *with respect to 2. In fact this reference can only point to objects 3, 4 and 5 since these are the only objects owned by object 2. Similarly, 3 has the* peer next *field correctly assigned to 4 which is owned by the same owner of 3; from (4.3) above. Trivially, the* any value *field of 3 assigned to 6 also respects the universe annotation because of (4.4) above. It can however point to any object in the heap since any type t is a subtype of* any Object*.*

Universe Composition and Decomposition

The universe information given by Universe types is *relative* with respect to a particular viewpoint. To translate Universe types from one viewpoint to another we define composition and decomposition operators over extended universes, u, and universes, w. These operators are denoted as $u \oplus w$ and $u \ominus w$ respectively and are described in Figure 4.5.

- Universe composition is used to determine the universe of a reference that is *twice removed* from our current viewpoint. If the second reference is outside the current viewpoint's range, then we cannot express it in terms of the concrete universes rep and peer; in such cases $u \oplus w$ is assigned to any. For instance $\mathsf{rep} \oplus \mathsf{rep} = \mathsf{any}$.
- Universe decomposition is the complement of the former operation: $u \ominus w$ returns a universe annotation w' such that $u \oplus w' = w$ if it exists and is unique. For instance $\mathsf{rep} \ominus \mathsf{rep} = \mathsf{peer}$ because $\mathsf{rep} \oplus \mathsf{peer} = \mathsf{rep}$ and there is no other universe w such that $\mathsf{rep} \oplus w = \mathsf{rep}$. When the w' in $u \oplus w' = w$ is not unique or does not exist, then $u \ominus w = \mathsf{any}$. Thus, $\mathsf{rep} \ominus \mathsf{self} = \mathsf{any}$.

In Lemma 4.1.2.4 we show that the intuitions of $(\oplus)$ and $(\ominus)$ are sound with respect to the interpretation of universes as object ownership in a heap, that is judgement (4.1).

Lemma 4.1.2.4 (Sound Universe Composition and Decomposition)**.**

$$\left.\begin{array}{l} a, o \vdash a', o' : u \\ a', o' \vdash a'', o'' : u' \end{array}\right\} \implies a, o \vdash a'', o'' : u \oplus u'$$

$$\left.\begin{array}{l} a, o \vdash a', o' : u \\ a, o \vdash a'', o'' : u' \end{array}\right\} \implies a', o' \vdash a'', o'' : u \ominus u'$$

Example 4.1.2.5 (Composing and Decomposing Universes)**.** *From judgements (4.2), (4.3) from Example 4.1.2.1, using Lemma 4.1.2.4, we derive*

$$2, 1 \vdash 4, 2 : \mathit{rep} \tag{4.5}$$

$$
\begin{array}{rcl}
a, b \in Addr & : & \mathbb{N} \\
o \in Own & : & a \mid \mathsf{root} \\
v \in Val & : & a \mid \mathsf{null} \\
\\
\mathit{flds} \in Flds & : & Id_f \rightharpoonup Val \\
\\
h \in Heap & : & Addr \rightharpoonup (Own \times Id_c \times Flds) \\
\sigma \in Stack & ::= & (Addr \times Val) \\
\\
e \in RunExpr & ::= & v \mid \mathsf{this} \mid \mathsf{x} \mid \mathsf{frame}\ \sigma\ e \mid e.f \mid e.f := e \mid e.m(e) \mid \mathsf{new}\ t \mid (t)\ e \\
E[\cdot] & ::= & [\cdot] \mid E[\cdot].f \mid E[\cdot].f := e \mid a.f := E[\cdot] \mid E[\cdot].m(e) \mid a.m(E[\cdot]) \mid (t)\ E[\cdot]
\end{array}
$$

Figure 4.6: Runtime Syntax

because $\mathsf{rep} \oplus \mathsf{peer} = \mathsf{rep}$. *Conversely, from (4.2) and (4.5), using Lemma 4.1.2.4 and* $\mathsf{rep} \ominus \mathsf{rep} = \mathsf{peer}$, *we recover (4.3)*

$$3, 2 \vdash 4, 2 : \mathit{peer}$$

4.1.3 Operational Semantics

We give the semantics of UJ in terms of a small-step operational semantics. We assume an infinite set of addresses, denoted by a, b. At runtime, a value, denoted by v, may be either an address or null. Owners, denoted by o, can either be any address or the special owner root.

Runtime expressions are described in Figure 4.6. During execution, expressions may contain addresses as values; they may also contain the keyword this and parameter identifiers x. Thus, a runtime expression is interpreted with respect to a heap, h, which gives meaning to addresses, and a stack, σ, which gives meaning to the keyword this and parameter identifier x.

A heap is defined in Figure 4.6 as a partial function from addresses to objects.[2] An object is denoted by the triple (o, c, flds). Every object has an immutable owner o, belongs to a fixed class c, and has a state, *flds*, which is a mutable field map (a partial function from fields names to values). In the remaining text we use the following heap operations [3]:

$$
\begin{array}{rcl}
\mathbf{owner}(h, a) \stackrel{\mathbf{def}}{=} h(a)\downarrow_1 \qquad \mathbf{class}(h, a) \stackrel{\mathbf{def}}{=} h(a)\downarrow_2 \qquad \mathbf{fields}(h, a) \stackrel{\mathbf{def}}{=} h(a)\downarrow_3 \\
h(a.f) & \stackrel{\mathbf{def}}{=} & \mathbf{fields}(h, a)(f) \\
h[(a, f) \mapsto v] & \stackrel{\mathbf{def}}{=} & h\left[a \mapsto \left(\ \mathbf{owner}(h, a), \mathbf{class}(h, a), \mathbf{fields}(h, a)[f \mapsto v]\ \right)\right] \\
h \uplus \{a \mapsto (o, c, \mathit{flds})\} & \stackrel{\mathbf{def}}{=} & h[a \mapsto (o, c, \mathit{flds})] \qquad \text{if } a \notin \mathbf{dom}(h)
\end{array}
$$

The first three operations extract the components making up an object. The fourth operation is merely a shorthand notation for *field access* in a heap. The fifth operation is *heap update*, updating the field f of an object mapped to by the address a in the heap h to the value v. The final operation on heaps is *heap extension* with a new mapping from a *fresh* address a.

A stack σ consists of an address and a value (a, v). The address a denotes the current active object referred to by this whereas v denotes the value of the parameter x. We find it convenient to define the following operations on stacks

$$\sigma(\mathsf{this}) \stackrel{\mathbf{def}}{=} \sigma\downarrow_1 \qquad\qquad \sigma(\mathsf{x}) \stackrel{\mathbf{def}}{=} \sigma\downarrow_2$$

For evaluating method calls, we require to push and pop new address and value pairs on the stack. To model this, runtime expressions also include the expression $\mathsf{frame}\ \sigma\ e$ which denotes that the sub-expression e is evaluated with respect to the inner stack σ.

[2] The arrow $\rightharpoonup$ indicates partial mappings.

[3] SDAs usual, $\downarrow_k$ indicates the k-th projection from a tuple.

$$\frac{}{\sigma \vdash \mathsf{x}, h \rightsquigarrow \sigma(\mathsf{x}), h}(\textsc{rVar}) \qquad \frac{}{\sigma \vdash \mathsf{this}, h \rightsquigarrow \sigma(\mathsf{this}), h}(\textsc{rThis})$$

$$\frac{h' = h \uplus \{a \mapsto \mathbf{initO}(t, h, \sigma)\}}{\sigma \vdash \mathsf{new}\ t, h \rightsquigarrow a, h'}(\textsc{rNew}) \qquad \frac{h, \sigma \vdash a : t}{\sigma \vdash (t)\ a, h \rightsquigarrow a, h}(\textsc{rCast})$$

$$\frac{}{\sigma \vdash a.f, h \rightsquigarrow \mathbf{fields}(h, a)(f), h}(\textsc{rField}) \qquad \frac{h' = h[(a, f) \mapsto v]}{\sigma \vdash a.f := v, h \rightsquigarrow v, h'}(\textsc{rAssign})$$

$$\frac{e = \mathcal{MBody}(\mathbf{class}(h, a), m)}{\sigma \vdash a.m(v), h \rightsquigarrow \mathsf{frame}\ (a, v)\ e,\ h}(\textsc{rCall}) \qquad \frac{\sigma \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash E[e], h \rightsquigarrow E[e'], h'}(\textsc{rEvalCtx})$$

$$\frac{\sigma' \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash \mathsf{frame}\ \sigma'\ e, h \rightsquigarrow \mathsf{frame}\ \sigma'\ e', h'}(\textsc{rFrame1}) \qquad \frac{}{\sigma \vdash \mathsf{frame}\ \sigma'\ v, h \rightsquigarrow v, h}(\textsc{rFrame2})$$

Figure 4.7: Small step operational semantics

Expressions e are evaluated in the context of a heap h and a stack σ. We define the small-step semantics

$$\sigma \vdash e, h \rightsquigarrow e', h' \tag{4.6}$$

in terms of the reduction rules in Figure 4.7. When creating a new object in a heap h, its owner is initialised relative to the universe specified in its declared type t and the current stack σ, whereas all its field values are initialised to null; this initialisation operation is handled by the following function:

$$\begin{array}{l} \mathbf{initO}(u\, c, h, \sigma) \stackrel{\mathbf{def}}{=} (u_{h,\sigma},\ c,\ \{f \mapsto \mathsf{null} \mid \mathcal{F}(c, f) = t\}) \\ \quad \text{where} \\ \qquad u_{h,\sigma} \stackrel{\mathbf{def}}{=} \begin{cases} \sigma(\mathsf{this}) & u = \mathsf{rep} \\ \mathbf{owner}(h, \sigma(\mathsf{this})) & u = \mathsf{peer} \end{cases} \end{array}$$

The above function is partial: it is only defined for universes rep and peer since the owner of a new object cannot be determined if the universe is any. Most of the rules in Figure 4.7 are more or less straightforward. In (rCall) a method call launches a sub-frame with sub-stack σ' to evaluate the body of the method, where $\sigma'(\mathsf{this})$ is the receiver object a and $\sigma'(\mathsf{x})$ is the value passed by the call, v. Once a frame evaluates to a value v, we discard the sub-frame and return to the outer frame, as shown in (rFrame2). We also note that the rule (rEvalCtx) dictates the evaluation order of an expression, based on the evaluation contexts $E[\cdot]$ defined in Figure 4.6.

Example 4.1.3.1 (Runtime Execution)**.** *Let h denote the heap depicted in Figure 4.1 and the current stack be $\sigma = (2, 9)$. Then, if we execute the expression* **this**.*push(7) with respect to σ and h we get the following reductions*[4] *where the rule names on the side indicate the main reduction rule applied to derive the reduction, not mentioning the use of context rules (*rEvalCtxt*) and (*rFrame1*).*

[4]In order to follow the Java code of Figure 4.2, the reductions use an object constructor that immediately initialises values to the parameters passed. This is more advanced than the simpler new construct considered in our language, which initialises all the fields of a fresh object to null. These details are however orthogonal to the determination of the owner of the object upon creation addressed here, derived from the type of the new object and the current active object.

$$
\begin{array}{lll}
\sigma \vdash \textsf{this}.\mathit{push(7)}, h & \leadsto\ \mathit{2.push(7)}\ ,\ h & (\textsc{rThis}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \textsf{this}.\mathit{top} := \textsf{new}\ \mathit{rep\ Node(x,}\ \textsf{this}.\mathit{top)}\ \ ,\ h & (\textsc{rCall}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \mathit{2.top} := \textsf{new}\ \mathit{rep\ Node(x,}\ \textsf{this}.\mathit{top)}\ \ ,\ h & (\textsc{rThis}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \mathit{2.top} := \textsf{new}\ \mathit{rep\ Node(7,}\ \textsf{this}.\mathit{top)}\ \ ,\ h & (\textsc{rVar}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \mathit{2.top} := \textsf{new}\ \mathit{rep\ Node(7,\ 2.top)}\ \ ,\ h & (\textsc{rThis}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \mathit{2.top} := \textsf{new}\ \mathit{rep\ Node(7,\ 3)}\ \ ,\ h & (\textsc{rField}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \mathit{2.top} := \mathit{13}\ \ ,\ h' & (\textsc{rNew}) \\
& \leadsto \mathit{frame}\ \sigma'\ \ \mathit{13}\ \ ,\ h'[(2, \mathit{top}) \mapsto \mathit{13}] & (\textsc{rAssign}) \\
& \leadsto\ \mathit{13}\ ,\ h'[(2, \mathit{top}) \mapsto \mathit{13}] & (\textsc{rFrame2})
\end{array}
$$

where $\sigma' = (2, 7)$, $h' = h \uplus \{\mathit{13} \mapsto (2, Node, \{\mathit{value} \mapsto 7,\ \mathit{next} \mapsto 3\})\}$ *and* $\mathit{13}$ *is a fresh address in the heap* h.

4.1.4 Topological Types

In this section we define the topological type system for UJ. The formalism is based on earlier work [127, 71] but has some differences: as we said in the introduction, we here focus on the hierarchical topology imposed by Universes but do not enforce the *owner-as-modifier* property at this stage[5] — this is dealt with later in Section 4.1.5. The main result of this section is Subject Reduction, stating that a type assigned to an expression and the ownership hierarchical heap structure is preserved during execution.

Subtyping and Type Composition/Decomposition

As was already stated in Section 4.1.1, types, t, are made up of two components: a universe u and a class name c. For every program we already assume a class inheritance reflexive transitive closure on class names, $\leq_c$. Using the ordering universe relation $\leq_u$ of Section 4.1.2 and $\leq_c$ we define the subtype relation as:

$$u\ c \leq u'\ c' \stackrel{\textbf{def}}{=} u \leq_u u' \text{ and } c \leq_c c' \tag{4.7}$$

We extend $(\oplus)$ and $(\ominus)$, defined earlier in Section 4.1.2 for universes, to types as $u \oplus t$ and $u \ominus t$ using the straightforward definitions

$$
\begin{array}{rcl}
u \oplus (u'\ c) & \stackrel{\textbf{def}}{=} & (u \oplus u')\ c \\
u \ominus (u'\ c) & \stackrel{\textbf{def}}{=} & (u \ominus u')\ c
\end{array}
$$

We use these two auxiliary operators whenever we need to change the viewpoint of the types. We can prove that our universe composition and decomposition operations on types respect the subtype relation (4.7).

Lemma 4.1.4.1 (Universe Composition and Decomposition preserves Subtyping)**.**

$$t' \leq t \implies \begin{cases} u \oplus t' \leq u \oplus t \\ u \ominus t' \leq u \ominus t \end{cases}$$

UJ Source Language Types

We typecheck UJ source expressions with respect to a typing environment Γ, which keeps typing information for this and the method parameter x. The Universe type derived from this judgement is interpreted with respect to the current this in Γ.

Definition 4.1.4.2 (Type Environment)**.** *A type environment* Γ *consists of a pair of types,* (t, t')*, assigning types to the current active object* this *and the parameter* x *respectively. We define the following operations on* Γ*:*

$$\Gamma(\mathit{this}) \stackrel{\mathit{def}}{=} \Gamma{\downarrow_1} \qquad\qquad \Gamma(x) \stackrel{\mathit{def}}{=} \Gamma{\downarrow_2}$$

[5] We therefore allow assignments and impure method calls on any objects in the heap.

$$\frac{}{\Gamma \vdash \mathsf{null} : t}(\textsc{Null}) \qquad \frac{}{\Gamma \vdash \mathsf{x} : \Gamma(\mathsf{x})}(\textsc{Var}) \qquad \frac{}{\Gamma \vdash \mathsf{this} : \Gamma(\mathsf{this})}(\textsc{This})$$

$$\frac{\Gamma \vdash e : t'}{\Gamma \vdash (t)\ e : t}(\textsc{Cast}) \qquad \frac{\Gamma \vdash e : t' \quad t' < t}{\Gamma \vdash e : t}(\textsc{Sub}) \qquad \frac{w \neq \mathsf{any}}{\Gamma \vdash \mathsf{new}\ w\ c : w\ c}(\textsc{New})$$

$$\frac{\Gamma \vdash e : u\ c \quad \mathcal{F}(c, f) = t}{\Gamma \vdash e.f : u \oplus t}(\textsc{Field}) \qquad \frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{F}(c, f) = u \ominus t}{\Gamma \vdash e.f := e' : t}(\textsc{Assign})$$

$$\frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{M}(c, m) = p : t_r\ (u \ominus t)}{\Gamma \vdash e.m(e') : u \oplus t_r}(\textsc{Call})$$

Figure 4.8: Source type system

The source expression type-judgement takes the form

$$\Gamma \vdash e : t$$

denoting that expression e has Universe type t with respect to the typing environment Γ. It is defined as the least relation satisfying the rules given in Figure 4.8. We sometimes find it convenient to use the shorthand judgement notation

$$\Gamma \vdash e : u \qquad \Gamma \vdash e : c$$

whenever components of the type judgement are not important, that is $\Gamma \vdash e : u\ \_$ and $\Gamma \vdash e : \_\ c$ respectively. Most of the rules are standard, with the exception of the type judgements (Field), (Assign) and (Call) which use the auxiliary operations $u \oplus t$ and $u \ominus t$ defined in Section 4.1.2 to *translate* types from one viewpoint to another.

Example 4.1.4.3 (Type Viewpoint Translation)**.** *If* $\Gamma \vdash$ *this.top* : *rep Node and field next in class Node has type peer Node, then using (*Field*), from the viewpoint* Γ*, the dereference this.top.next has type rep* $\oplus$ (*peer Node*) = *rep Node, that is*

$$\Gamma \vdash \textit{this.top.next} : \textit{rep Node}$$

*Conversely, we use (*Assign*) to check that when*

$$\Gamma \vdash \textit{this.top} : \textit{rep Node} \qquad \textit{and} \qquad \Gamma \vdash \textit{new rep Node} : \textit{rep Node}$$

then the assignment this.top.next := *new rep Node respects the field type assigned to next in class Node. For this calculation we use*

$$\mathcal{F}(\textit{Node}, \textit{next}) \ = \ \textit{peer}\ \mathsf{Node} \ = \ \textit{rep} \ \ominus (\textit{rep}\ \mathsf{Node})$$

The source expression type judgement allows us to formally define well-formed classes by requiring consistency between subclasses, that is field types of the class concord with the field types of any superclass of the class and method signatures are specialisations of the signatures of overridden methods, and that method bodies are consistent with the signature of that method.

The types in a method signature are meant to be interpreted with respect to this, the current active object. Thus when typing a method body, the observer is implicitly assumed to be this, even though there are no actual objects at compile-time. We assign the self (Extended) universe to Γ(this) when type-checking method bodies because we want the universes of the method calls and field accesses in the method bodies to be exactly the annotation w given in the class. We note that $\forall w.\ \mathsf{self} \oplus w = w$.[6]

[6]The inquisitive reader may wonder whether peer could have been used instead of self to type Γ(this). We note that peer would cause us to loose information when the universe w is rep, since peer $\oplus$ rep = any, thereby making the type system unnecessarily restrictive.

$$\frac{}{h, \sigma \vdash \mathsf{null} : t}(\textsc{tNull}) \qquad \frac{h, \sigma \vdash \sigma(\mathsf{x}) : t}{h, \sigma \vdash \mathsf{x} : t}(\textsc{tVar}) \qquad \frac{h, \sigma \vdash \sigma(\mathsf{this}) : t}{h, \sigma \vdash \mathsf{this} : t}(\textsc{tThis})$$

$$\frac{h, \sigma \vdash e : t'}{h, \sigma \vdash (t)\ e : t}(\textsc{tCast}) \qquad \frac{\begin{array}{l} h, \sigma \vdash e : t' \\ t' < t \end{array}}{h, \sigma \vdash e : t}(\textsc{tSub}) \qquad \frac{}{h, \sigma \vdash \mathsf{new}\ t : t}(\textsc{tNew})$$

$$\frac{\begin{array}{l} \mathbf{class}(h, a) = c \\ \sigma(\mathsf{this}), \mathbf{owner}(h, \sigma(\mathsf{this})) \vdash a, \mathbf{owner}(h, a) : u \end{array}}{h, \sigma \vdash a : u\ c}(\textsc{tAddr})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ \mathcal{F}(c, f) = t \end{array}}{h, \sigma \vdash e.f : u \oplus t}(\textsc{tField}) \qquad \frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ h, \sigma \vdash e' : t \\ \mathcal{F}(c, f) = u \ominus t \end{array}}{h, \sigma \vdash e.f := e' : t}(\textsc{tAssign})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ h, \sigma \vdash e' : t \\ \mathcal{M}(c, m) = p : t_r\ (u \ominus t) \end{array}}{h, \sigma \vdash e.m(e') : u \oplus t_r}(\textsc{tCall}) \qquad \frac{\begin{array}{l} h, \sigma' \vdash e : t \\ h, \sigma \vdash \sigma'(\mathsf{this}) : u \end{array}}{h, \sigma \vdash \mathsf{frame}\ \sigma'\ e : u \oplus t}(\textsc{tFrame})$$

Figure 4.9: Runtime type system

Definition 4.1.4.4 (Well-Formed Class)**.**

$$\frac{\begin{array}{l} \forall c' \geq_c c\ .\ \mathcal{F}(c', f) = t \Longrightarrow \mathcal{F}(c, f) = t \\ \forall c' \geq_c c\ .\ \mathcal{M}(c', m) = p : t'_r\ (t'_x) \Longrightarrow \left\{ \begin{array}{l} \mathcal{M}(c, m) = p : t_r\ (t_x) \\ \mathit{where}\ t_r \leq t'_r\ \mathit{and}\ t_x \geq t'_x \end{array} \right. \\ \forall \mathcal{M}(c, m) = p : t_r\ (t_x)\ .\ (\mathsf{self}\ c, t_x) \vdash \mathcal{MBody}(c, m) : t_r \end{array}}{\vdash c}(\textsc{WFClass})$$

The program is said to be *well-formed* if all the defined classes are well-formed.

Runtime Types

We define a type system for runtime expressions. These are type checked with respect to the base stack frame σ, which contains actual values for the current receiver this and the parameter x. Since runtime expressions also contain addresses, we also need to typecheck them with respect to the current heap, so as to retrieve the class membership and owner information for addresses.

The runtime Universe type system allows us to assign Universe types to runtime expressions with respect to a particular heap h and stack σ, through a judgement of the form

$$h, \sigma \vdash e : t$$

It is defined as the least relation satisfying the rules in Figure 4.9. Once again, we use the shorthand notation $h, \sigma \vdash e : u$ and $h, \sigma \vdash e : c$ whenever the other components of t in the judgement are not important. In the rule (tAddr), the type of an address in a heap is derived from the class of the object and the universe obtained using judgement (4.1) of Section 4.1.2. The three rules (tField), (tAssign) and (tCall) use the universe composition and decomposition operators in the same way as their static-expression counterparts in Figure 4.8. The new rule (tFrame) also uses the composition universe operation to translate the type of the sub-expression, obtained with respect to the sub-stack of the frame, to the current frame's viewpoint.

Lemma 4.1.4.5 states the composition and decomposition operations correctly characterise the translation of value that types from one viewpoint, denoted by σ in the runtime type system, to another. The viewpoint translations of Lemma 4.1.4.5 trivially apply to null values since we assign arbitrary types to such values using rule (tNull).

Lemma 4.1.4.5 (Determining the relative Universe Types of Values)**.**

(i) If $h, \sigma \vdash a : u\ \_$ *and* $h, (a, \_) \vdash v : t$ *then* $h, \sigma \vdash v : u \oplus t$

(ii) If $h, \sigma \vdash a : u\ \_$ *and* $h, \sigma \vdash v : t$ *then* $h, (a, \_) \vdash v : u \ominus t$

Example 4.1.4.6 (Relative Viewpoints in a Heap)**.** *In Figure 4.1, using* (4.2) *and* (4.3) *from Example 4.1.2.1 and rule (*TADDR*) we derive*

$$h, (2, \_) \vdash 3 : \mathsf{rep\ Node} \qquad \textit{and} \qquad h, (3, \_) \vdash 4 : \mathsf{peer\ Node}$$

From Lemma 4.1.4.5(i) we immediately derive

$$h, (2, \_) \vdash 4 : \mathsf{rep\ Node}$$

Conversely, using $h, (2, \_) \vdash 3 : \mathsf{rep\ Node}$, $h, (2, \_) \vdash 4 : \mathsf{rep\ Node}$ *and Lemma 4.1.4.5(ii) we can recover* $h, (3, \_) \vdash 4 : \mathsf{peer\ Node}$.

At this point, we have enough machinery to define well-formed addresses and heaps. An address is well-formed in a heap whenever its owner is valid (that is it is another address in the heap or root) and the type of its fields respect the type of the fields defined in $\mathcal{F}$. As described in Definition 4.1.4.8, a heap is well-formed, denoted as $\vdash h$, if transitive ownership, denoted by $\mathbf{owner}^*(h, o)$, is acyclic and all its addresses are well-formed.

Definition 4.1.4.7 (Transitive Ownership)**.**

$$\begin{array}{lcl} \boldsymbol{owner}^*(h, o) & \stackrel{def}{=} & \begin{cases} \{o\} \cup \boldsymbol{owner}^*(h, \boldsymbol{owner}(h, o)) & o \neq \mathsf{root} \\ \{\mathsf{root}\} & o = \mathsf{root} \end{cases} \\ \boldsymbol{owner}^+(h, o) & \stackrel{def}{=} & \boldsymbol{owner}^*(h, o) \backslash \{o\} \end{array}$$

Definition 4.1.4.8 (Well-Formed Addresses and Heaps)**.**

$$\frac{\begin{array}{l} \boldsymbol{owner}(h, a) \in (\boldsymbol{dom}(h) \cup \{\mathsf{root}\}) \\ \boldsymbol{class}(h, a) = c \\ \mathcal{F}(c, f) = t \implies h, (a, \_) \vdash h(a.f) : t \end{array}}{h \vdash a} \text{(WFADDR)}$$

$$\frac{\begin{array}{l} (a, b \in \boldsymbol{dom}(h) \ \wedge\ a \in \boldsymbol{owner}^*(h, b) \ \wedge\ b \in \boldsymbol{owner}^*(h, a)) \implies a = b \\ a \in \boldsymbol{dom}(h) \implies h \vdash a \end{array}}{\vdash h} \text{(WFHEAP)}$$

We conclude the section by showing the correspondence between the source-expression type system and runtime-expression type system. The Substitution Lemma 4.1.4.9 states that, with respect to a suitable stack σ, where $\sigma(\mathsf{this})$ and $\sigma(\mathsf{x})$ match the respective type assignments in Γ, a well-formed source expression is also a well-formed runtime expression. Despite the name used, we note that no "substitution" occurs in the expression itself, which is the same in both type judgements of the Lemma.

Lemma 4.1.4.9 (Substitution)**.**

$$\left.\begin{array}{l} \Gamma \vdash e : t \\ h, \sigma \vdash x : \Gamma(x) \\ h, \sigma \vdash \mathit{this} : \Gamma(\mathit{this}) \end{array}\right\} \implies h, \sigma \vdash e : t$$

Subject Reduction

The first main result of this section states that if a well-formed runtime expression e reduces with respect to a stack, σ, and a well-formed heap, h, then the resulting expression preserves its type, and the resulting heap preserves its well-formedness.

Theorem 4.1.4.10 (Subject Reduction)**.** *If a program is well-formed then*

$$\left.\begin{array}{l} \vdash h \\ h, \sigma \vdash e : t \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array}\right\} \Longrightarrow \begin{array}{l} \vdash h' \\ h', \sigma \vdash e' : t \end{array}$$

4.1.5 Encapsulation

In Section 4.1.4 we showed how the *topological* type system guarantees that the topology of the objects on the heap agrees with the one described by the Universe types. In this section we enhance the topological type system and obtain the *encapsulation* type system. We show that the latter system guarantees the owner-as-modifier property [127], which localises the effects of execution in a heap with respect to the current active object. We then show how this result can be used to deduce preservation of object invariants.

Notation: For the subsequent discussion we find it convenient to define the following notation and predicates. The expression $C[e]$ denotes that it contains sub-expression e in some expression context $C[\cdot]$. The notation $h(a.f_1 \dots f_n)$ is used as a shorthand when we want to refer to multiple dereferencing in a heap. The predicate $\mathbf{rFlds}(c, \mathsf{f}_1 \dots \mathsf{f}_n)$ holds if a list of field accesses yield an object within the representation of an object of class c, where the first field access f_1 is defined in class c (not inherited from superclasses of c); this is denoted as $\mathcal{DF}(c, f_1)$. The predicate $\mathbf{rFlds}(c, \mathsf{f}_1 \dots \mathsf{f}_n)$ thus holds if the first field f_1 is a rep field defined in c, and the remaining field accesses are either rep or peer. The predicate $\mathbf{pure}(c, m)$ holds if m is pure in c.

Modular Verification and Universe Types

Object invariants are often used in object-oriented program specification and verification. *Modularity* aims to allow subsets of the code (e.g. classes, modules, functions) to be checked in isolation, without consideration of the whole program. There have been various approaches to achieve modularity in verification [147, 94, 127].

For the case of object-oriented programs, [130] suggest:

Restrictions on Invariants: They can only be defined in terms of a restricted form of field accesses.

Restrictions on Effects: Methods can only affect the invariants of a restricted subset of objects during their execution.

These restrictions are expressed in terms of the hierarchic heap topology introduced through Universes.

Ownership Admissible Invariants [130] for an object of class c may only be defined in terms of fields that fall under the object's representation (i.e. itself and the objects it transitively owns). Such invariants also have the additional restriction that the first field access **this**.f needs to be defined in class c directly, that is $\mathcal{DF}(c, f)$; this allows for subclass separation [130] whereby we can verify the methods of class c with respect to the invariant of class c without having to reevaluate methods inherited from superclasses of c. More formally, an invariant in class c may only contain:

1. **this**.g where $\mathcal{DF}(c, \mathsf{g})$.
2. **this**.\ textsf {f}_1\ldots \ textsf {f}_n.\ textsf {g} where $\mathbf{rFlds}(c, \mathsf{f}_1 \dots \mathsf{f}_n)$ and g is a field defined in the class of type $\mathcal{F}(\dots \mathcal{F}(c, f_1), f_n)$.

With respect to the effect restrictions, an object x in a heap h is allowed to affect

1. invariants of objects its owner transitively owns, $\mathsf{inv}(y)$ where $\mathbf{owner}(h, x) \in \mathbf{owner}^*(h, y)$; note that this includes $\mathsf{inv}(x)$
2. invariants of objects which transitively own it, $\mathsf{inv}(y)$ where $y \in \mathbf{owner}^+(h, x)$.

In its current state, the Universe type system can only describe *direct ownership*. Hence the above effect restrictions translate to the *owner-as-modifier* property [71] whereby an object is only allowed to

1. directly assign to its own fields, the fields of rep objects (which it owns) and the fields of peer objects (which its owner owns).
2. call impure methods on itself, on objects it owns and on peer objects.

Example 4.1.5.1 (Universe, Invariants and Effects)**.**

```
1 class A {
2     rep A fra;
3     ...
4 } class B extends A {
5     rep B frb;   peer B fp;   any B fa;
6     ...
7 }
```

As an example, consider the class definitions A and B above. In terms of the restriction of invariants, the invariant of class B may mention the following fields

- **this**.*frb*, **this**.*fp* *and* **this**.*fa* *- its fields*
- **this**.*frb* ... *fr*.*fp* , **this**.*frb*.*fp* ... *fp*.*fr* *and* **this**.*frb*.*fp* ... *fr*.*fp*.*fa* *- fields of objects in its representation where the first field access* **this**.*frb* *is defined in B and not inherited from A.*

The invariant of class B may not mention **this**.*fp*.*fr* *and* **this**.*fr*.*fa*.*fr* *because they constitute fields of objects that do not belong to its representation. It may not mention* **this**.*fra* ... *fr*.*fp* *either because the first field access* *fra* *is defined in the superclass A.*

In terms of the restriction on effects, let us consider the heap depicted in Figure 4.10. If 5 is the active object (i.e. the current **this***), then it may affect:*

- *inv(5) - itself*
- *inv(6) and inv(8) - objects in its owner's representation*
- *inv(3), inv(2) and inv(1) - objects which transitively own it*

It however cannot affect the invariants of the objects 4 and 7.

Encapsulation Types

Encapsulation types impose extra restrictions so as to support the owner-as-modifier approach and guarantee the restriction on effects. We define an encapsulation judgement for expressions, $\Gamma \vdash_{\mathbf{enc}} e$, reflecting the expression restrictions imposed in [130] to control the effects of method executions on the invariants of other objects, as discussed above. These restrictions state that for an expression e to respect encapsulation, it can only assign to and call impure methods on itself, on rep receivers or peer receivers. Recall that z is a Universe metavariable that ranges over self, rep and peer only. To separate between pure and impure methods we require a purity judgement for expressions $\Gamma \vdash_{\mathbf{pure}} e$. An expression e is pure if it never assigns to fields and *only* calls pure methods.

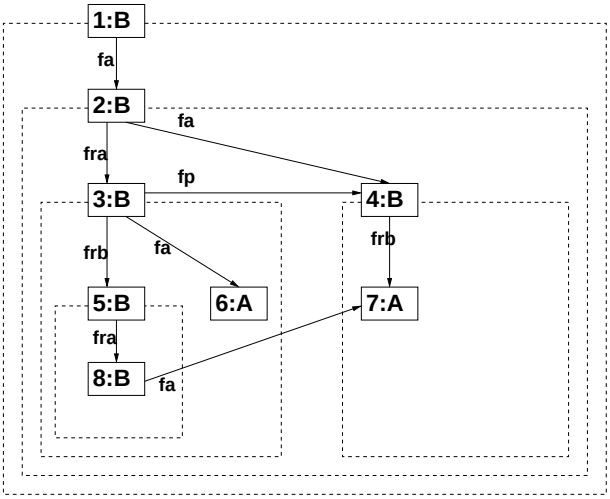

Figure 4.10: A heap

Definition 4.1.5.2 (Encapsulation and Purity for Source Expressions)**.**

$$\frac{\begin{array}{l}\Gamma \vdash e : t \\ e = C[e_1.f := e_2] \Longrightarrow \Gamma \vdash e_1 : z \\ e = C[e_1.m(e_2)] \Longrightarrow \Gamma \vdash e_1 : z \ \vee \ (\textbf{\textit{pure}}(c,\, m) \ \textit{where}\ \Gamma \vdash e_1 : c)\end{array}}{\Gamma \vdash_{\textbf{\textit{enc}}} e} \ (\textsc{Enc})$$

$$\frac{\begin{array}{l}\Gamma \vdash e : t \\ e \neq C[e_1.f := e_2] \\ e = C[e_1.m(e_2)] \Longrightarrow (\textbf{\textit{pure}}(c,\, m) \ \textit{where}\ \Gamma \vdash e_1 : c)\end{array}}{\Gamma \vdash_{\textbf{\textit{pure}}} e} \ (\textsc{Pure})$$

A class is well-formed with respect to encapsulation, denoted as $\vdash_{\textbf{enc}} c$, if and only if all pure methods have bodies that neither assign to fields nor call impure methods and impure methods have bodies that only assign to fields and call impure methods on rep or peer receivers. We recall that according to Definition 4.1.4.4, pure methods are only overridden by pure methods, and similarly for impure methods. A Program P is well-formed with respect to encapsulation if all its classes are encapsulated.

Definition 4.1.5.3 (Encapsulated Well-Formed Class)**.**

$$\frac{\begin{array}{l}\vdash c \\ \forall m. \quad \begin{array}{rcl}\textbf{\textit{pure}}(c,\, m) & \Longrightarrow & (\mathsf{self}\ c, \_) \vdash_{\textbf{\textit{pure}}} \mathcal{MBody}(c, m) \\ \neg\textbf{\textit{pure}}(c,\, m) & \Longrightarrow & (\mathsf{self}\ c, \_) \vdash_{\textbf{\textit{enc}}} \mathcal{MBody}(c, m)\end{array}\end{array}}{\vdash_{\textbf{\textit{enc}}} c} \ (\text{WFEncClass})$$

$$\vdash_{\textbf{\textit{enc}}} P \implies \forall c \in Id_c \vdash_{\textbf{\textit{enc}}} c$$

We also define encapsulation and purity judgements for *runtime expressions* subject to a heap h and a stack σ; these judgements are denoted as $h, \sigma \vdash_{\textbf{enc}} e$ and $h, \sigma \vdash_{\textbf{pure}} e$ respectively. Encapsulation and purity for runtime expressions impose similar requirements to those for source expressions but add an extra clause for frame expressions. In particular, encapsulation for frames, $h, \sigma \vdash_{\textbf{enc}} \mathsf{frame}\ \sigma'\ e'$, requires that the receiver in σ', that is $\sigma'(\mathsf{this})$, is a rep, peer or self of that in σ. This condition is expressed through the predicate $h \vdash \sigma' \preceq_{\textbf{enc}} \sigma$, defined below.

Definition 4.1.5.4 (Frame Encapsulation)**.**

$$h \vdash \sigma' \preceq_{\textbf{\textit{enc}}} \sigma \quad \stackrel{\textbf{\textit{def}}}{=} \quad h, \sigma \vdash \sigma'(\mathsf{this}) : z$$

Definition 4.1.5.5 (Purity and Encapsulation for Runtime Expressions)**.**

$$\frac{\begin{array}{l} h, \sigma \vdash e : t \\ e = C[\mathsf{frame}\ \sigma_1\ e_1] \Longrightarrow h, \sigma_1 \vdash_{\boldsymbol{pure}} e_1 \\ e \neq C[e_1.f := e_2] \\ e = C[e_1.m(e_2)] \Longrightarrow (\boldsymbol{pure}(c,\, m)\ \textit{where}\ h, \sigma \vdash e_1 : c) \end{array}}{h, \sigma \vdash_{\boldsymbol{pure}} e} \quad (\textsc{rPure})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : t \\ e = C[\mathsf{frame}\ \sigma_1\ e_1] \Longrightarrow (h \vdash \sigma_1 \preceq_{\boldsymbol{enc}} \sigma\ \wedge\ h, \sigma_1 \vdash_{\boldsymbol{enc}} e_1)\ \vee\ (h, \sigma_1 \vdash_{\boldsymbol{pure}} e_1) \\ e = C[e_1.f := e_2] \Longrightarrow h, \sigma \vdash e_1 : z \\ e = C[e_1.m(e_2)] \Longrightarrow h, \sigma \vdash e_1 : z\ \vee\ (\boldsymbol{pure}(c,\, m)\ \textit{where}\ h, \sigma \vdash e_1 : c) \end{array}}{h, \sigma \vdash_{\boldsymbol{enc}} e} \quad (\textsc{rEnc})$$

Our proof for encapsulation starts by showing a correspondence between source expression encapsulation and purity and runtime expression encapsulation and purity (with respect to a suitable stack) and that purity implies encapsulation. We also need to prove two core (but fairly standard) intermediary results. Subject reduction for purity and encapsulation states that if a runtime expression respects purity (or encapsulation) the resulting expression after reduction will still respect purity (or encapsulation). Safety for purity and encapsulationstates that a pure expression never changes the state of existing objects and that an expression that respects encapsulation only changes objects that are transitively owned by the *owner of* the current active object.

We can now prove the Encapsulation Theorem, the main result of this section. It states that if an expression respects encapsulation (with respect to some h, σ), then during its execution it will only update objects that form part of the representation of the owner of the current active object.

Theorem 4.1.5.6 (Encapsulation)**.**

$$\left.\begin{array}{l} h, \sigma \vdash_{\boldsymbol{enc}} e \\ \sigma \vdash e, h \rightsquigarrow^* e', h' \\ a \in \boldsymbol{dom}(h) \\ \boldsymbol{owner}(h, \sigma(\mathsf{this})) \notin \boldsymbol{owner}^*(h, a) \end{array}\right\} \Longrightarrow\ h(a.f) = h'(a.f)$$

Modular Invariant Preservation

We conclude by relating the Encapsulation Theorem 4.1.5.6 to modular verification. We (abstractly) define *legal invariants* as those satisfying the constraints outlined earlier in Section 4.1.5.

Definition 4.1.5.7 (Predicate Satisfaction)**.** *We assume a mapping from pairs of addresses and classes to predicates*

$$\mathsf{inv} \in \mathit{INV} : \mathit{Addr} \times \mathit{Id}_c \rightharpoonup \mathit{Pred}$$

We also assume a predicate satisfaction relation

$$\models \subseteq \mathit{Heap} \times \mathit{Pred}$$

In particular, $h \models \mathsf{inv}(a, c)$ *represents the satisfaction of the invariant of the object at address* a *with class* c *in heap* h*.*

Definition 4.1.5.8 (Legal Invariants)**.** *An invariant* $\mathsf{inv}(a, c)$ *is legal if it satisfies the property:*

$$\forall h, h' \left.\begin{array}{l} h \models \mathsf{inv}(a, c) \\ h(a.f) = h'(a.f) \\ h(a.f_1 \ldots f_n.g) = h'(a.f_1 \ldots f_n.g) \\ \qquad \textit{where}\ \boldsymbol{rFlds}(c,\, f_1 \ldots f_n) \end{array}\right\} \Longrightarrow h' \models \mathsf{inv}(a, c)$$

We finally can prove the Modular Invariant Preservation Theorem stating that execution preserves invariants of unrelated objects, that is objects that neither belong to the representation of the active object's owner nor transitively own the active object.

Theorem 4.1.5.9 (Modular Invariant Preservation)**.**

$$\left.\begin{array}{l} h \models \mathit{inv}(a,c) \\ \mathit{inv}(a,c)\ \mathit{legal} \\ h,\sigma \vdash_{\mathbf{enc}} e \\ a \notin \mathbf{owner}^*(h,\sigma(\mathsf{this})) \\ \mathbf{owner}(h,\sigma(\mathsf{this})) \notin \mathbf{owner}^*(h,a) \\ \sigma \vdash e, h \rightsquigarrow^* e', h' \end{array}\right\} \implies h' \models \mathit{inv}(a,c)$$

Proof. Use Theorem 4.1.5.6 and the conditions defining $h \models \mathsf{inv}(a)$ in Definition 4.1.5.8. □

Example 4.1.5.10 (Modular Invariant Preservation)**.** *Recall the heap depicted in the beginning in Figure 4.1. If we take σ such that the active object $\sigma(\mathsf{this}) = 2$, then we can show that $h,\sigma \vdash_{\mathbf{enc}} 2.\mathsf{push}(7)$. We can also show that all the classes in Figure 4.2 are well formed with respect to encapsulation. Thus if we consider the case where $a = 10$, Theorem 4.1.5.9 guarantees that if $h \models \mathit{inv}(10, \mathit{Bag})$ and we execute expression $\sigma \vdash 2.\mathsf{push}(7), h \rightsquigarrow^* e', h'$, then $h' \models \mathit{inv}(10, \mathit{Bag})$ for the resultant heap h'. The same argument can be applied for $a \in \{6,7,8,9,11,12\}$.*

However, we cannot derive the same conclusion for $a \in \{3,4,5\}$ because these object are in the representation of the active object 2. We also cannot apply Theorem 4.1.5.9 for $a = 1$ since $\mathit{inv}(1, \mathit{Bag})$ may mention the fields **this**.*state*, **this**.*state*.*next* *and* **this**.*state*.*next*.*next* *which are objects whose states may have been affected by the execution of expression 2.*$\mathsf{push}(7)$.

4.1.6 Conclusion

We given an alternative formalisation of the Universe type system in two steps, by first presenting a topological type system which preserves the ownership topology and then augmenting it to the encapsulation type system, which can lead to modular reasoning about programs. This two-step formalisation primarily allows for separation of concerns when extending this work; some extensions and applications of Universes do not require encapsulation properties such as owner-as-modifier, introduced in the latter step. Also, the two-step formalisation permits a gentler presentation of the mathematical machinery we develop. Both these factors facilitate the adoption of the work as a starting point for further work.

Our formalisation also presented some novel techniques. We introduced a distinction between the original Universes as defined in [127] and an extended version which includes the specialisation of the peer universe called self; extended universes lead to more succinct definitions such as that for well-formed classes. We also introduced the notion of universe composition and decomposition, which was then used extensively for type manipulation in the typing rules. Both these aspects were already present in earlier work on Universes; we merely made them more explicit thereby facilitating their understanding. We have proven subject reduction (for both topological and encapsulation type systems) for a small-step semantics of a strict subset of Java. We have also shown how our formalisation can be used for modular verification of object invariants.

The formalisation of the Universe type system is used as a basis for various extensions. Section 4.2 extends UJ to avoid races, and 4.3 extends UJ to handle Generics in Java [97]. We plan to adapt the type system to the bytecode translation of the Java code directly. This will permit the use of Universes for the verification of mobile bytecode in a Proof-Carrying-Code architecture such as the one proposed by MOBIUS [124].

4.2 Universe Types for Race-free programs

A *race condition* is an error that can occur in concurrent programs, if two threads are not properly synchronised, and they simultaneously access the same object. This can then lead to corruption of data structures,

```
1  class Student { int mark ; bool clean_room  }

2  class Dept {  // Closed list
3      rep DeptStudentNode first;
4      void releaseMarks () { ... }
5  }
6  class DeptStudentNode {
7      peer Student s;
8      peer DeptStudentNode next;
9  }

10 class Hall {  // Open list
11     rep HallStudentNode first;
12     void cleanRooms () { ... }
13 }
14 class HallStudentNode {
15     any Student s;
16     peer HallStudentNode next;
17 }
```

Figure 4.11: Example program showing heap hierarchy structure

and eventual software failure. To date, many well-known pieces of software have fallen foul of race conditions, often long after their initial development, sometimes leading to denial-of-service attacks or other security problems[7].

Therefore, we developed a type system for race safety using universes to partition the heap. As in earlier work by Boyapati, Flanaggan and others[50, 78], we treat objects in the same *ownership domain* (i.e. all objects sharing the same owner) as guarded by the same lock. At run-time we associate this lock with the objects' owner. All objects have an implicit reference to their owner.

The use of anyallows the expression of data structures containing objects from various ownership domains. Use of such data structures does not require us to compromise the design of other data structures in our system. In the case where the type does not indicate the owner of an object, we use paths as an alternative mechanism to guarantee correct synchronisation. We use an effect system that ensures correctness even if these paths are not final as would be required in [50, 78].

We summarise the advantages of our system below:

1. The annotations are succinct.
2. Sync blocks need not be split when data structures share elements.
3. Paths do not have to be final.
4. Single objects can be locked.

The full paper has appeared in [60], and here we explain the use of the system through an example:

We use the tree-hierarchy imposed by the Universe types to avoid races as follows: We require that the run-time system records the owner of an object. We associate a lock with each object, with objects guarded by their owner's lock rather than their own. Any accesses to a field of an object, *e.g.,* $e'.f$ or $e'.f := ...$, must be within a `sync` e block where e resolves to an object that is part of the same ownership domain as e'. In other words, `sync` e locks the object owning e. This is statically verifiable when the universe qualifier of e' is not `any`.

Consider the code of `releaseMarks` from Fig. 4.12. Adhering to the rule set out above, the field access to `this.first` (line 20) is enclosed within `sync(this)` (line 19). More interesting is the body of the `while` loop, where a statically unknown number of field accesses through `i.next` (line 24) is correctly synchronised by acquiring a *single* lock, `sync(i)`, before the loop (line 21). Even though `i` will point to different objects at each iteration, the synchronisation is correct, because the field `next` is peerand thus all these objects will have the same owner. The same is true when we access the student (line 23). We call the list inside `Dept` a *closed* list as the students are enclosed in the ownership domain of the list. In contrast the list inside `Hall` is *open* because its students can be anywhere.

A challenge we needed to tackle is, how to avoid races when we do not know which ownership domain the accessed object will belong to, *i.e.,* when it has type `anyC` for some class `C`. In such a case, any accesses

[7] Querying the SecurityFocus website [**?**] for "race condition" reveals hundreds of problems.

```
18  void releaseMarks () {
19      sync (this) {  // so we can access fields of this
20          rep DeptStudentNode i = this.first;
21           sync (i) {  // so we can access the nodes
22               while (i) {
23                  i.s.mark = ...;
24                  i = i.next;
25  }   }   }   }
26   void cleanRooms () {
27      sync (this) {  // so we can access fields of this
28          rep HallStudentNode i = this.first;
29          sync (i) {  // so we can access the nodes
30              while (i) {
31                  sync (i.s) { i.s.room_clean = true; }
32                  i = i.next;
33  }   }  }   }
```

Figure 4.12: Method bodies for Fig. 4.11

of the form $p.f$ or $p.f = ...$, where p is a path[8] must be within a `sync` p block provided that the block does not assign to any of the fields appearing in p.

Consider the body of `cleanRooms` in figure 4.12. The difference between this and `releaseMarks` is that the latter uses `HallStudentNode` which has an `any`pointer to `Student`. Thus the student is not necessarily a peer of the node and when we access `i.s` (line 31) the `sync(i)` (line 29) is no longer sufficient. We must lock the owner of the student `i.s` and this is possible through the "fresh" `sync(i.s)` (line 31) even though `i.s` is `any`. We must be sure however that the body of the `sync` block does not write to the field `s`, otherwise the type system would reject our program.

Note that through `releaseMarks`, students will receive their marks *atomically*, i.e. there is never a state visible where a subset of students have their marks, but this is not the case for the cleaning of rooms. A student may notice their room has been cleaned whereas another student's room has not.

The above example demonstrates the power given by the modifier `any`. We can design a *closed list* which contains only objects of a specific ownership domain, and an *open list* which contains objects of any ownership domain. We want the open and closed lists to contain some objects in common. In [78], an open list of students can be written if we design the student so that it has a final field that stores the owner. In other words, we create a special class that can be referenced by a variable whose type does not specify an owner. However, this change is global to the program so every other reference must use the same type that does not specify an owner. This means we cannot make a closed list of students, because the owner of the student is no-longer indicated by its type. The only solution is to use open lists everywhere, which have the undesirable property that we cannot lock all the elements of the list at once, we have to acquire the same lock m times where m is the number of students in the list. Another implication is that iterating through the list *cannot be atomic* (as in `releaseMarks`).

The type system to avoid races is an extension of that of UJ, and is given in [60].

4.3 Generic Universe Types

Although ownership type systems have covered all features of Java-like languages (including for example exceptions, inner classes, and static class members) there are only three proposals of ownership type systems that support generic types. SafeJava [49] supports type parameters and ownership parameters independently, but does not integrate both forms of parametricity. This leads to significant annotation overhead. Ownership Domains [12] combine type parameters and domain parameters into a single parameter space and thereby

[8] A path is a sequence of field accesses starting from a parameter or `this`.

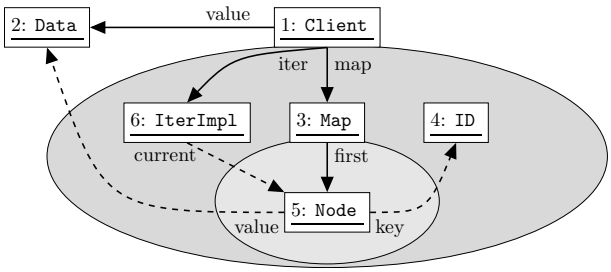

Figure 4.13: Object structure of a map from `ID` to `Data` objects. The map is represented by `Node` objects. The iterator has a direct reference to a node. Objects, references, and contexts are depicted by rectangles, arrows, and ellipses, respectively. Owner objects sit atop the context of objects they own. Arrows are labelled with the name of the variable that stores the reference. Dashed arrows depict references that cross context boundaries without going through the owner. Such references must not be used to modify the state of the referenced objects.

reduce the annotation overhead. However, their formalization does not cover type parameters. Ownership Generic Java (OGJ) [142] allows programmers to attach ownership information through type parameters. For instance, a collection of `Book` objects can be typed as "my collection of library books", expressing that the collection object belongs to the current `this` object, whereas the `Book` objects in the collection belong to an object "library". OGJ enforces the owner-as-dominator discipline. It piggybacks ownership information on type parameters. In particular, each class `C` has a type parameter to encode the owner of a `C` object. This encoding allows OGJ to use a slight adaptation of the normal Java type rules to also check ownership, which makes the formalization very elegant.

However, OGJ cannot be easily adapted to enforce the owner-as-modifier discipline. For example, OGJ would forbid a reference from the iterator (object 6) in Fig. 4.13 to a node (object 5) of the map (object 3), because the reference bypasses the node's owner. However, such references are necessary, and are legal in the owner-as-modifier discipline. A type system can permit such references in two ways.

First, if the iterator contained a field `theMap` that references the associated map object, then path-dependent types [12, 49] can express that the `current` field of the iterator points to a `Node` object that is owned by `theMap`. Unfortunately, path-dependent types require the fields on the path (here, `theMap`) to be final, which is too restrictive for many applications.

Second, one can loosen up the static ownership information by allowing certain references to point to objects in any context [71]. Subtyping allows values with specific ownership information to be assigned to "any" variables, and downcasts with runtime checks can be used to recover specific ownership information from such variables. In OGJ, this subtype relation between any-types and other types would require covariant subtyping, for instance, that `Node<This>` is a subtype of `Node<Any>`, which is not supported in Java (or C#). Therefore, piggybacking ownership on the standard Java type system is not possible in the presence of "any".

We developed Generic Universe Types (GUT), an ownership type system for a programming language with generic types similar to Java 5 and C# 2.0. GUT enforces the owner-as-modifier discipline using the `any`modifier. Our type system supports type parameters for classes and methods. The annotation overhead for programmers is as low as in OGJ, although the presence of `any` makes the type rules more involved. A particularly interesting aspect of our work is how generics and ownership can be combined in the presence of an `any` modifier, in particular, how a restricted form of ownership covariance can be permitted without runtime checks.

Outline. Sec. 4.3.1 illustrates the main concepts of GUT by an example. Secs. 4.3.2 and 4.3.3 present the type rules and the runtime model of GUT, respectively. Sec. 4.3.4 presents the type safety and the owner-as-modifier property theorems. Details and proofs can be found in the accompanying technical report [68].

4.3.1 Main Concepts

In this section, we explain the main concepts of GUT informally by an example. Class `Map` (Fig. 4.14) implements a generic map from keys to values. Key-value pairs are stored in singly-linked `Node` objects. Class `Node` extends the superclass `MapNode` (both Fig. 4.15), which is used by the iterator (classes `Iter` and `IterImpl` in Fig. 4.16). The `main` method of class `Client` (Fig. 4.17) builds up the map structure shown in Fig. 4.13. For simplicity, we omit access modifiers from all examples.

Ownership Modifiers. A type in GUT is either a type variable or consists of an ownership modifier, a class name, and possibly type arguments. The *ownership modifier* expresses object ownership relative to the current receiver object `this`[9]. Programs may contain the ownership modifiers `peer`, `rep`, and `any`. `peer` expresses that an object has the same owner as the `this` object, `rep` expresses that an object is owned by `this`, and `any` expresses that an object may have any owner. `any` types are supertypes of the `rep` and `peer` types with the same class and type arguments because they convey less specific ownership information.

The use of ownership modifiers is illustrated by class `Map` (Fig. 4.14). A `Map` object owns its `Node` objects since they form the internal representation of the map and should, therefore, be protected from unwanted modifications. This ownership relation is expressed by the `rep` modifier of `Map`'s field `first`, which points to the first node of the map.

```
1  class Map<K, V> {
2      rep Node<K, V> first;
3
4      void put(K key, V value) {
5          rep Node<K, V> newfirst = new rep Node<K, V>();
6          newfirst.init(key, value, first);
7          first = newfirst;
8      }
9
10     pure V get(K key) {
11         peer Iter<K, V> i = iterator();
12         while (i.hasNext()) {
13             if (i.getKey().equals(key)) return i.getValue();
14             i.next();
15         }
16         return null;
17     }
18
19     pure peer Iter<K, V> iterator() {
20         peer IterImpl<K, V, rep Node<K, V> > res;
21         res = new peer IterImpl<K, V, rep Node<K, V> >();
22         res.setCurrent( first );
23         return res;
24     }
25
26     pure peer IterImpl<K, V, rep Node<K, V> > altIterator() {
27         /* same implementation as method iterator() above */
28     }
29 }
```

Figure 4.14: An implementation of a generic map. `Map` objects own their `Node` objects, as indicated by the `rep` modifier in all occurrences of class `Node`. Method `altIterator` is for illustration purposes only.

The owner-as-modifier discipline is enforced by disallowing modifications of objects through `any` references. That is, an expression of an `any` type may be used as receiver of field reads and calls to side-effect free

[9] We ignore static methods in this section, but an extension is possible [127].

(*pure*) methods, but not of field updates or calls to non-pure methods. To check this property, we require side-effect free methods to be annotated with the keyword `pure`.

Viewpoint Adaptation. As discussed in section 4.1, ownership modifiers express ownership relative to `this`, and have to be adapted when this "viewpoint" changes. Consider `Node`'s inherited method `init` (Fig. 4.15). After substituting the type variable `X`, the third parameter has type `peer Node<K,V>`. The `peer` modifier expresses that the parameter object must have the same owner as the receiver of the method. On the other hand, `Map`'s method `put` calls `init` on a `rep Node` receiver, that is, an object that is owned by `this`. Therefore, the third parameter of the call to `init` also has to be owned by `this`. This means that from this particular call's viewpoint, the third parameter needs a `rep` modifier, although it is declared with a `peer` modifier. In the type system, this *viewpoint adaptation* is done by combining the type of the receiver of a call (here, `rep Node<K,V>`) with the type of the formal parameter (here, `peer Node<K,V>`). This combination yields the argument type from the caller's point of view (here, `rep Node<K,V>`).

```
1  class MapNode<K, V, X extends peer MapNode<K, V, X> > {
2      K key; V value; X next;
3
4      void init (K k, V v, X n) { key = k; value = v; next = n; }
5  }
6
7  class Node<K, V> extends MapNode<K, V, peer Node<K, V> > {}
```

Figure 4.15: Nodes form the internal representation of maps. Class `MapNode` implements nodes for singly-linked lists. Using a type variable for the type of `next` is useful to implement iterators. The subclass `Node` instantiates `MapNode`'s type parameter `X` to implement a list of nodes with the same owner.

Viewpoint adaptation and the owner-as-modifier discipline provide encapsulation of internal representation objects. Assume that class `Map` by mistake leaked a reference to an internal node, for instance, by making `first` public or by providing a method that returns the node. By viewpoint adaptation of the node type, `rep Node<K,V>`, clients of the map can only obtain an `any` reference to the node and, thus, the owner-as-modifier discipline guarantees that clients cannot directly modify the node structure. This allows the map to maintain invariants over the node, for instance, that the node structure is acyclic.

Type Parameters. Ownership modifiers are also used in actual type arguments. For instance, `Map`'s method `iterator` instantiates `IterImpl` with the type arguments `K`, `V`, and `rep Node<K,V>`. Thus, local variable `res` has type `peer IterImpl<K,V,rep Node<K,V>>`, which has two ownership modifiers. The *main modifier* `peer` expresses that the `IterImpl` object has the same owner as `this`, whereas the *argument modifier* `rep` expresses that the `Node` objects used by the iterator are owned by `this`. It is important to understand that this argument modifier again expresses ownership relative to the current `this` object (here, the `Map` object), and not relative to the instance of the generic class that contains the argument modifier (here, the `IterImpl` object `res`).

Type variables have upper bounds, which default to `any Object`. In a class `C`, the ownership modifiers of an upper bound express ownership relative to the `C` instance `this`. However, when `C`'s type variables are instantiated, the modifiers of the actual type arguments are relative to the receiver of the method that contains the instantiation. Therefore, checking the conformance of a type argument to its upper bound requires a viewpoint adaptation. For instance, to check the instantiation `peer IterImpl<K,V,rep Node<K,V>>` in class `Map`, we adapt the upper bound of `IterImpl`'s type variable `X` (`any MapNode<K,V,X>`) from viewpoint `peer IterImpl<K,V,rep Node<K,V>>` to the viewpoint `this`. With the appropriate substitutions, this adaptation yields `any MapNode<K,V,rep Node<K,V>>`. The actual type argument `rep Node<K,V>` is a subtype of the adapted upper bound. Therefore, the instantiation is correct. The `rep` modifier in the type argument and the adapted upper bound reflects correctly that the `current` node of this particular iterator is owned by `this`.

Type variables are not subject to the viewpoint adaptation that is performed for non-variable types.

```
interface Iter<K, V> {
    pure K getKey();
    pure V getValue();
    pure boolean hasNext();
    void next();
}

class IterImpl<K, V, X extends any MapNode<K, V, X>> implements Iter<K, V> {
    X current;

    void setCurrent(X c) { current = c; }
    pure K getKey() { return current.key; }
    pure V getValue() { return current.value; }
    pure boolean hasNext() { return current != null; }
    void next() { current = current.next; }
}
```

Figure 4.16: Class `IterImpl` implements iterators over `MapNode` structures. The precise node type is passed as type parameter. The upper bound allows methods to access a node's fields. Interface `Iter` hides `IterImpl`'s third type parameter from clients.

```
class ID { /* ... */ }
class Data { /* ... */ }

class Client {
    void main(any Data value) {
        rep Map<rep ID, any Data> map = new rep Map<rep ID, any Data>();
        map.put(new rep ID(), value);

        rep Iter<rep ID, any Data> iter = map.iterator();
        rep ID id = iter.getKey();
    }
}
```

Figure 4.17: Main program for our example. The execution of method `main` creates the object structure in Fig. 4.13.

When type variables are used, for instance, in field declarations, the ownership information they carry stays implicit and does, therefore, not have to be adapted. The substitution of type variables by their actual type arguments happens in the scope in which the type variables were instantiated. Therefore, the viewpoint is the same as for the instantiation, and no viewpoint adaptation is required. For instance, the call expression `iter.getKey()` in method `main` (Fig. 4.17) has type `rep ID`, because the result type of `getKey()` is the type variable `K`, which gets substituted by the first type argument of `iter`'s type, `rep ID`.

Thus, even though an `IterImpl` object does not know the owner of the keys and values (due to the implicit `any` upper bound for `K` and `V`), clients of the iterator can recover the exact ownership information from the type arguments. This illustrates that Generic Universe Types provide strong static guarantees similar to those of owner-parametric systems [58], even in the presence of `any` types. The corresponding implementation in non-generic Universe types requires a downcast from the `any` type to a `rep` type and the corresponding runtime check [71].

Limited Covariance and Viewpoint Adaptation of Type Arguments. Subtyping with covariant type arguments is in general not statically type safe. For instance, if `List<String>` were a subtype of `List<Object>`, then clients that view a string list through type `List<Object>` could store `Object` instances in the string list, which breaks type safety. The same problem occurs for the ownership information encoded in types. If `peer IterImpl<K,V,rep Node<K,V>>` were a subtype of `peer IterImpl<K,V,any`

`Node<K,V>>`, then clients that view the iterator through the latter type could use method `setCurrent` (Fig. 4.16) to set the iterator to a `Node` object with an arbitrary owner, even though the iterator requires a specific owner. The covariance problem can be prevented by disallowing covariant type arguments (like in Java and C#), by runtime checks, or by elaborate syntactic support [75].

However, the owner-as-modifier discipline supports a limited form of covariance without any additional checks. Covariance is permitted if the main modifier of the supertype is `any`. For example, `peer IterImpl<K,V,rep Node<K,V>>` is an admissible subtype of `any IterImpl<K,V,any Node<K,V>>`. This is safe because the owner-as-modifier discipline prevents mutations of objects referenced through `any` references. In particular, it is not possible to set the iterator to an `any Node` object, which prevents the unsoundness illustrated above.

Besides subtyping, GUT provides another way to view objects through different types, namely viewpoint adaptation. If the adaptation of a type argument yields an `any` type, the same unsoundness as through covariance could occur. Therefore, when a viewpoint adaptation changes an ownership modifier of a type argument to `any`, it also changes the main modifier to `any`.

This behavior is illustrated by method `main` of class `Client` in Fig. 4.17. Assume that `main` calls `altIterator()` instead of `iterator()`. As illustrated by Fig. 4.13, the most precise type for the call expression `map.altIterator()` would be `rep IterImpl<rep ID, any Data, any Node<rep ID, any Data>>` because the `IterImpl` object is owned by the `Client` object `this` (hence, the main modifier `rep`), but the nodes referenced by the iterator are neither owned by `this` nor peers of `this` (hence, `any Node`). However, this viewpoint adaptation would change an argument modifier of `altIterator`'s result type from `rep` to `any`. This would allow method `main` to use method `setCurrent` to set the iterator to an `any Node` object and is, thus, not type safe. The correct viewpoint adaptation yields `any IterImpl<rep ID, any Data, any Node<rep ID, any Data>>`. This type is safe, because it prevents the `main` method from mutating the iterator, in particular, from calling the non-pure method `setCurrent`.

Since `next` is also non-pure, `main` must not call `iter.next()` either, which renders `IterImpl` objects useless outside the associated `Map` object. To solve this issue, we provide interface `Iter`, which does not expose the type of internal nodes to clients. The call `map.iterator()` has type `rep Iter<rep ID, any Data>`, which does allow `main` to call `iter.next()`. Nevertheless, the type variable `X` for the type of `current` in class `IterImpl` is useful to improve static type safety. Since the current node is neither a `rep` nor a `peer` of the iterator, the only alternative to a type variable is an `any` type. However, an `any` type would not capture the relationship between an iterator and the associated list. In particular, it would allow clients to use `setCurrent` to set the iterator to a node of an arbitrary map. For a discussion of alternative designs see [68].

4.3.2 Static Checking

In this section, we formalize the compile time aspects of GUT. We define the syntax of the programming language, formalize viewpoint adaptation, define subtyping and well-formedness conditions, and present the type rules.

Programming Language

We formalize Generic Universe Types for a sequential subset of Java 5 and C# 2.0 including classes and inheritance, instance fields, dynamically-bound methods, and the usual operations on objects (allocation, field read, field update, casts). For simplicity, we omit several features of Java and C# such as interfaces, exceptions, constructors, static fields and methods, inner classes, primitive types and the corresponding expressions, and all statements for control flow. We do not expect that any of these features is difficult to handle (see for instance [49, 70, 127]). The language we use is similar to Featherweight Generic Java [97]. We added field updates because the treatment of side effects is essential for ownership type systems and especially the owner-as-modifier discipline.

Fig. 4.18 summarizes the syntax of our language and our naming conventions for variables. We assume that all identifiers of a program are globally unique except for `this` as well as method and parameter names

of overridden methods. This can be achieved easily by preceding each identifier with the class or method name of its declaration (but we omit this prefix in our examples).

The superscript $^{\mathtt{s}}$ distinguishes the sorts for static checking from corresponding sorts used to describe the runtime behavior, but is omitted whenever the context determines whether we refer to static or dynamic entities.

$\overline{\mathtt{T}}$ denotes a sequence of Ts. In such a sequence, we denote the i-th element by $\mathtt{T}_i$. We sometimes use sequences of tuples $S = \overline{\mathtt{X}\ \mathtt{T}}$ as maps and use a function-like notation to access an element $S(\mathtt{X}_i) = \mathtt{T}_i$. A sequence $\overline{\mathtt{T}}$ can be empty. The empty sequence is denoted by ϵ.

A program ($\mathtt{P} \in \mathtt{Program}$) consists of a sequence of classes, the identifier of a main class ($\mathtt{C} \in \mathtt{ClassId}$), and a main expression ($\mathtt{e} \in \mathtt{Expr}$). A program is executed by creating an instance `o` of `C` and then evaluating `e` with `o` as `this` object. We assume that we always have access to the current program `P`, and keep `P` implicit in the notations. Each class ($\mathtt{Cls} \in \mathtt{Class}$) has a class identifier, type variables with upper bounds, a superclass with type arguments, a list of field declarations, and a list of method declarations. `FieldId` is the sort of field identifiers `f`. Like in Java, each class directly or transitively extends the predefined class `Object`.

A type ($^{\mathtt{s}}\mathtt{T} \in {}^{\mathtt{s}}\mathtt{Type}$) is either a non-variable type or a type variable identifier ($\mathtt{X} \in \mathtt{TVarId}$). A non-variable type ($^{\mathtt{s}}\mathtt{N} \in {}^{\mathtt{s}}\mathtt{NType}$) consists of an ownership modifier, a class identifier, and a sequence of type arguments.

An ownership modifier ($\mathtt{u} \in \mathtt{OM}$) can be $\mathtt{peer}_u$, $\mathtt{rep}_u$, or $\mathtt{any}_u$, as well as the modifier $\mathtt{this}_u$, which is used solely as main modifier for the type of `this`. The modifier $\mathtt{this}_u$ may not appear in programs, but is used by the type system to distinguish accesses through `this` from other accesses. We omit the subscript u if it is clear from context that we mean an ownership modifier.

A method ($\mathtt{mt} \in \mathtt{Meth}$) consists of the method type variables with their upper bounds, the purity annotation, the return type, the method identifier ($\mathtt{m} \in \mathtt{MethId}$), the formal method parameters ($\mathtt{x} \in \mathtt{ParId}$) with their types, and an expression as body. The result of evaluating the expression is returned by the method. `ParId` includes the implicit method parameter `this`.

To be able to enforce the owner-as-modifier discipline, we have to distinguish statically between side-effect free (pure) methods and methods that potentially have side effects. Pure methods are marked by the keyword `pure`. In our syntax, we mark all other methods by `nonpure`, although we omit this keyword in our examples. To focus on the essentials of the type system, we do not include purity checks, but they can be added easily [127].

An expression ($\mathtt{e} \in \mathtt{Expr}$) can be the `null` literal, method parameter access, field read, field update, method call, object creation, and cast.

Type checking is performed in a type environment ($^{\mathtt{s}}\Gamma \in {}^{\mathtt{s}}\mathtt{Env}$), which maps the type variables of the enclosing class and method to their upper bounds and method parameters to their types. Since the domains of these mappings are disjoint, we overload the notation, where $^{\mathtt{s}}\Gamma(\mathtt{X})$ refers to the upper bound of type variable `X`, and $^{\mathtt{s}}\Gamma(\mathtt{x})$ refers to the type of method parameter `x`.

$$
\begin{array}{rcll}
\mathtt{P} & \in & \mathtt{Program} & ::= \quad \overline{\mathtt{Cls}}\ \mathtt{C}\ \mathtt{e} \\
\mathtt{Cls} & \in & \mathtt{Class} & ::= \quad \mathtt{class\ C<}\overline{\mathtt{X}\ {}^{\mathtt{s}}\mathtt{N}}\mathtt{>\ extends\ C<}\overline{{}^{\mathtt{s}}\mathtt{T}}\mathtt{>\ \{\ }\overline{\mathtt{f}\ {}^{\mathtt{s}}\mathtt{T}}\mathtt{;}\ \overline{\mathtt{mt}}\ \mathtt{\}} \\
{}^{\mathtt{s}}\mathtt{T} & \in & {}^{\mathtt{s}}\mathtt{Type} & ::= \quad {}^{\mathtt{s}}\mathtt{N} \mid \mathtt{X} \\
{}^{\mathtt{s}}\mathtt{N} & \in & {}^{\mathtt{s}}\mathtt{NType} & ::= \quad \mathtt{u\ C<}\overline{{}^{\mathtt{s}}\mathtt{T}}\mathtt{>} \\
\mathtt{u} & \in & \mathtt{OM} & ::= \quad \mathtt{peer}_u \mid \mathtt{rep}_u \mid \mathtt{any}_u \mid \mathtt{this}_u \\
\mathtt{mt} & \in & \mathtt{Meth} & ::= \quad \mathtt{<}\overline{\mathtt{X}\ {}^{\mathtt{s}}\mathtt{N}}\mathtt{>\ w\ }{}^{\mathtt{s}}\mathtt{T\ m(}\overline{\mathtt{x}\ {}^{\mathtt{s}}\mathtt{T}}\mathtt{)\ \{\ return\ e\ \}} \\
\mathtt{w} & \in & \mathtt{Purity} & ::= \quad \mathtt{pure} \mid \mathtt{nonpure} \\
\mathtt{e} & \in & \mathtt{Expr} & ::= \quad \mathtt{null} \mid \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.f=e} \mid \mathtt{e.m<}\overline{{}^{\mathtt{s}}\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ }{}^{\mathtt{s}}\mathtt{N} \mid \mathtt{(}{}^{\mathtt{s}}\mathtt{T)\ e} \\
{}^{\mathtt{s}}\Gamma & \in & {}^{\mathtt{s}}\mathtt{Env} & ::= \quad \overline{\mathtt{X}\ {}^{\mathtt{s}}\mathtt{N}}\mathtt{;}\ \overline{\mathtt{x}\ {}^{\mathtt{s}}\mathtt{T}}
\end{array}
$$

Figure 4.18: Syntax and type environments

Viewpoint Adaptation

As we already discussed earlier, ownership modifiers express ownership relative to an object, and have

to be adapted whenever the viewpoint changes. As for UJ, we need to *adapt a type* T *from a viewpoint* that is described by another type T′ *to the viewpoint* this. In the following, we omit the phrase "to the viewpoint this". To perform the viewpoint adaptation, we define an overloaded operator $\triangleright$ to: (1) Adapt an ownership modifier from a viewpoint that is described by another ownership modifier; (2) Adapt a type from a viewpoint that is described by an ownership modifier; (3) Adapt a type from a viewpoint that is described by another type.Note, that $\triangleright$ is therefore analogous to $\triangleleft$ from 4.1.

Adapting an Ownership Modifier w.r.t. an Ownership Modifier. We explain viewpoint adaptation using a field access $\mathtt{e}_1.\mathtt{f}$. Analogous adaptations occur for method parameters and results as well as upper bounds of type parameters. Let u be the main modifier of $\mathtt{e}_1$'s type, which expresses ownership relative to this. Let u′ be the main modifier of f's type, which expresses ownership relative to the object that contains f. Then relative to this, the type of the field access $\mathtt{e}_1.\mathtt{f}$ has main modifier $\mathtt{u}\triangleright\mathtt{u}'$.

$$
\begin{array}{rcll}
\triangleright :: \mathtt{OM} \times \mathtt{OM} & \to & \mathtt{OM} & \\
\mathtt{this}\triangleright\mathtt{u}' & = & \mathtt{u}' & \qquad\qquad \mathtt{u}\triangleright\mathtt{this} = \mathtt{u} \\
\mathtt{peer}\triangleright\mathtt{peer} & = & \mathtt{peer} & \qquad\qquad \mathtt{rep}\triangleright\mathtt{peer} = \mathtt{rep} \\
\mathtt{u}\triangleright\mathtt{u}' & = & \mathtt{any} \quad \text{otherwise} &
\end{array}
$$

The field access $\mathtt{e}_1.\mathtt{f}$ illustrates the motivation for this definition: (1) Accesses through this (that is, $\mathtt{e}_1$ is the variable this) do not require a viewpoint adaptation since the ownership modifier of the field is already relative to this. (2) In the formalization of subtyping (see ST-1) we combine an ownership modifier u with $\mathtt{this}_u$. Again, this does not require a viewpoint adaptation.

(3) If the main modifiers of both $\mathtt{e}_1$ and f are peer, then the object referenced by $\mathtt{e}_1$ has the same owner as this and the object referenced by $\mathtt{e}_1.\mathtt{f}$ has the same owner as $\mathtt{e}_1$ and, thus, the same owner as this. Consequently, the main modifier of $\mathtt{e}_1.\mathtt{f}$ is also peer. (4) If the main modifier of $\mathtt{e}_1$ is rep and the main modifier of f is peer, then the main modifier of $\mathtt{e}_1.\mathtt{f}$ is rep, because the object referenced by $\mathtt{e}_1$ is owned by this and the object referenced by $\mathtt{e}_1.\mathtt{f}$ has the same owner as $\mathtt{e}_1$, that is, this. (5) In all other cases, we cannot determine statically that the object referenced by $\mathtt{e}_1.\mathtt{f}$ has the same owner as this or is owned by this. Therefore, in these cases the main modifier of $\mathtt{e}_1.\mathtt{f}$ is any.

Adapting a Type w.r.t. an Ownership Modifier. As explained in Sec. 4.3.1, type variables are not subject to viewpoint adaptation. For non-variable types, we determine the adapted main modifier using the auxiliary function $\triangleright_m$ below and adapt the type arguments recursively:

$$
\begin{array}{rcll}
\triangleright :: \mathtt{OM} \times {}^{\mathtt{s}}\mathtt{Type} & \to & {}^{\mathtt{s}}\mathtt{Type} & \\
\mathtt{u}\triangleright\mathtt{X} & = & \mathtt{X} & \\
\mathtt{u}\triangleright\mathtt{N} & = & (\mathtt{u}\triangleright_m\mathtt{N})\ \mathtt{C<}\overline{\mathtt{u}\triangleright\mathtt{T}}\mathtt{>} & \quad \text{where } \mathtt{N} = \mathtt{u}'\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}
\end{array}
$$

The adapted main modifier is determined by $\mathtt{u}\triangleright\mathtt{u}'$, except for unsafe (covariance-like) viewpoint adaptations, as described in Sec. 4.3.1, in which case it is any. Unsafe adaptations occur if at least one of N's type arguments contains the modifier rep, u′ is peer, and u is rep or peer. This leads to the following definition:

$$
\begin{array}{rcll}
\triangleright_m :: \mathtt{OM} \times {}^{\mathtt{s}}\mathtt{NType} & \to & \mathtt{OM} & \\
\mathtt{u}\triangleright_m\mathtt{u}'\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} & = & \begin{cases} \mathtt{any} & \text{if } (\mathtt{u} = \mathtt{rep} \vee \mathtt{u} = \mathtt{peer}) \wedge \mathtt{u}' = \mathtt{peer} \wedge \mathtt{rep} \in \overline{\mathtt{T}} \\ \mathtt{u}\triangleright\mathtt{u}' & \text{otherwise} \end{cases} &
\end{array}
$$

The notation $\mathtt{u} \in \overline{\mathtt{T}}$ expresses that at least one type $\mathtt{T}_i$ or its (transitive) type arguments contain ownership modifier u.

Adapting a Type w.r.t. a Type. We adapt a type T from the viewpoint described by another type, u $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$:

$$
\begin{array}{rcll}
\triangleright :: {}^{\mathtt{s}}\mathtt{NType} \times {}^{\mathtt{s}}\mathtt{Type} & \to & {}^{\mathtt{s}}\mathtt{Type} & \\
\mathtt{u}\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\triangleright\mathtt{T} & = & (\mathtt{u}\triangleright\mathtt{T})[\overline{\mathtt{T}/\mathtt{X}}] & \quad \text{where } \overline{\mathtt{X}} = \mathrm{dom}(\mathtt{C})
\end{array}
$$

The operator $\triangleright$ adapts the ownership modifiers of T and then substitutes the type arguments $\overline{\mathtt{T}}$ for the type variables $\overline{\mathtt{X}}$ of C. This substitution is denoted by $[\overline{\mathtt{T}/\mathtt{X}}]$. Since the type arguments already are relative to

$$\text{SC-1}\ \frac{\texttt{class C<}\overline{\texttt{X}}\ \texttt{\_> extends C'<}\overline{\texttt{T'}}\texttt{>}}{\texttt{C<}\overline{\texttt{X}}\texttt{>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{>}} \qquad \text{SC-2}\ \frac{}{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C<}\overline{\texttt{T}}\texttt{>}}$$

$$\text{SC-3}\ \frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C''<}\overline{\texttt{T''}}\texttt{>} \quad \texttt{C''<}\overline{\texttt{T''}}\texttt{>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{>}}{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{>}} \qquad \text{SC-4}\ \frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{>}}{\texttt{C<}\overline{\texttt{T}}\texttt{[T''/X'']>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{[T''/X'']>}}$$

Figure 4.19: Rules for subclassing

`this`, they are not subject to viewpoint adaptation. Therefore, the substitution of type variables happens after the viewpoint adaptation of `T`'s ownership modifiers. For a declaration `class C<`$\overline{\texttt{X}}$ `_> ...`, dom(`C`) denotes `C`'s type variables $\overline{\texttt{X}}$.

Note that the first parameter is a non-variable type, because concrete ownership information `u` is needed to adapt the viewpoint and the actual type arguments $\overline{\texttt{T}}$ are needed to substitute the type variables $\overline{\texttt{X}}$. In the type rules, subsumption will be used to replace type variables by their upper bounds and thereby obtain a concrete type as first argument of $\triangleright$.

Example. The hypothetical call `map.altIterator()` in `main` (Fig. 4.17) illustrates the most interesting viewpoint adaptation, which we discussed in Sec. 4.3.1. The type of this call is the adaptation of `peer IterImpl<K,V,rep Node<K,V>>` (the return type of `altIterator`) from `rep Map<rep ID,any Data>` (the type of the receiver expression). According to the above definition, we first adapt the return type from the viewpoint of the receiver type, `rep`, and then substitute type variables.

The type arguments of the adapted type are obtained by applying viewpoint adaptation recursively to the type arguments. The type variables `K` and `V` are not affected by the adaptation. For the third type argument, `rep` $\triangleright$ `rep Node<K,V>` yields `any Node<K,V>` because `rep` $\triangleright$ `rep`=`any`, and again because the type variables `K` and `V` are not subject to viewpoint adaptation. Note that here, an ownership modifier of a type argument is promoted from `rep` to `any`. Therefore, to avoid unsafe covariance-like adaptations, the main modifier of the adapted type must be `any`. This is, indeed, the case, as the main modifier is determined by `rep` $\triangleright_m$ `peer IterImpl<K,V,rep Node<K,V>>`, which yields `any`.

So far, the adaptation yields `any IterImpl<K,V,any Node<K,V>>`. Now we substitute the type variables `K` and `V` by the instantiations given in the receiver type, `rep ID` and `any Data`, and obtain the type of the call:

```
any IterImpl<rep ID, any Data, any Node<rep ID,any Data>>
```

Subclassing and Subtyping

We use the term *subclassing* to refer to the relation on classes as declared in a program by the `extends` keyword, irrespective of main modifiers. *Subtyping* takes main modifiers into account.

Subclassing. The subclass relation $\sqsubseteq$ is defined on instantiated classes, which are denoted by `C<`$\overline{\texttt{T}}$`>`. The subclass relation is the smallest relation satisfying the rules in Fig. 4.19. Each uninstantiated class is a subclass of the class it extends (SC-1). The form `class C<`$\overline{\texttt{X N}}$`> extends C'<`$\overline{\texttt{T'}}$`> { `$\overline{\texttt{f T}}$`; `$\overline{\texttt{m}}$` }`, or a prefix thereof, expresses that the program contains such a class declaration. Subclassing is reflexive (SC-2) and transitive (SC-3). Subclassing is preserved by substitution of type arguments for type variables (SC-4). Note that such substitutions may lead to ill-formed types, for instance, when the upper bound of a substituted type variable is not respected. We prevent such types by well-formedness rules, presented in Fig. 4.21.

Subtyping. The subtype relation `<:` is defined on types. The judgment $\Gamma \vdash \texttt{T} <: \texttt{T}'$ expresses that type `T` is a subtype of type `T'` in type environment Γ. The environment is needed since types may contain type variables. The rules for this subtyping judgment are presented in Fig. 4.20. Two types with the same main modifier are subtypes if the corresponding classes are subclasses. Ownership modifiers in the `extends` clause ($\overline{\texttt{T'}}$) are relative to the instance of class `C`, whereas the modifiers in a type are relative to `this`. Therefore,

$$\text{ST-1}\frac{\texttt{C<}\overline{\texttt{T}}\texttt{>} \sqsubseteq \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>}}{\Gamma \vdash \texttt{u C<}\overline{\texttt{T}}\texttt{>} <: \texttt{u}\triangleright(\texttt{this}_u\ \texttt{C}'\texttt{<}\overline{\texttt{T}'}\texttt{>})} \qquad \text{ST-2}\frac{}{\Gamma \vdash \texttt{this}_u\ \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{peer C<}\overline{\texttt{T}}\texttt{>}}$$

$$\text{ST-3}\frac{\Gamma \vdash \texttt{T} <: \texttt{T}'' \quad \Gamma \vdash \texttt{T}'' <: \texttt{T}'}{\Gamma \vdash \texttt{T} <: \texttt{T}'} \qquad \text{ST-4}\frac{}{\Gamma \vdash \texttt{X} <: \Gamma(\texttt{X})} \qquad \text{ST-5}\frac{\texttt{T} <:_\texttt{a} \texttt{T}'}{\Gamma \vdash \texttt{T} <: \texttt{T}'}$$

$$\text{TA-1}\frac{}{\texttt{T} <:_\texttt{a} \texttt{T}} \qquad \text{TA-2}\frac{\overline{\texttt{T} <:_\texttt{a} \texttt{T}'}}{\texttt{u C<}\overline{\texttt{T}}\texttt{>} <:_\texttt{a} \texttt{any C<}\overline{\texttt{T}'}\texttt{>}}$$

Figure 4.20: Rules for subtyping and limited covariance

$\overline{\texttt{T}'}$ has to be adapted from the viewpoint of the `C` instance to `this` (ST-1). Since both $\texttt{this}_u$ and `peer` express that an object has the same owner as `this`, a type with main modifier $\texttt{this}_u$ is a subtype of the corresponding type with main modifier `peer` (ST-2). This rule allows us to treat `this` as an object of a `peer` type. Subtyping is transitive (ST-3). A type variable is a subtype of its upper bound in the type environment (ST-4). Two types are subtypes, if they obey the limited covariance described in Sec. 4.3.1 (ST-5). Covariant subtyping is expressed by the relation $<:_\texttt{a}$. Covariant subtyping is reflexive (TA-1). A supertype may have more general type arguments than the subtype if the main modifier of the supertype is `any` (TA-2). Note that the sequences $\overline{\texttt{T}}$ and $\overline{\texttt{T}'}$ in rule TA-2 can be empty, which allows one to derive, for instance, `peer Object` $<:_\texttt{a}$ `any Object`. Reflexivity of $<:$ follows from TA-1 and ST-5.

In our example, using rule TA-1 for `K` and `V`, and rule TA-2 we obtain `rep Node<K,V>` $<:_\texttt{a}$ `any Node<K,V>`. Rules TA-2 and ST-5 allow us to derive

`peer IterImpl<K,V,rep Node<K,V>>` $<:$ `any IterImpl<K,V,any Node<K,V>>`,

which is an example for limited covariance. Note that it is not possible to derive

`peer IterImpl<K,V,rep Node<K,V>>` $<:$ `peer IterImpl<K,V,any Node<K,V>>`;

that would be unsafe covariant subtyping as discussed in Sec. 4.3.1.

Lookup Functions

In this subsection, we define the functions to look up the type of a field or the signature of a method.

Field Lookup. The function ${}^{\texttt{s}}fType(\texttt{C}, \texttt{f})$ yields the type of field `f` as declared in class `C`. The result is undefined if `f` is not declared in `C`. Since identifiers are assumed to be globally unique, there is only one declaration for each field identifier.

$$\text{SFT}\frac{\texttt{class C<\_> extends \_<\_> \{ } \ldots \texttt{T f} \ldots \texttt{; \_ \}}}{{}^{\texttt{s}}fType(\texttt{C}, \texttt{f}) = \texttt{T}}$$

Method Lookup. The function $mType(\texttt{C}, \texttt{m})$ yields the signature of method `m` as declared in class `C`. The result is undefined if `m` is not declared in `C`. We do not allow overloading of methods; therefore, the method identifier is sufficient to uniquely identify a method.

$$\text{SMT}\frac{\texttt{class C<\_> extends \_<\_> \{\_ ; } \ldots \texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{> w T}_r\ \texttt{m(}\overline{\texttt{x T}_p}\texttt{)} \ldots \texttt{\}}}{mType(\texttt{C}, \texttt{m}) = \texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{> w T}_r\ \texttt{m(}\overline{\texttt{x T}_p}\texttt{)}}$$

Well-Formedness

In this subsection, we define well-formedness of types, methods, classes, programs, and type environments. The well-formedness rules are summarized in Fig. 4.21 and explained in the following.

Well-Formed Types. The judgment $\Gamma \vdash \texttt{T ok}$ expresses that type `T` is well-formed in type environment Γ. Type variables are well-formed, if they are contained in the type environment (WFT-1). A non-variable type $\texttt{u C<}\overline{\texttt{T}}\texttt{>}$ is well-formed if its type arguments $\overline{\texttt{T}}$ are well-formed and for each type parameter the actual type

$$\text{WFT-1}\ \frac{\texttt{X} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{X ok}} \qquad \text{WFT-2}\ \frac{\texttt{class C<}\overline{\_\ \texttt{N}}\texttt{>}\ \ldots \quad \Gamma \vdash \overline{\texttt{T}}\ \texttt{ok} \qquad \Gamma \vdash \overline{\texttt{T}} <: ((\texttt{u C<}\overline{\texttt{T}}\texttt{>}) \triangleright \overline{\texttt{N}})}{\Gamma \vdash \texttt{u C<}\overline{\texttt{T}}\texttt{> ok}}$$

$$\text{WFM-1}\ \frac{\begin{array}{c}\Gamma = \overline{\texttt{X}_m\ \texttt{N}_b}, \overline{\texttt{X N}}; \texttt{this}\ (\texttt{this}_u\ \texttt{C<}\overline{\texttt{X}}\texttt{>}), \overline{\texttt{x T}_p} \\ \Gamma \vdash \texttt{T}_r, \overline{\texttt{N}_b}, \overline{\texttt{T}_p}\ \texttt{ok} \qquad \Gamma \vdash \texttt{e} : \texttt{T}_r \qquad \mathit{override}(\texttt{C}, \texttt{m}) \\ \texttt{w} = \texttt{pure}\ \Rightarrow\ (\overline{\texttt{T}_p} = \overline{\texttt{any}\triangleright\texttt{T}_p}\ \wedge\ \overline{\texttt{N}_b} = \overline{\texttt{any}\triangleright\texttt{N}_b})\end{array}}{\texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{> w T}_r\ \texttt{m(}\overline{\texttt{x T}_p}\texttt{)}\ \{\ \texttt{return e}\ \}\ \texttt{ok in C<}\overline{\texttt{X N}}\texttt{>}}$$

$$\text{WFM-2}\ \frac{\begin{array}{c}\forall\ \texttt{class C'<}\overline{\texttt{X' N'}}\texttt{>}:\ \ \texttt{C<}\overline{\texttt{X}}\texttt{>} \sqsubseteq \texttt{C'<}\overline{\texttt{T'}}\texttt{>} \wedge \mathrm{dom}(\texttt{C}) = \overline{\texttt{X}} \Rightarrow \\ \mathit{mType}(\texttt{C'}, \texttt{m})\ \text{is undefined}\ \vee\ \mathit{mType}(\texttt{C}, \texttt{m}) = \mathit{mType}(\texttt{C'}, \texttt{m})[\overline{\texttt{T'}/\texttt{X'}}]\end{array}}{\mathit{override}(\texttt{C}, \texttt{m})}$$

$$\text{WFC}\ \frac{\begin{array}{c}\overline{\texttt{X N}}; \_ \vdash \overline{\texttt{N}}, \overline{\texttt{T}}, (\texttt{this}_u\ \texttt{C'<}\overline{\texttt{T'}}\texttt{>})\ \texttt{ok} \\ \overline{\texttt{mt}}\ \texttt{ok in C<}\overline{\texttt{X N}}\texttt{>} \qquad \texttt{rep} \notin \overline{\texttt{N}}\end{array}}{\texttt{class C<}\overline{\texttt{X N}}\texttt{> extends C'<}\overline{\texttt{T'}}\texttt{> \{ }\overline{\texttt{f T}}\texttt{; }\overline{\texttt{mt}}\ \texttt{\}\ ok}}$$

$$\text{WFP}\ \frac{\begin{array}{c}\overline{\texttt{Cls}}\ \texttt{ok} \\ \texttt{class C<>}\ldots \in \overline{\texttt{Cls}} \\ \epsilon;\ \texttt{this}\ (\texttt{this}_u\ \texttt{C<>}) \vdash \texttt{e} : \texttt{N}\end{array}}{\overline{\texttt{Cls}}, \texttt{C}, \texttt{e ok}} \qquad \text{SWFE}\ \frac{\begin{array}{c}\Gamma = \overline{\texttt{X N}}, \overline{\texttt{X' N'}}\ ; \\ \texttt{this}\ (\texttt{this}_u\ \texttt{C<}\overline{\texttt{X}}\texttt{>}), \overline{\texttt{x T}} \\ \texttt{class C<}\overline{\texttt{X N}}\texttt{>}\ \ldots \\ \Gamma \vdash \overline{\texttt{N}}, \overline{\texttt{N'}}, \overline{\texttt{T}}\ \texttt{ok}\end{array}}{\Gamma\ \texttt{ok}}$$

Figure 4.21: Well-formedness rules

argument is a subtype of the upper bound, adapted from the viewpoint `u C<`$\overline{\texttt{T}}$`>` (WFT-2). The viewpoint adaptation is necessary because the type arguments describe ownership relative to the `this` object where `u C<`$\overline{\texttt{T}}$`>` is used, whereas the upper bounds are relative to the object of type `u C<`$\overline{\texttt{T}}$`>`. Note that rule WFT-2 permits type variables of a class `C` to be used in upper bounds of `C`. For instance in class `IterImpl` (Fig. 4.16), type variable `X` is used in its own upper bound, `any MapNode<K, V, X>`.

Well-Formed Methods. The judgment `mt ok in C<`$\overline{\texttt{X N}}$`>` expresses that method `mt` is well-formed in a class `C` with type parameters $\overline{\texttt{X N}}$. According to rule WFM-1, `mt` is well-formed if: (1) the return type, the upper bounds of `mt`'s type variables, and `mt`'s parameter types are well-formed in the type environment that maps `mt`'s and `C`'s type variables to their upper bounds as well as `this` and the explicit method parameters to their types. The type of `this` is the enclosing class, `C<`$\overline{\texttt{X}}$`>`, with main modifier $\texttt{this}_u$; (2) the method body, expression `e`, is well-typed with `mt`'s return type; (3) `mt` respects the rules for overriding, see below; (4) if `mt` is pure then the only ownership modifier that occurs in a parameter type or the upper bound of a method type variable is `any`. We will motivate the fourth requirement when we explain the type rule for method calls.

Method `m` respects the rules for overriding if it does not override a method or if all overridden methods have the identical signatures after substituting type variables of the superclasses by the instantiations given in the subclass (WFM-2). For simplicity, we require that overrides do not change the purity of a method, although overriding non-pure methods by pure methods would be safe.

Well-Formed Classes. The judgment `Cls ok` expresses that class declaration `Cls` is well-formed. According to rule WFC, this is the case if: (1) the upper bounds of `Cls`'s type variables, the types of `Cls`'s fields, and the instantiation of the superclass are well-formed in the type environment that maps `Cls`'s type variables to their upper bounds; (2) `Cls`'s methods are well-formed; (3) `Cls`'s upper bounds do not contain the `rep` modifier.

Note that `Cls`'s upper bounds express ownership relative to the current `Cls` instance. If such an upper bound contains a `rep` modifier, clients of `Cls` cannot instantiate `Cls`. The ownership modifiers of an actual type argument are relative to the client's viewpoint. From this viewpoint, none of the modifiers `peer`, `rep`, or `any` expresses that an object is owned by the `Cls` instance. Therefore, we forbid upper bounds with `rep` modifiers by Requirement (3).

Well-Formed Programs. The judgment $\texttt{P ok}$ expresses that program $\texttt{P}$ is well-formed. According to rule WFP, this holds if all classes in $\texttt{P}$ are well-formed, the main class $\texttt{C}$ is a non-generic class in $\texttt{P}$, and the main expression $\texttt{e}$ is well-typed in an environment with $\texttt{this}$ as an instance of $\texttt{C}$. We omit checks for valid appearances of the ownership modifier $\texttt{this}_u$. As explained earlier, $\texttt{this}_u$ must not occur in the program.

Well-Formed Type Environments. The judgment $\Gamma\ \texttt{ok}$ expresses that type environment Γ is well-formed. According to rule SWFE, this is the case if all upper bounds of type variables and the types of method parameters are well-formed. Moreover, $\texttt{this}$ must be mapped to a non-variable type with main modifier $\texttt{this}_u$ and an uninstantiated class.

Type Rules

We are now ready to present the type rules (Fig. 4.22). The judgment $\Gamma \vdash \texttt{e} : \texttt{T}$ expresses that expression $\texttt{e}$ is well-typed with type $\texttt{T}$ in environment Γ. Our type rules implicitly require types to be well-formed, that is, a type rule is applicable only if all types involved in the rule are well-formed in the respective environment.

$$\text{GT-Subs}\frac{\Gamma \vdash \texttt{e} : \texttt{T} \quad \Gamma \vdash \texttt{T} <: \texttt{T}'}{\Gamma \vdash \texttt{e} : \texttt{T}'} \qquad \text{GT-Var}\frac{\texttt{x} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{x} : \Gamma(\texttt{x})} \qquad \text{GT-Null}\frac{\texttt{T} \neq \texttt{this}_u\ \texttt{\_<\_>}}{\Gamma \vdash \texttt{null} : \texttt{T}}$$

$$\text{GT-New}\frac{\texttt{N} \neq \texttt{any}_u\ \texttt{\_<\_>}}{\Gamma \vdash \texttt{new N} : \texttt{N}} \qquad \text{GT-Cast}\frac{\Gamma \vdash \texttt{e}_0 : \texttt{T}_0}{\Gamma \vdash \texttt{(T) e}_0 : \texttt{T}}$$

$$\text{GT-Read}\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \\ \texttt{N}_0 = \texttt{\_ C}_0\texttt{<\_>} \\ \texttt{T}_1 = \mathit{fType}(\texttt{C}_0, \texttt{f})\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.f} : \texttt{N}_0 \triangleright \texttt{T}_1} \qquad \text{GT-Upd}\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \quad \texttt{N}_0 = \texttt{u}_0\ \texttt{C}_0\texttt{<\_>} \\ \texttt{T}_1 = \mathit{fType}(\texttt{C}_0, \texttt{f}) \\ \Gamma \vdash \texttt{e}_2 : \texttt{N}_0 \triangleright \texttt{T}_1 \\ \texttt{u}_0 \neq \texttt{any} \qquad \mathit{rp}(\texttt{u}_0, \texttt{T}_1)\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.f=e}_2 : \texttt{N}_0 \triangleright \texttt{T}_1}$$

$$\text{GT-Invk}\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_0 : \texttt{N}_0 \qquad \texttt{N}_0 = \texttt{u}_0\ \texttt{C}_0\texttt{<\_>} \\ \mathit{mType}(\texttt{C}_0, \texttt{m}) = \texttt{<}\overline{\texttt{X}_m\ \texttt{N}_b}\texttt{>}\ \texttt{w}\ \texttt{T}_r\ \texttt{m}(\overline{\texttt{x}\ \texttt{T}_p}) \\ \Gamma \vdash \overline{\texttt{T}} <: (\texttt{N}_0 \triangleright \overline{\texttt{N}_b})[\overline{\texttt{T}/\texttt{X}_m}] \qquad \Gamma \vdash \overline{\texttt{e}_2} : (\texttt{N}_0 \triangleright \overline{\texttt{T}_p})[\overline{\texttt{T}/\texttt{X}_m}] \\ (\texttt{u}_0 = \texttt{any} \Rightarrow \texttt{w} = \texttt{pure}) \qquad \mathit{rp}(\texttt{u}_0, \overline{\texttt{T}_p} \circ \overline{\texttt{N}_b})\end{array}}{\Gamma \vdash \texttt{e}_0\texttt{.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{e}_2}) : (\texttt{N}_0 \triangleright \texttt{T}_r)[\overline{\texttt{T}/\texttt{X}_m}]}$$

Figure 4.22: Type rules

An expression of type $\texttt{T}$ can also be typed with $\texttt{T}$'s supertypes (GT-Subs). The type of method parameters (including $\texttt{this}$) is determined by a lookup in the type environment (GT-Var). The $\texttt{null}$-reference can have any type other than a $\texttt{this}_u$ type (GT-Null). Objects must be created in a specific context. Therefore only non-variable types with an ownership modifier other than $\texttt{any}_u$ are allowed for object creations (GT-New). The rule for casts (GT-Cast) is straightforward; it could be strengthened to prevent more cast errors statically, but we omit this check since it is not strictly needed.

As explained in detail in Sec. 4.3.2, the type of a field access is determined by adapting the declared type of the field from the viewpoint described by the type of the receiver (GT-Read). If this type is a type variable, subsumption is used to go to its upper bound because *fType* is defined on class identifiers. Subsumption is also used for inherited fields to ensure that $\texttt{f}$ is actually declared in $\texttt{C}_0$. (Recall that $\mathit{fType}(\texttt{C}_0, \texttt{f})$ is undefined otherwise.)

For a field update, the right-hand side expression must be typable as the viewpoint-adapted field type, which is also the type of the whole field update expression (GT-Upd). The rule is analogous to field read, but has two additional requirements. First, the main modifier $\texttt{u}_0$ of the type of the receiver expression must not be $\texttt{any}$. With the owner-as-modifier discipline, a method must not update fields of objects in arbitrary contexts. Second, the requirement $\mathit{rp}(\texttt{u}_0, \texttt{T}_1)$ enforces that $\texttt{f}$ is updated through receiver $\texttt{this}$ if its declared type $\texttt{T}_1$ contains a $\texttt{rep}$ modifier. For all other receivers, the viewpoint adaptation $\texttt{N}_0 \triangleright \texttt{T}_1$ yields an $\texttt{any}$ type, but it is obviously unsafe to update $\texttt{f}$ with an object with an arbitrary owner. It is convenient to define *rp*

for sequences of types. The definition uses the fact that the ownership modifier $\texttt{this}_u$ is only used for the type of `this`:

$$\begin{array}{rcl} rp :: \texttt{OM} \times \overline{\texttt{sType}} & \rightarrow & \mathit{bool} \\ rp(\texttt{u}, \overline{\texttt{T}}) & = & \texttt{u} = \texttt{this}_u \vee (\forall i : \texttt{rep} \notin \texttt{T}_i) \end{array}$$

The rule for method calls (GT-Invk) is in many ways similar to field reads (for result passing) and updates (for argument passing). The method signature is determined using the receiver type $\texttt{N}_0$ and subsumption. The type of the invocation expression is determined by viewpoint adaptation of the return type $\texttt{T}_r$ from the receiver type $\texttt{N}_0$. Modulo subsumption, the actual method arguments must have the formal parameter types, adapted from $\texttt{N}_0$ and with actual type arguments $\overline{\texttt{T}}$ substituted for the method's type variables $\texttt{X}_m$. For instance, in the call `first.init(key, value, first)` in method `put` (Fig. 4.14), the adapted third formal parameter type is `rep Node<K,V>` $\triangleright$ `peer Node<K,V>` (note that `Node` substitutes the type variable `X` by `peer Node<K,V>`). This adaptation yields `rep Node<K,V>`, which is also the type of the third actual method argument.

To enforce the owner-as-modifier discipline, only pure methods may be called on receivers with main modifier `any`. For a call on a receiver with main modifier `any`, the viewpoint-adapted formal parameter type contains only the modifier `any`. Consequently, arguments with arbitrary owners can be passed. For this to be type safe, pure methods must not expect arguments with specific owners. This is enforced by rule WFM-1 (Fig. 4.21). Finally, if the receiver is different from `this`, then neither the formal parameter types nor the upper bounds of the method's type variables must contain `rep`.

4.3.3 Runtime Model

In this section, we explain the runtime model of Generic Universe Types. We present the heap model, the runtime type information, well-formedness conditions, and an operational semantics.

Heap Model

Fig. 4.23 defines our model of the heap. The prefix $\texttt{r}$ distinguishes sorts of the runtime model from their static counterparts.

$$\begin{array}{rcll} \texttt{h} & \in & \texttt{Heap} & = & \texttt{Addr} \rightarrow \texttt{Obj} \\ \iota & \in & \texttt{Addr} & = & \text{Address} \mid \texttt{null}_a \\ \texttt{o} & \in & \texttt{Obj} & = & {}^{\texttt{r}}\texttt{T, Fs} \\ {}^{\texttt{r}}\texttt{T} & \in & {}^{\texttt{r}}\texttt{Type} & = & \iota_o\ \texttt{C<}\overline{{}^{\texttt{r}}\texttt{T}}\texttt{>} \\ \texttt{Fs} & \in & \texttt{Fields} & = & \texttt{FieldId} \rightarrow \texttt{Addr} \\ \iota_o & \in & \texttt{OwnerAddr} & = & \iota \mid \texttt{any}_a \\ {}^{\texttt{r}}\Gamma & \in & {}^{\texttt{r}}\texttt{Env} & = & \overline{\texttt{X}\ {}^{\texttt{r}}\texttt{T}};\ \overline{\texttt{x}\ \iota} \end{array}$$

Figure 4.23: Definitions for the heap model

A heap ($\texttt{h} \in \texttt{Heap}$) maps addresses to objects. An address ($\iota \in \texttt{Addr}$) can be the special null-reference $\texttt{null}_a$. An object ($\texttt{o} \in \texttt{Obj}$) consist of its runtime type and a mapping from field identifiers to the addresses stored in the fields.

The runtime type (${}^{\texttt{r}}\texttt{T} \in {}^{\texttt{r}}\texttt{Type}$) of an object `o` consists of the address of `o`'s owner object, of `o`'s class, and of runtime types for the type arguments of this class. We store the runtime type arguments including the associated ownership information explicitly in the heap because this information is needed in the runtime checks for casts. In that respect, our runtime model is similar to that of the .NET CLR [100]. The owner address of objects in the root context is $\texttt{null}_a$. The special owner address $\texttt{any}_a$ is used when the corresponding static type has the $\texttt{any}_u$ modifier. Consider for instance an execution of method `main` (Fig. 4.17), where the address of `this` is 1. The runtime type of the object stored in `map` is 1 `Map<1 ID,`$\texttt{any}_a$ `Data>`. For simplicity we drop the subscript o from ι_o whenever it is clear from context whether we refer to an `Addr` or an `OwnerAddr`.

The first component of a runtime environment ($^{\mathtt{r}}\Gamma \in {}^{\mathtt{r}}\mathtt{Env}$) maps method type variables to their runtime types. The second component is the stack, which maps method parameters to the addresses they store.

Subtyping on Runtime Types. Judgment $\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'$ expresses that the runtime type $^{\mathtt{r}}\mathtt{T}$ is a subtype of $^{\mathtt{r}}\mathtt{T}'$ from the viewpoint of address ι. The viewpoint, ι, is required in order to give meaning to the ownership modifier `rep`. Subtyping for runtime types is defined in Fig. 4.24. Subtyping is transitive (RT-3), and allows owner-invariant (RT-1) and covariant subtyping (RT-2).

Rule RTL introduces owner-invariant subtyping $<:_1$ and defines how subtyping follows subclassing if (1) the runtime types have the same owner address ι', (2) in the type arguments, the ownership modifiers $\mathtt{this}_u$ and `peer` are substituted by the owner address ι' of the runtime types (we use the same owner address for both modifiers since they both express ownership by the owner of `this`), (3) `rep` is substituted by the viewpoint address ι, (4) $\mathtt{any}_u$ is substituted by $\mathtt{any}_a$, (5) the type variables $\overline{\mathtt{X}}$ of the subclass `C` are substituted consistently by $\overline{^{\mathtt{r}}\mathtt{T}}$, and (6) either the owner of ι is ι' or `rep` does not appear in the instantiation of the superclass. This ensures that the substitution of ι for `rep`-modifiers is meaningful. Note that in a well-formed program, $\mathtt{this}_u$ never occurs in a type argument; nevertheless we include the substitution for consistency. Rule RTL gives the most concrete runtime type deducible from static subclassing.

$$\text{RT-1}\ \frac{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <:_1 {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'} \qquad \text{RT-2}\ \frac{{}^{\mathtt{r}}\mathtt{T} <:_a {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'} \qquad \text{RT-3}\ \frac{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'' \quad \mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T}'' <: {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'}$$

$$\text{RTL}\ \frac{\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>} \sqsubseteq \mathtt{C}'\texttt{<}\overline{{}^{\mathtt{s}}\mathtt{T}}\texttt{>} \qquad \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad \mathit{owner}(\mathtt{h}, \iota) = \iota' \ \vee\ \mathtt{rep} \notin \overline{{}^{\mathtt{s}}\mathtt{T}}}{\mathtt{h}, \iota \vdash \iota'\ \mathtt{C}\texttt{<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>} <:_1 \iota'\ \mathtt{C}'\texttt{<}\overline{{}^{\mathtt{s}}\mathtt{T}}[\iota'/\mathtt{this}_u, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{\mathtt{r}}\mathtt{T}/\mathtt{X}}]\texttt{>}}$$

$$\text{RTA-1}\ \frac{}{{}^{\mathtt{r}}\mathtt{T} <:_a {}^{\mathtt{r}}\mathtt{T}} \qquad \text{RTA-2}\ \frac{\overline{{}^{\mathtt{r}}\mathtt{T} <:_a {}^{\mathtt{r}}\mathtt{T}'}}{\iota'\ \mathtt{C}\texttt{<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>} <:_a \mathtt{any}_a\ \mathtt{C}\texttt{<}\overline{{}^{\mathtt{r}}\mathtt{T}'}\texttt{>}} \qquad \text{RTH-1}\ \frac{h(\iota) = {}^{\mathtt{r}}\mathtt{T}, \_ \quad \mathtt{h}, \iota \vdash {}^{\mathtt{r}}\mathtt{T} <: {}^{\mathtt{r}}\mathtt{T}'}{\mathtt{h} \vdash \iota : {}^{\mathtt{r}}\mathtt{T}'}$$

$$\text{RTH-2}\ \frac{}{\mathtt{h} \vdash \mathtt{null}_a : {}^{\mathtt{r}}\mathtt{T}'} \qquad \text{RTS}\ \frac{\mathtt{h} \vdash \iota : \mathit{dyn}({}^{\mathtt{s}}\mathtt{T}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \quad {}^{\mathtt{s}}\mathtt{T} = \mathtt{this}_u\ \_\texttt{<}\_\texttt{>} \Rightarrow \iota = {}^{\mathtt{r}}\Gamma(\mathtt{this})}{\mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T}}$$

Figure 4.24: Rules for subtyping on runtime types

As for subtyping for static types, we have limited covariance for runtime types. Covariant subtyping is expressed by the relation $<:_a$. The rules for limited covariance, RTA-1 and RTA-2, are analogous to the rules TA-1 and TA-2 for static types (Fig. 4.20). Reflexivity of $<:$ follows from RTA-1 and RT-2.

The judgment $\mathtt{h} \vdash \iota : {}^{\mathtt{r}}\mathtt{T}'$ expresses that in heap `h`, the address ι has type $^{\mathtt{r}}\mathtt{T}'$. The type of ι is determined by the type of the object at ι and the subtype relation (RTH-1). The `null` reference can have any type (RTH-2).

Finally, the judgment $\mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T}$ expresses that in heap `h` and runtime environment $^{\mathtt{r}}\Gamma$, the address ι has a runtime type that corresponds to the static type $^{\mathtt{s}}\mathtt{T}$ (see below for the definition of *dyn*) and that the main modifier $\mathtt{this}_u$ is used solely for the type of `this` (RTS).

From Static Types to Runtime Types. Static types and runtime types are related by the following *dynamization function*, which is defined by rule DYN:

$$\mathit{dyn} :: {}^{\mathtt{s}}\mathtt{Type} \times \mathtt{Heap} \times {}^{\mathtt{r}}\mathtt{Env} \to {}^{\mathtt{r}}\mathtt{Type}$$

$$\text{DYN}\ \frac{{}^{\mathtt{r}}\Gamma = \overline{\mathtt{X}'\ {}^{\mathtt{r}}\mathtt{T}'};\ \mathtt{this}\ \iota, \_ \qquad \mathtt{h}, \iota \vdash \mathtt{h}(\iota)\!\downarrow_1\ <:_1 \iota'\ \mathtt{C}\texttt{<}\overline{{}^{\mathtt{r}}\mathtt{T}}\texttt{>} \quad \mathrm{dom}(\mathtt{C}) = \overline{\mathtt{X}} \qquad \mathrm{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \overline{\mathtt{X}} \circ \overline{\mathtt{X}'}}{\mathit{dyn}({}^{\mathtt{s}}\mathtt{T}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) = {}^{\mathtt{s}}\mathtt{T}[\iota'/\mathtt{this}, \iota'/\mathtt{peer}, \iota/\mathtt{rep}, \mathtt{any}_a/\mathtt{any}_u, \overline{{}^{\mathtt{r}}\mathtt{T}/\mathtt{X}}, \overline{{}^{\mathtt{r}}\mathtt{T}'/\mathtt{X}'}]}$$

This function maps a static type $^{\mathtt{s}}\mathtt{T}$ to the corresponding runtime type. The viewpoint is described by a heap `h` and a runtime environment $^{\mathtt{r}}\Gamma$. In $^{\mathtt{s}}\mathtt{T}$, *dyn* substitutes `rep` by the address of the `this` object (ι), `peer` and $\mathtt{this}_u$ by the owner of ι (ι'), and $\mathtt{any}_u$ by $\mathtt{any}_a$. It also substitutes all type variables in $^{\mathtt{s}}\mathtt{T}$

by the instantiations given in ι' C<$\overline{\texttt{r}\,\texttt{T}}$>, a supertype of ι's runtime type, or in the runtime environment. The substitutions performed by *dyn* are analogous to the ones in rule RTL (Fig. 4.24), which also involves mapping static types to runtime types. We do not use *dyn* in RTL to avoid that the definitions of $<:$ and *dyn* are mutually recursive. We use projection $\downarrow_i$ to select the i-th component of a tuple, for instance, the runtime type and field mapping of an object.

Note that the outcome of *dyn* depends on finding ι' C<$\overline{\texttt{r}\,\texttt{T}}$>, an appropriate supertype of the runtime type of the this object ι, which contains substitutions for all type variables not mapped by the environment (free($^{\texttt{s}}\texttt{T}$) yields the free type variables in $^{\texttt{s}}\texttt{T}$). Thus, one may wonder whether there is more than one such appropriate superclass. However, because type variables are globally unique, if the free variables of $^{\texttt{s}}\texttt{T}$ are in the domain of a class then they are not in the domain of any other class. To obtain the most precise ownership information we use the owner-invariant runtime subtype relation $<:_{\texttt{l}}$ defined in rule RTL.

To illustrate dynamization, consider an execution of put (Fig. 4.14), in an environment $^{\texttt{r}}\Gamma$ whose this object has address 3 and a heap h where address 3 has runtime type 1 Map<1 ID, any$_a$ Data> (see Fig. 4.13). We determine the runtime type of the object created by new rep Node<K,V>. The dynamization of the type of the new object w.r.t. h and $^{\texttt{r}}\Gamma$ is *dyn*(rep Node<K,V>, h, $^{\texttt{r}}\Gamma$), which yields 3 Node<1 ID, any$_a$ Data>. This runtime type correctly reflects that the new object is owned by this (owner address 3) and has the same type arguments as the runtime type of this.

It is convenient to define the following overloaded version of *dyn*:

$$dyn(^{\texttt{s}}\texttt{T}, \texttt{h}, \iota) \quad = \quad dyn(^{\texttt{s}}\texttt{T}, \texttt{h}, (\epsilon; \texttt{this}\ \iota))$$

Lookup Functions

In this subsection, we define the functions to look up the runtime type of a field or the body of a method.

Field Lookup. The runtime type of a field f is essentially the dynamization of its static type. The function $^{\texttt{r}}fType(\texttt{h}, \iota, \texttt{f})$ yields the runtime type of f in an object at address ι in heap h. In its definition (RFT, in Fig. 4.25), C is the runtime class of ι, and C' is the superclass of C which contains the definition of f.

Method Lookup. The function $mBody(\texttt{C}, \texttt{m})$ yields a tuple consisting of m's body expression as well as the identifiers of its formal parameters and type variables. This is trivial if m is declared in C (RMT-1, Fig. 4.25). Otherwise, m is looked up in C's superclass C′ (RMT-2).

Well-Formedness

In this subsection, we define well-formedness of runtime types, heaps, and runtime environments. The rules are presented in Fig. 4.25.

Well-Formed Runtime Types. The judgment $\texttt{h}, \iota \vdash \iota'$ C<$\overline{\texttt{r}\,\texttt{T}}$> ok expresses that runtime type ι' C<$\overline{\texttt{r}\,\texttt{T}}$> is well-formed for viewpoint address ι in heap h. According to rule WFRT, the owner address ι' must be the address of an object in the heap h or one of the special owners $\texttt{null}_a$ and $\texttt{any}_a$. All type arguments must also be well-formed types. A runtime type must have a type argument for each type variable of its class. Each runtime type argument must be a subtype of the dynamization of the type variable's upper bound. We use $\texttt{h}, ^{\texttt{r}}\Gamma \vdash ^{\texttt{r}}\texttt{T}$ ok as shorthand for $\texttt{h}, ^{\texttt{r}}\Gamma(\texttt{this}) \vdash ^{\texttt{r}}\texttt{T}$ ok.

Well-Formed Heaps. A heap h is well-formed, denoted by h ok, if and only if the $\texttt{null}_a$ address is not mapped to an object, the runtime types of all objects are well-formed, the root owner $\texttt{null}_a$ is in the set of owners of all objects, and all addresses stored in fields are well-typed (WFH). By mandating that all objects are (transitively) owned by $\texttt{null}_a$ and because each runtime type has one unique owner address, we ensure that ownership is a tree structure.

$$\text{RFT}\ \frac{\mathtt{h}(\iota)\downarrow_1 = \_\ \mathtt{C<\_>} \qquad \mathtt{C<\_>} \sqsubseteq \mathtt{C'<\_>}}{{}^{\mathtt{r}}fType(\mathtt{h},\iota,\mathtt{f}) = dyn({}^{\mathtt{s}}fType(\mathtt{C'},\mathtt{f}),\mathtt{h},\iota)}$$

$$\text{RMT-1}\ \frac{\mathtt{class\ C<\_>\ extends\ \_<\_>\ \{\ \_\ ;\ldots\ \overline{\mathtt{<X\ \_>}}\ \_\ \_\ \mathtt{m}(\overline{\mathtt{x\ \_}})\ \{\ \mathtt{return\ e}\ \}\ldots\ \}}}{mBody(\mathtt{C},\mathtt{m}) = (\mathtt{e},\overline{\mathtt{x}},\overline{\mathtt{X}})}$$

$$\text{RMT-2}\ \frac{\mathtt{class\ C<\_>\ extends\ C'<\_>\ \{\ no\ method\ m\ \}}}{mBody(\mathtt{C},\mathtt{m}) = mBody(\mathtt{C'},\mathtt{m})}$$

$$\text{WFRT}\ \frac{\iota' \in \mathrm{dom}(\mathtt{h}) \cup \{\mathtt{null}_a, \mathtt{any}_a\} \qquad \mathtt{h},\iota \vdash \overline{{}^{\mathtt{r}}\mathtt{T}}\ \mathtt{ok} \qquad \mathtt{class\ C<\_}\ \overline{{}^{\mathtt{s}}\mathtt{N}}\mathtt{>}\ \ldots \qquad \mathtt{h},\iota \vdash \overline{{}^{\mathtt{r}}\mathtt{T}} <: dyn(\overline{{}^{\mathtt{s}}\mathtt{N}},\mathtt{h},\iota)}{\mathtt{h},\iota \vdash \iota'\ \mathtt{C<}\overline{{}^{\mathtt{r}}\mathtt{T}}\mathtt{>\ ok}}$$

$$\text{WFH}\ \frac{\mathtt{null}_a \notin \mathrm{dom}(\mathtt{h}) \quad \forall \iota\colon\ \mathtt{h},\iota \vdash \mathtt{h}(\iota)\downarrow_1\ \mathtt{ok} \ \wedge\ \mathtt{null}_a \in owners(\mathtt{h},\iota) \qquad \forall \iota,\mathtt{f}\colon\ \mathtt{h}(\iota)\downarrow_2 = \mathtt{Fs} \ \wedge\ {}^{\mathtt{r}}fType(\mathtt{h},\iota,\mathtt{f}) = {}^{\mathtt{r}}\mathtt{T} \implies \mathtt{h} \vdash \mathtt{Fs(f)} : {}^{\mathtt{r}}\mathtt{T}}{\mathtt{h\ ok}}$$

$$\text{WFRE}\ \frac{\begin{array}{c}{}^{\mathtt{r}}\Gamma = \overline{\mathtt{X}\ {}^{\mathtt{r}}\mathtt{T}}\ ;\ \mathtt{this}\ \iota, \overline{\mathtt{x}\ \iota'} \qquad {}^{\mathtt{s}}\Gamma = \overline{\mathtt{X}\ {}^{\mathtt{s}}\mathtt{N}},\ \overline{\mathtt{X'}\ \_}\ ;\ \mathtt{this}\ (\mathtt{this}_u\ \mathtt{C<}\overline{\mathtt{X'}}\mathtt{>}), \overline{\mathtt{x}\ {}^{\mathtt{s}}\mathtt{T'}} \\ \mathtt{h\ ok} \quad {}^{\mathtt{s}}\Gamma\ \mathtt{ok} \quad \iota \neq \mathtt{null}_a \quad \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \overline{{}^{\mathtt{r}}\mathtt{T}}\ \mathtt{ok} \quad \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \overline{{}^{\mathtt{r}}\mathtt{T}} <: \overline{dyn(\overline{{}^{\mathtt{s}}\mathtt{N}},\mathtt{h},{}^{\mathtt{r}}\Gamma)} \\ \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \iota :\ \mathtt{this}_u\ \mathtt{C<}\overline{\mathtt{X'}}\mathtt{>} \qquad \mathtt{h},{}^{\mathtt{r}}\Gamma \vdash \overline{\iota'} :\ \overline{{}^{\mathtt{s}}\mathtt{T'}}\end{array}}{\mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma}$$

Figure 4.25: Rules for field and method lookup, and well-formedness

Well-Formed Runtime Environments. The judgment $\mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma$ expresses that runtime environment ${}^{\mathtt{r}}\Gamma$ is well-formed w.r.t. a well-formed heap $\mathtt{h}$ and a well-formed static type environment ${}^{\mathtt{s}}\Gamma$. According to rule WFRE, this is the case if and only if: (1) ${}^{\mathtt{r}}\Gamma$ maps all method type variables $\overline{\mathtt{X}}$ that are contained in ${}^{\mathtt{s}}\Gamma$ to well-formed runtime types $\overline{{}^{\mathtt{r}}\mathtt{T}}$, which are subtypes of the dynamizations of the corresponding upper bounds $\overline{{}^{\mathtt{s}}\mathtt{N}}$; (2) ${}^{\mathtt{r}}\Gamma$ maps `this` to an address ι. The object at address ι is well-typed with the static type of `this`, $\mathtt{this}_u$ $\mathtt{C<}\overline{\mathtt{X'}}\mathtt{>}$. (3) ${}^{\mathtt{r}}\Gamma$ maps the formal parameters $\overline{\mathtt{x}}$ that are contained in ${}^{\mathtt{s}}\Gamma$ to addresses $\overline{\iota'}$. The objects at addresses $\overline{\iota'}$ are well-typed with the static types of $\overline{\mathtt{x}}$, $\overline{{}^{\mathtt{s}}\mathtt{T'}}$.

Operational Semantics

We describe program execution by a big-step operational semantics. The transition $\mathtt{h},{}^{\mathtt{r}}\Gamma,\mathtt{e} \rightsquigarrow \mathtt{h'},\iota$ expresses that the evaluation of an expression `e` in heap `h` and runtime environment ${}^{\mathtt{r}}\Gamma$ results in address ι and successor heap $\mathtt{h'}$. A program with main class `C` is executed by evaluating the main expression in a heap $\mathtt{h}_0$ that contains exactly one `C` instance in the root context where all fields $\overline{\mathtt{f}}$ are initialized to $\mathtt{null}_a$ ($\mathtt{h}_0 = \{\iota \mapsto (\mathtt{null}_a\ \mathtt{C<>},\ \overline{\mathtt{f}\ \mathtt{null}_a})\}$) and a runtime environment ${}^{\mathtt{r}}\Gamma_0$ that maps `this` to this `C` instance (${}^{\mathtt{r}}\Gamma_0 = \epsilon\mathtt{;this}\iota$). The rules for evaluating expressions are presented in Fig. 4.26 and explained in the following.

Parameters, including `this`, are evaluated by looking up the stored address in the stack, which is part of the runtime environment ${}^{\mathtt{r}}\Gamma$ (OS-Var). The `null` expression always evaluates to the $\mathtt{null}_a$ address (OS-Null). For cast expressions, we evaluate the expression $\mathtt{e}_0$ and check that the resulting address is well-typed with the static type given in the cast expression w.r.t. the current environment (OS-Cast). Object creation picks a fresh address, allocates an object of the appropriate type, and initializes its fields to $\mathtt{null}_a$ (OS-New). *fields*(`C`) yields all fields declared in or inherited by `C`.

For field reads (OS-Read) we evaluate the receiver expression and then look up the field in the heap, provided that the receiver is non-null. For the update of a field `f`, we evaluate the receiver expression to address ι_0 and the right-hand side expression to address ι, and update the heap $\mathtt{h}_2$, which is denoted by $\mathtt{h}_2[\iota_0\mathtt{.f} := \iota]$ (OS-Upd). Note that the limited covariance of Generic Universe Types does not require a runtime ownership check for field updates.

For method calls (OS-Invk) we evaluate the receiver expression and actual method arguments in the usual order. The class of the receiver object is used to look up the method body. Its expression is then evaluated in the runtime environment that maps `m`'s type variables to actual type arguments as well as `m`'s formal method parameters (including `this`) to the actual method arguments. The resulting heap and address are the result of the call. Note that method invocations do not need any runtime type checks or

$$\text{OS-Var}\ \frac{}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{x} \rightsquigarrow \mathtt{h}, {}^{\mathtt{r}}\Gamma(\mathtt{x})} \qquad \text{OS-Null}\ \frac{}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{null} \rightsquigarrow \mathtt{h}, \mathtt{null}_a}$$

$$\text{OS-Cast}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota \quad \mathtt{h}', {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T}}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, ({}^{\mathtt{s}}\mathtt{T})\ \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota} \qquad \text{OS-New}\ \frac{\iota \notin \mathrm{dom}(h) \quad \iota \neq \mathtt{null}_a \quad {}^{\mathtt{r}}\mathtt{T} = \mathit{dyn}({}^{\mathtt{s}}\mathtt{N}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) = \_\ \mathtt{C}\texttt{<}\_\texttt{>} \quad \mathtt{Fs}(\mathit{fields}(\mathtt{C})) = \mathtt{null}_a \quad \mathtt{h}' = \mathtt{h}[\iota \mapsto ({}^{\mathtt{r}}\mathtt{T}, \mathtt{Fs})]}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{new}\ {}^{\mathtt{s}}\mathtt{N} \rightsquigarrow \mathtt{h}', \iota}$$

$$\text{OS-Read}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}', \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \iota = \mathtt{h}'(\iota_0)\downarrow_2(\mathtt{f})}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{f} \rightsquigarrow \mathtt{h}', \iota} \qquad \text{OS-Upd}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \mathtt{h}_0, {}^{\mathtt{r}}\Gamma, \mathtt{e}_2 \rightsquigarrow \mathtt{h}_2, \iota \quad \mathtt{h}' = \mathtt{h}_2[\iota_0.\mathtt{f} := \iota]}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{f}{=}\mathtt{e}_2 \rightsquigarrow \mathtt{h}', \iota}$$

$$\text{OS-Invk}\ \frac{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0 \rightsquigarrow \mathtt{h}_0, \iota_0 \quad \iota_0 \neq \mathtt{null}_a \quad \mathtt{h}_0, {}^{\mathtt{r}}\Gamma, \overline{\mathtt{e}_2} \rightsquigarrow \mathtt{h}_2, \overline{\iota_2} \quad \mathtt{h}_0(\iota_0)\downarrow_1 = \_\ \mathtt{C}_0\texttt{<}\_\texttt{>} \quad \mathit{mBody}(\mathtt{C}_0, \mathtt{m}) = (\mathtt{e}_1, \overline{\mathtt{x}}, \overline{\mathtt{X}}) \quad \overline{{}^{\mathtt{r}}\mathtt{T}} = \mathit{dyn}(\overline{{}^{\mathtt{s}}\mathtt{T}}, \mathtt{h}, {}^{\mathtt{r}}\Gamma) \quad {}^{\mathtt{r}}\Gamma' = \overline{\mathtt{X}\ {}^{\mathtt{r}}\mathtt{T}}\ ;\ \mathtt{this}\ \iota_0, \overline{\mathtt{x}\ \iota_2} \quad \mathtt{h}_2, {}^{\mathtt{r}}\Gamma', \mathtt{e}_1 \rightsquigarrow \mathtt{h}', \iota}{\mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e}_0.\mathtt{m}\texttt{<}\overline{{}^{\mathtt{s}}\mathtt{T}}\texttt{>}(\overline{\mathtt{e}_2}) \rightsquigarrow \mathtt{h}', \iota}$$

Figure 4.26: Operational semantics

purity checks.

4.3.4 Properties

In this section, we present the theorems and proof sketches for type safety and the owner-as-modifier property as well as two important auxiliary lemmas.

Lemmas. The following lemma expresses that viewpoint adaptation from a viewpoint to `this` is correct. Consider the `this` object of a runtime environment ${}^{\mathtt{r}}\Gamma$ and two objects $\mathtt{o}_1$ and $\mathtt{o}_2$. If from the viewpoint `this`, $\mathtt{o}_1$ has the static type ${}^{\mathtt{s}}\mathtt{N}$, and from viewpoint $\mathtt{o}_1$, $\mathtt{o}_2$ has the static type ${}^{\mathtt{s}}\mathtt{T}$, then from the viewpoint `this`, $\mathtt{o}_2$ has the static type ${}^{\mathtt{s}}\mathtt{T}$ adapted from ${}^{\mathtt{s}}\mathtt{N}$, ${}^{\mathtt{s}}\mathtt{N}{\triangleright}{}^{\mathtt{s}}\mathtt{T}$. The following lemma expresses this property using the addresses ι_1 and ι_2 of the objects $\mathtt{o}_1$ and $\mathtt{o}_2$, respectively.

Lemma 4.3.4.1 (Adaptation from a Viewpoint)**.**

$$\left.\begin{array}{l} \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_1 : {}^{\mathtt{s}}\mathtt{N}, \quad \iota_1 \neq \mathtt{null}_a \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma' \vdash \iota_2 : {}^{\mathtt{s}}\mathtt{T} \\ \mathit{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \mathit{dom}({}^{\mathtt{s}}\mathtt{N}) \circ \overline{\mathtt{X}} \\ {}^{\mathtt{r}}\Gamma' = \overline{\mathtt{X}}\ \mathit{dyn}(\overline{{}^{\mathtt{s}}\mathtt{T}}, \mathtt{h}, {}^{\mathtt{r}}\Gamma);\ \mathtt{this}\ \iota_1, \_ \end{array}\right\} \implies \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_2 : ({}^{\mathtt{s}}\mathtt{N}{\triangleright}{}^{\mathtt{s}}\mathtt{T})[\overline{{}^{\mathtt{s}}\mathtt{T}/\mathtt{X}}]$$

This lemma justifies the type rule GT-Read. The proof runs by induction on the shape of static type ${}^{\mathtt{s}}\mathtt{T}$. The base case deals with type variables and non-generic types. The induction step considers generic types, assuming that the lemma holds for the actual type arguments. Each of the cases is done by a case distinction on the main modifiers of ${}^{\mathtt{s}}\mathtt{N}$ and ${}^{\mathtt{s}}\mathtt{T}$.

The following lemma is the converse of Lemma 4.3.4.1. It expresses that viewpoint adaptation from `this` to an object $\mathtt{o}_1$ is correct. If from the viewpoint `this`, $\mathtt{o}_1$ has the static type ${}^{\mathtt{s}}\mathtt{N}$ and $\mathtt{o}_2$ has the static type ${}^{\mathtt{s}}\mathtt{N}{\triangleright}{}^{\mathtt{s}}\mathtt{T}$, then from viewpoint $\mathtt{o}_1$, $\mathtt{o}_2$ has the static type ${}^{\mathtt{s}}\mathtt{T}$. The lemma requires that the adaptation of ${}^{\mathtt{s}}\mathtt{T}$ does not change ownership modifiers in ${}^{\mathtt{s}}\mathtt{T}$ from non-`any` to `any`, because the lost ownership information cannot be recovered. Such a change occurs if ${}^{\mathtt{s}}\mathtt{N}$'s main modifier is `any` or if ${}^{\mathtt{s}}\mathtt{T}$ contains `rep` and is not accessed through `this` (see definition of *rp*, Sec. 4.3.2).

Lemma 4.3.4.2 (Adaptation to a Viewpoint)**.**

$$\left.\begin{array}{l} \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_1 : {}^{\mathtt{s}}\mathtt{N}, \quad \iota_1 \neq \mathtt{null}_a \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma \vdash \iota_2 : ({}^{\mathtt{s}}\mathtt{N}{\triangleright}{}^{\mathtt{s}}\mathtt{T})[\overline{{}^{\mathtt{s}}\mathtt{T}/\mathtt{X}}] \\ {}^{\mathtt{s}}\mathtt{N} = \mathtt{u}\ \_\texttt{<}\_\texttt{>}, \quad \mathtt{u} \neq \mathtt{any}, \quad \mathit{rp}(\mathtt{u}, {}^{\mathtt{s}}\mathtt{T}) \\ \mathit{free}({}^{\mathtt{s}}\mathtt{T}) \subseteq \mathit{dom}({}^{\mathtt{s}}\mathtt{N}) \circ \overline{\mathtt{X}}, \quad {}^{\mathtt{s}}\mathtt{T} \neq \mathtt{this}_u\ \_\texttt{<}\_\texttt{>} \\ {}^{\mathtt{r}}\Gamma' = \overline{\mathtt{X}}\ \mathit{dyn}(\overline{{}^{\mathtt{s}}\mathtt{T}}, \mathtt{h}, {}^{\mathtt{r}}\Gamma);\ \mathtt{this}\ \iota_1, \_ \end{array}\right\} \implies \mathtt{h}, {}^{\mathtt{r}}\Gamma' \vdash \iota_2 : {}^{\mathtt{s}}\mathtt{T}$$

This lemma justifies the type rule GT-Upd and the requirements for the types of the parameters in GT-Invk. The proof is analogous to the proof for Lemma 4.3.4.1.

Type Safety for Generic Universe Types is expressed by the following theorem. If a well-typed expression `e` is evaluated in a well-formed environment (including a well-formed heap), then the resulting environment is well-formed and the result of `e`'s evaluation has the type that is the dynamization of `e`'s static type.

Theorem 4.3.4.3 (Type Safety)**.**

$$\left.\begin{array}{l} \mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ {}^{\mathtt{s}}\Gamma \vdash \mathtt{e} : {}^{\mathtt{s}}\mathtt{T} \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e} \rightsquigarrow \mathtt{h}', \iota \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \mathtt{h}' \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ \mathtt{h}', {}^{\mathtt{r}}\Gamma \vdash \iota : {}^{\mathtt{s}}\mathtt{T} \end{array}\right.$$

The proof of Theorem 4.3.4.3 runs by rule induction on the operational semantics. Lemma 4.3.4.1 is used to prove field read and method results, whereas Lemma 4.3.4.2 is used to prove field updates and method parameter passing.

We omit a proof of progress since this property is not affected by adding ownership to a Java-like language. The basic proof can be adapted from FGJ [97] and extensions for field updates and casts. The new runtime ownership check in casts can be treated analogously to standard Java casts.

Owner-as-Modifier discipline enforcement is expressed by the following theorem. The evaluation of a well-typed expression `e` in a well-formed environment modifies only those objects that are (transitively) owned by the owner of `this`.

Theorem 4.3.4.4 (Owner-as-Modifier)**.**

$$\left.\begin{array}{l} \mathtt{h} \vdash {}^{\mathtt{r}}\Gamma : {}^{\mathtt{s}}\Gamma \\ {}^{\mathtt{s}}\Gamma \vdash \mathtt{e} : {}^{\mathtt{s}}\mathtt{T} \\ \mathtt{h}, {}^{\mathtt{r}}\Gamma, \mathtt{e} \rightsquigarrow \mathtt{h}', \_ \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \forall \iota \in \mathit{dom}(\mathtt{h}), \mathtt{f} : \\ \quad \mathtt{h}(\iota)\!\downarrow_2(\mathtt{f}) = \mathtt{h}'(\iota)\!\downarrow_2(\mathtt{f}) \vee \\ \quad \mathit{owner}(\mathtt{h}, {}^{\mathtt{r}}\Gamma(\mathtt{this})) \in \mathit{owners}(\mathtt{h}, \iota) \end{array}\right.$$

where $\mathit{owner}(\mathtt{h}, \iota)$ *denotes the direct owner of the object at address* ι *in heap* `h`*, and* $\mathit{owners}(\mathtt{h}, \iota)$ *denotes the set of all (transitive) owners of this object.*

The proof of Theorem 4.3.4.4 runs by rule induction on the operational semantics. The interesting cases are field update and calls of non-pure methods. In both cases, the type rules (Fig. 4.22) enforce that the receiver expression does not have the main modifier `any`. That is, the receiver object is owned by `this` or the owner of `this`. For the proof we assume that pure methods do not modify objects that exist in the pre-state of the call. In this section we do not describe how this is enforced in the program. A simple but conservative approach forbids all object creations, field updates, and calls of non-pure methods [127]. The above definition also allows weaker forms of purity that permit object creations [71] and also approaches that allow the modification of newly created objects [160].

4.3.5 Conclusions

We presented Generic Universe Types, an ownership type system for Java-like languages with generic types. Our type system permits arbitrary references through `any` types, but controls modifications of objects, that is, enforces the owner-as-modifier discipline. This allows us to handle interesting implementations beyond simple aggregate objects, for instance, shared buffers [71]. We show how `any` types and generics can be combined in a type safe way using limited covariance and viewpoint adaptation.

Generic Universe Types require little annotation overhead for programmers. As we have shown for non-generic Universe Types [71], this overhead can be further reduced by appropriate defaults. The default ownership modifier is generally `peer`, but the modifier of upper bounds, exceptions, and immutable types (such as `String`) defaults to `any`. These defaults make the conversion from Java 5 to Generic Universe Types simple.

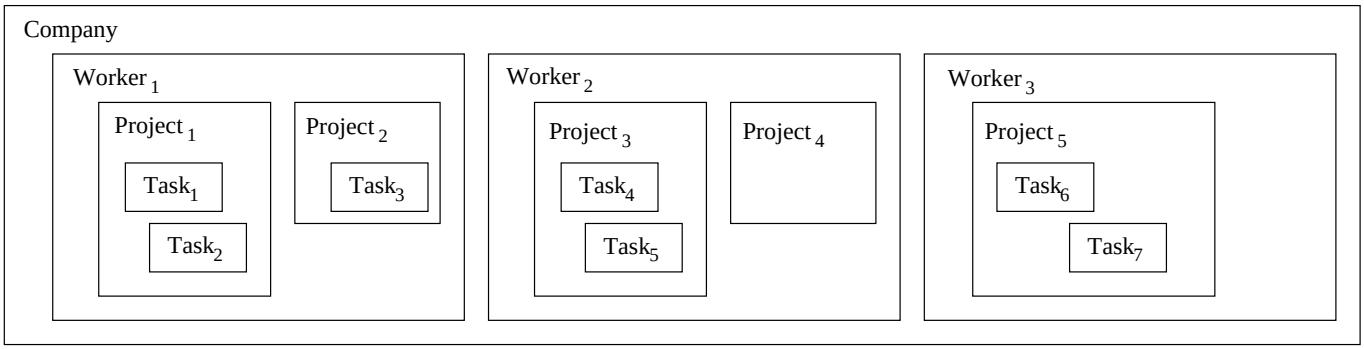

Figure 4.27: A Single Ownership Structure. Three `Worker`s belong to the Company, each `Worker` is working on several `Project`s, and each `Project` has `Task`s the `Worker` must complete.

The type checker and runtime support for Generic Universe Types are implemented in the JML tool suite [130].

As future work, we plan to use Generic Universe Types for program verification, extending our earlier work [127, 130]. We are also working on path-dependent Universe Types to support more fine-grained information about object ownership, and to extend our inference tools for non-generic Universe Types to Generic Universe Types.

4.4 Multiple Ownership

In ownership systems, each object has one owner and the ownership relation forms a tree. While different versions of ownership have proved effective for a variety of tasks [12, 172, 50, 58, 68], empirical studies have shown that this ownership structure does not suit all programs [34, 122, 143]. In this section we present an ownership type system that removes this restriction, so that an object may have multiple owners, and the ownership relation forms a DAG. We make the following contributions:

- the *objects in boxes* model, a simple, straightforward model of object ownership based on sets of objects, which describes the fundamental features of single ownership, and generalises smoothly to multiple ownership.

- a language design incorporating **M**ultiple **O**wnership into a **J**ava-like language with **O**bjects (MOJO). MOJO's novel constructs include multiple ownership types, constraint declarations to indicate that two boxes either intersect or are disjoint, and a restricted form of existential ownership. Thus, existing ownership type systems can support multiple ownership via relatively small extensions.

- a formal definition for MOJO, including a type system which we have proved sound.

- an effects system for MOJO that works with multiple ownership, that again, we have proved sound.

We have developed MOJO as an extension of traditional ownership types systems instead of an extension of Universe types, because MOJO requires explicit ownership parameters, and at that time GUJ was not that well understood. We now plan to extend GUJ accordingly.

Outline The next section informally introduces our conceptual model of ownership, the language MOJO, and the effects system. We then give a formal presentation of the syntax, operational semantics, type and effects system of MOJO, and its soundness. The section concludes with a discussion of MOJO idioms and extensions and a brief survey of related work.

4.4.1 The Benefits of Putting Objects into Boxes

In this section we present our conceptual model — the "*objects in boxes*" model — of multiple ownership and effects in object-oriented systems. We begin by modelling single ownership, then show how the objects in boxes model generalises to multiple owners. Interleaved with the conceptual presentation, we show how these models can be described using ownership types in programming languages. Our examples are expressed in our core language MOJO but would apply in most languages with ownership types.

Upon reflection, given that ownership has been studied for at least ten years [139], and alias control for fifteen [90], it seems odd that only now we are presenting something as naïve as a model based purely on sets. Compared with previous work, the objects in boxes model focuses on ownership sets (boxes), the objects in the boxes, and the effects of computation, and abstracts away from language constructs, types, messages, capabilities, and especially the pointer structures that feature prominently in most other treatments of object ownership [12, 56, 58, 127]. While some of these concerns must be reintroduced as we move from a conceptual model to a programming language, we have found the abstraction offered by the objects in boxes model to be very useful in designing and reasoning about ownership, and multiple ownership in particular.

Single Ownership

The object structure from Figure 4.27 shows a company that carries out a range of different `Project`s. Each `Project` has one or more `Worker`s allocated to it, and each `Worker` has one or more `Task`s they need to complete.

The key relationship this diagram brings out is *object ownership*: each `Task` is owned by a `Project`, and each `Project` in turn is owned by the `Worker` responsible for it. Ownership models abstraction, encapsulation and aggregation: `Task`s are *part of* their `Project`s; `Worker`s are part of the Company they work for. A change to one of the parts — say a `Project` being cancelled — necessarily affects the whole abstraction in which that part is contained. Similarly, a change to a whole – say a `Worker` going on leave — may change any of its subparts — perhaps delaying all of the `Task`s comprising its `Project`.

Partitioning objects is key to ownership systems, whether they use types [58] or specifications [128]. Different systems have chosen different names for these partitions: islands [90], balloons [13], domains [12], contexts [56], regions [83] — with each name being associated with a particular detailed proposal.

We propose the neutral term **boxes** to describe these partitions: in a sense, every ownership system "puts objects into boxes" and differs in the details of those boxes. Figure 4.27 also gives a hint to the most fundamental semantics of these boxes: *a box is a set of objects*. So, for example, we could write $[\![\,\texttt{Worker}_2\,]\!]$ to mean all the objects contained within $\texttt{Worker}_2$'s box. Here we have:

$$\begin{array}{rcl} [\![\,\texttt{Project}_2\,]\!] & = & \{\texttt{Task}_3\} \\ [\![\,\texttt{Project}_3\,]\!] & = & \{\texttt{Task}_4, \texttt{Task}_5\} \end{array}$$

The first consequence of this model is that diagrams such as Figure 4.27 (which have adorned almost every ownership paper ever published) can now be ascribed clear semantics: they are just the diagrams of sets we are familiar with from primary school.

The second consequence of this model is that semantics of object composition — box nesting — follows naturally. So, for example, reading from the diagram:

$$\begin{array}{rcl} [\![\,\texttt{Worker}_1\,]\!] & = & \{\texttt{Project}_1, \texttt{Project}_2, \texttt{Task}_1, \\ & & \texttt{Task}_2, \texttt{Task}_3\} \\ [\![\,\texttt{Worker}_3\,]\!] & = & \{\texttt{Project}_5, \texttt{Task}_6, \texttt{Task}_7\} \end{array}$$

Given that an object o is given a box $[\![\,o\,]\!]$ upon its creation, we can establish the invariant that if x is inside its owner o, written $x \ll o$, then its box must be inside its owner's box. More generally:

$$x \ll o \Leftrightarrow x \in [\![\,o\,]\!] \qquad \textsc{Objects in Boxes}$$

An object's box must be a subset of its owner's box:

$$x \ll o \Rightarrow [\![x]\!] \subseteq [\![o]\!] \qquad \text{Box Nesting}$$

And, in single ownership, the inside relation is a tree:

$$[\![o_1]\!] \cap [\![o_2]\!] \neq \emptyset \Rightarrow [\![o_1]\!] \subseteq [\![o_2]\!] \vee [\![o_2]\!] \subseteq [\![o_1]\!] \qquad \text{Single owners}$$

These invariants should hold however we model heaps, and also independently of whether objects are permitted to change owner — type systems generally do not support ownership change; specification languages do.

Single Ownership Languages In an ownership-aware programming or specification language we could define these classes as follows. First, the `Task` class contains two fields — straightforward value types giving the tasks's name and duration: the single method delays a task by increasing its duration.

```
1 class Task<o> {
2     String name;
3     int time;
4     void delay() { time++; }
5 }
```

The `Task` class also has the *ownership parameter* `o` that is a special form of type parameter (a phantom type) that records ownership information. The `Task` class needs to be ownership parametric, because different tasks will have different owners (e.g. in Figure 4.27, $\texttt{Task}_1$ is owned by $\texttt{Project}_1$ while $\texttt{Task}_4$ is owned by $\texttt{Project}_3$). Ownership parameters connect compile-time static types to run-time dynamic boxes. An object's owner parameter in its type represents the box it is inside:

$$x : \texttt{C<o>} \Rightarrow x \in [\![\texttt{o}]\!] \qquad \text{Owners as Boxes}$$

In ownership type languages, actual ownership parameters may be the formal parameters of the enclosing class (including the distinguished first parameter representing an instance's owner); "`this`" establishing that the current "`this`" instance is the owner of the new type; or final fields, establishing that the object contained in the field is the owner.

The `Project` class is also ownership parametric. `Project`s delay themselves by delaying every constituent task.

```
1 class Project<o> {
2     TaskList<this,this> tasks;
3     void delay() {
4         for(var t : tasks) { t.delay(); }
5     }
6 }
```

The field `tasks` stores a list of the project's tasks, and is declared as `TaskList<this,this>`. That means that the list of tasks pointed to by this field, and each `Task` stored in the List, will be owned by *this particular project instance*, and therefore will be inside the box belonging to this `Project` instance, a member of the set $[\![\texttt{this}]\!]$, which will be different for each different project. The box nesting invariant ensures that an object's box is inside its owner. That is, $\texttt{this} \ll \texttt{o}$, and thus $[\![\texttt{this}]\!] \subseteq [\![\texttt{o}]\!]$.

The `Worker` class is quite straightforward, keeping a list of `Project`s owned by this `Project` (i.e. inside its box) and delaying itself by delaying those projects.

```
class Worker<o> {
    ProjectList <this,this> projects;
    void delay() {
        for(var p : projects) { p.delay(); }
    }
}
```

Consider now the `TaskList` class (the `ProjectList` class is similar) whose instances we omitted in Figure 4.27 for space reasons. Its implementation is rudimentary, as we're mainly concerned with ownership types involved:

```
class TaskList<o, tO> {
    Task<tO> t;
    TaskList<o,tO> next;
    TaskList<o,tO> prev;

    void add(Task<tO> tt) {
        if (next==nil) {
            next = new TaskList<o, tO>();
            next.t = tt;
            next.prev = this;
        }
        else {
            next.add(i);
        }
    }
    Task<tO> get(int i) {
        return (i==0) ? item : next.get(i−1);
    }
}
```

`TaskList` has two ownership parameters. The first, `o`, is the "primary" owner parameter, just as in the other classes we've seen. The second, `tO`, is the ownership of the `Task` stored in each list node. In this way the ownership of the node and its contents do not have to be the same. The fields `next` and `prev` have type `TaskList<o, tO>` saying that the adjacent list entries have the same item ownership as this list entry, and the same owner as this object: all entries in a single list will be members of the *same* enclosing box; as will all the tasks — although they may be in different boxes. This differs from the fields in classes `Project` and `Worker`, which have `this` ownership, meaning that they belong to the box owned by the current object itself.

Effects within Single Ownership Ownership can help determine the *effects* of a computation in terms of the objects read or written. Two computations do not *interfere* (they do not write the same objects, or do not read objects the other writes) if the intersection of the boxes involved is empty.

Effects systems [57, 83] annotate methods with effects specifications, describing the boxes read or written. In `Task`, the fields `name` and `time` hold simple types, are local to the object, and can only be changed by the object itself. The `delay` method makes just such an assignment to `time`. The effects of, say, reading the `name` field would be `this / empty` meaning reading the "`this`" object and not writing anything. The effects of the `delay` method would be `this / this` — reading and writing the object to which the method is sent.

```
class Task<o> { ...
    void delay()   //effect : this/this
    ...
}
```

On the other hand, the `Project`'s `delay` method.

```
class Project<o> {
```

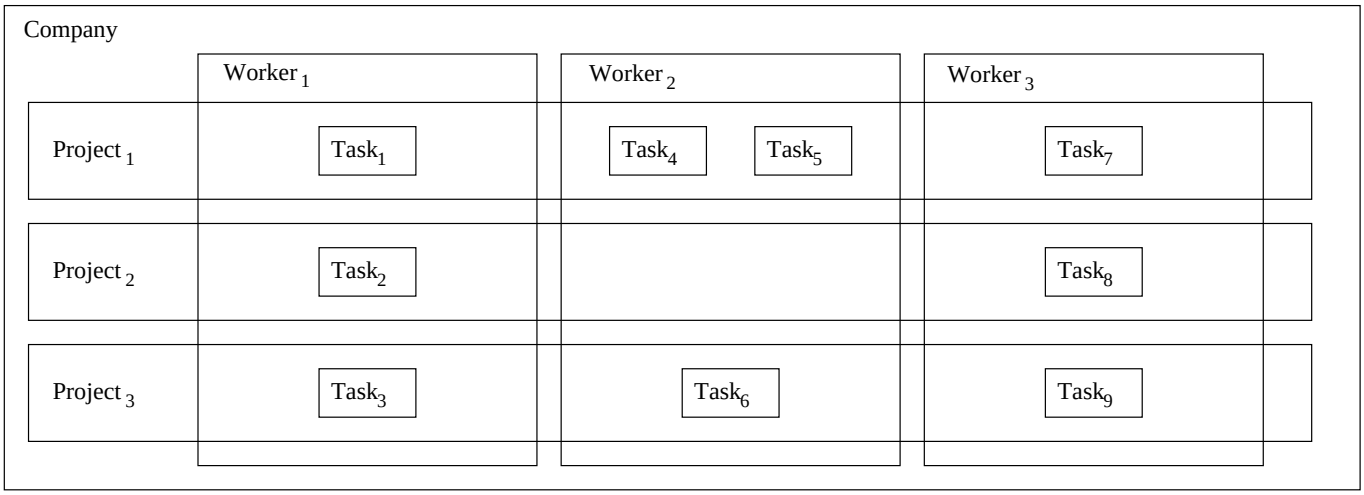

Figure 4.28: A Multiple Ownership Structure. The Company now requires its `Workers` to work on many different `Projects`— and different `Tasks` in a project can be carried out by different `Workers`.

```
2       TaskList<this,this> tasks;
3       void delay()      //effect: this/this
4       { for(var t : tasks) { t.delay(); } }
5   }
```

reads the `tasks` variable, the fields of those subordinate `Task` objects, and calls `delay` on them. From the effects of `delay()`, (reads `this`, writes `this`) we know that it will write whatever object it is called upon. The question is: which `Task` objects will be written?

Effects systems without ownership [83, 107] cannot easily distinguish *which* Task may be affected; effects like "`all.Task / all.Task`" say that `delay` on any project may read and write *any* `Task`. The upshot of this is that delaying any project must be assumed to delay *every other* project in the system.

This is precisely where boxes come to the rescue. Looking again at Figure 4.27, only the `Tasks` in the `Project`'s box are written. The type of these tasks, *i.e.* `Task <this>` gives that information. We interpret effects so that they apply to boxes, rather than objects: effects such as `.../this` means that a computation may write the "`this`" object itself, or any other object in its box ⟦`this`⟧. The effects for `Project`'s `delay` method are `this / this`, so the method may read or write the object itself or any other object that it owns, but may not read or write any object outside its own box. The `Worker`'s `delay` method also has effects `this / this`.

Multiple Ownership

Single ownership requires every object to have a *single* direct owner, thus the ownership structure is a tree. While easy to understand, easy to model, and (relatively) easy to formalise and enforce, single ownership is too restrictive for many kinds of programs. Empirical studies have shown that relationships between objects and between the classes that define them are scale free networks — tangled graphs where every object is only a few hops from every other object [122, 143]. Non-hierarchical relationships cannot be modelled by trees.

For example, imagine the following change to the `Projects`, `Workers`, and `Tasks` model, given in Figure 4.28. Here, the company has been restructured from a hierarchical style, where every project is carried out by just one worker, into a "matrix" management style where every task is assigned to *both* a project and a worker. As a result, tasks now somehow have to belong to both projects and workers; delaying a project must delay all employees who must work on tasks on that project, and similarly delaying an employee will delay all projects with which they are involved.

The topology in Figure 4.28 cannot be described with existing ownership type systems. Classical own-

ership enforces a very strong owners-as-dominators policy over pointers — all paths to an object must be via its owner — so if programmers attempt to write programs describing this interconnected ownership structure, their programs will be rejected as type-incorrect. Other systems support owners-as-modifiers or effective ownership, rather than pointer control [127]; so they would at least be able to pick one of either `Project`s or `Worker`s as a primary axis of organisation — say `Project`s— and grant permission to `Worker`s to have pointers into tasks even though they belonged to projects. Unfortunately, when a `Worker` is delayed in such a system, it would not have permission to *modify* its `Project` objects because it does not own them.

In single ownership systems, programmers get around these restrictions by "flattening" or "lifting" the ownership hierarchy: rather than nesting boxes, every task, worker, and project can exist in one very large company box, and use "peer" ownership — types like `Task<o>` that refer to other objects in the same box as `this`, rather than `this`'s box — to access every required object directly. In both owners-as-dominators and owners-as-modifiers systems, this would typecheck and allow e.g. projects and workers to update their tasks: there is no longer one primary "dominant" decomposition. The problem with this design is that it loses any benefit of ownership types: with everything in one large box, we cannot distinguish between tasks belonging to one project, or another project, or a worker. Once again, a change to one task will be taken as a change to all tasks.

This is where the interpretation of Figure 4.28, modelling boxes as sets, shows us the way out. Just as an element can be a member of *more than one set*, an object can be *inside more than one box*, that is, be owned by more than one object. Where two boxes overlap, objects in their intersection are within both boxes, and so have multiple owners. So all the `Task`s belonging to $\texttt{Project}_1$, say, will still be inside $\texttt{Project}_1$'s box — $[\![\,\texttt{Project}_1\,]\!]$. Similarly, `Task`s belonging to $\texttt{Worker}_2$ will be inside $[\![\,\texttt{Worker}_2\,]\!]$. And, crucially, `Task`s (or any other object) belonging to both $\texttt{Project}_1$ and $\texttt{Worker}_2$ (for example, $\texttt{Task}_2$ in Figure 4.28) will reside in *both* boxes, that is, in the intersection of the two sets: $[\![\,\texttt{Project}_1\,]\!] \cap [\![\,\texttt{Worker}_2\,]\!]$. This semantics follows directly from interpreting Figure 4.28 as a set diagram.

The set interpretation generalises equally well to effects in a multiple ownership setting. Read or write effects upon an object with multiple owners must be taken to be effects within the intersection of the boxes to which that object belongs — and if this intersection itself intersects the effects of another computation, then those two computations potentially interfere.

MOJO: Language Support for Multiple Ownership We generalise a single-owner language to support multiple owners. Our core language, MOJO, is a relatively simple extension to existing single-ownership languages such as JOE and OGJ [12, 57, 142]. In the rest of this section we present the various new features of MOJO based on the multiple-ownership version of task management.

First, we reconsider the `Task` class. Surprisingly, this is *exactly the same as the single owner version.* In particular, `Task` retains just one ownership parameter even though in the design — Figure 4.28 — every `Task` has multiple owners. In MOJO, multiple owners are supplied upon class instantiation, rather than upon declaration; therefore classes can be parametric in the number of owners they will have[10].

To instantiate objects with multiple owners, MOJO supports a special ownership combinator that provides multiple (intersection) ownership. The actual ownership argument `a&b` describes multiple owners `a` and `b`: a *single formal* argument is bound by *multiple actual* arguments (just as in a type-generic system, `List<T>` instantiated by `Pair<A,B>` gives `List<Pair<A,B>>`, the formal argument `T` is instantiated with the pair type `Pair<A,B>`). For example, we can declare a `Task` object that will be owned by a `Worker` and a `Project` object, both previously created:

```
1 final Project<this> prj = new Project<this>();
2 final Worker<this> wrk = new Worker<this>();
3
4 Task<prj & wrk> tsk = new Task<prj & wrk>();
```

The interpretation of `a&b` follows clearly from the Objects in Boxes constraint. If an object x's owner

[10]This is an innovation of the current work; in our earlier work multiple class owners were provided upon class declaration.

is `a&b` then we can assume that there will exist a and b, and x be inside both of them:

$$x \ll a \wedge x \ll b \Rightarrow x \in [\![a]\!] \cap [\![b]\!]$$

or via the Owners as Boxes constraint:

$$x : \texttt{C<a\&b>} \Rightarrow x \in [\![\texttt{a}]\!] \cap [\![\texttt{b}]\!]$$

The only single ownership constraint that does *not* hold for multiple ownership is the Single Ownership constraint itself: multiple ownership is a non-empty intersection of two (or more) boxes. Thus

$$\begin{array}{c} [\![o_1]\!] \cap [\![o_2]\!] \neq \emptyset \\ \not\Rightarrow \\ [\![o_1]\!] \subseteq [\![o_2]\!] \vee [\![o_2]\!] \subseteq [\![o_1]\!] \end{array} \qquad \text{Multiple owners}$$

Returning to the example, the `TaskList` is also the same as the single owner version. Then, the *code* for the `Project` class is mostly unchanged,

```
1  class Project<o> {
2      TaskList<this, this & ?> tasks;
3      void delay() {
4          for(var t : tasks) { t.delay(); }
5      }
6      void add(Task<this & ?> t) {
7          tasks.add(t);
8      }
9  }
```

except for the ownership type used to declare `tasks`. In particular, the second ownership argument of `TaskList` (the one representing the owners of the tasks) is now `this&?`, which says three things: First, that the `Task`s are inside more than one box — they have multiple ownership. Second, that one of those owners is `this`: the current `Project` object. And finally, that — at this point in the program — we do now know what the other owner(s) of each `Task` are. Again, this is true of the structure in Figure 4.28: each task in a project can have a different worker assigned to it, and vice versa — every worker can work on different projects.

The `?` wildcard (similar to Java's "?" wildcard for generics) can be thought of as an *existential owner*, and is very similar to the anyowner from Universe types. Such wildcards also appear in [108, 177], or i Wildcard owners are crucial in a multiple ownership system because one owner often does not, or cannot, know the other potential owners. A `this` owner is a prime example, since it is only (directly) accessible via the "`this`" object. Here, the `Project` class knows that it is one of the owners of the `Task` but not what the other owners are — furthermore, each `Task` in the list may have a different `Worker` as its other owner. This is precisely the interpretation of `this&?`.

The `Worker` class is now symmetrical to the `Project` class: again it has a list of `Task`s, which are owned by this worker and by something else.

```
1  class Worker<o> {
2      TaskList<this, this & ?> tasks;
3      void delay() {
4          for(var t : tasks) { t.delay(); }
5      }
6      void add(Task<this & ?> t) {
7          tasks.add(t);
8      }
9  }
```

`Task<p1&w1>` is a subtype of `Task<p1&?>` and of `Task<w1&?>`. This allows to add tasks owned by, say, project `p1` and worker `w1` to both `p1` and `w1` as in the following code (we discuss the meaning of `intersects` in the following section):

P	::=	$class^*$	program
$class$	::=	`class` $c < \overline{p} >$ $\overline{pCnstr}$ $\lhd$ $c' < \overline{Q} >$ { $\overline{finfld}$ $\overline{fCnstr}$ $\overline{fld}$ $\overline{mth}$ }	class definition
C	::=	$\Join \mid \lozenge$	interesects or disjoints
$pCnstr$	::=	$p\ C\ p$	parameter constraints
$finfld$	::=	`final` t ff	final field definition
$fCnstr$	::=	$ff \mathrm{C} ff \mid ff \mathrm{C} p$	field constraints
fld	::=	$t\ f$	field definition
mth	::=	$t\ m\ (t\ x)\ \{\ e\ \}$	method body
t	::=	$c < \overline{Q} >$	static type
Q	::=	$q \mid q \cap Q$	actual own. param. (poss. multiple)
q	::=	$p \mid$ `this` $\mid ff \mid x \mid$ `?` $\mid \iota$	one actual ownership parameter
R	::=	$r \mid r \cap R$	runtime actual ownership parameters
r	::=	$\iota \mid$ `?`	one runtime ownership parameter
e	::=	$x \mid$ `this` $\mid e.f \mid e.f{=}e$	
	$\mid$	`new` $t \mid e.m(e) \mid \iota$	expressions
c, p	::=	id	class identif., form. ownership param.
f, m	::=	id	field identif., method identif.

Figure 4.29: Syntax, runtime entitites in grey

```
1  final Project<this> p1 = new Project<this>();
2  final Worker<this> w1 = new Worker<this>();
3  // w1 intersects p1
4
5  Task<p1 & w1> t1 = new Task<p1 & w1>();
6  p1.add(t1); w1.add(t1);
```

Effects within Multiple Ownership Given the straightforward extension from single to multiple ownership promised by the objects in boxes model, it is tempting to expect that effects would generalise similarly: unfortunately, that is not the case.

In the following example we create two tasks, one shared between project `p1` and worker `w1`, the other shared between `p2` and worker `w1`:

```
1  class Test {
2      final Project<this> p1 = new Project<this>();
3      final Project<this> p2 = new Project<this>();
4      final Worker<this> w1 = new Worker<this>();
5      // w1 intersects p2; w1 intersects p1
6
7      Task<p1 & w1> t1 = new Task<p1 & w1>();
8      p1.add(t1); w1.add(t1);
9
10     Task<p2 & w1> t2 = new Task<p2 & w1>();
11     p2.add(t2); w1.add(t2);
12 }
```

In this program `p1.delay()` and `w1.delay()` potentially interfere. Given our intuition from Figure 4.28, we expect `p1.delay()` and `p2.delay()` not to interfere. However, the expressions have effects:

```
p1.delay()  :  p1&? / p1&?
p2.delay()  :  p2&? / p2&?
w1.delay()  :  w1&? / w1&?
```

and we have insufficient information to distinguish the relationship between `p1` and `p2` from that between `p1` and `w1`.

Intersection and Disjointness To solve this problem we have to provide more information about which boxes intersect, and which boxes are disjoint. Instantiating types with multiple owners like `p1&w1` creates objects in the set intersection $[\![\texttt{p1}]\!] \cap [\![\texttt{w1}]\!]$, which means that the `p1` box and the `w1` box *must* intersect. Conversely, for disjoint boxes `p1` and `p2` (like in the figure) the multiple owner `p1&p2` is illegal.

We introduce two declarations that make box topologies explicit. In the example, we'd need to declare `w1 intersects p1` and `w1 intersects p2` if we want to have workers whose tasks are in both `p1` and `p2`. Similarly, we need to declare `p1 disjoint p2` to ensure the `p1` and `p2` boxes are independent. Only one relationship (intersects or disjoint) may be declared between any two boxes: if no relationship is declared, then we don't know what the topology is and we make conservative assumptions.

Then, multiple ownership like `a&b` is legal only if it can be shown that `a` and `b` are legal, and that `a intersects b`. In our example, `p1&w1` and `w1&p1` and `p2&w1` are all legal ("`&`", `intersects` and `disjoint` are symmetric; `intersects` and "`&`" are reflexive; `disjoint` is irreflexive) while `p1&p2` is **not legal** because `p1` and `p2` are not declared as intersecting.

Effects are independent when we can show that their boxes will be disjoint. For effects involving multiple owners (like `p2&w1`) it is enough to consider owners pairwise, and to find `one` pair that is definitely disjoint: in the example, `p1` and `p2` are declared to be disjoint, so their intersection is empty, *i.e.* $[\![\texttt{p1}]\!] \cap [\![\texttt{p2}]\!] = \emptyset = [\![\texttt{p1\&w1}]\!] \cap [\![\texttt{p2\&w1}]\!] = [\![\texttt{p1\&?}]\!] \cap [\![\texttt{p2\&?}]\!]$. Therefore `p1.delay()` and `p2.delay()` cannot interfere. On the other hand, because `p1` intersects with `w1`, we are able to create types like `w1&p1` (while we cannot create `p1&p2`) — but the effects $[\![\texttt{w1\&?}]\!]$ and $[\![\texttt{p1\&?}]\!]$ are **not** independent; thus computations like `w1.delay()` and `p1.delay()` may interfere.

Ownership Type Constraints To make MOJO modular, we provide `where` clauses to constrain owner parameters. Inside a class `C` with three owner parameters, `a`, `b`, and `o`, we can create objects with ownership `a&o` only if we are sure that `a` intersects with `o`. We give this guarantee through a where clause:

```
1  class C<o, a, b>
2   where a intersects o {
3
4       Object<a & o> f1; // legal
5       Object<a & b> f2; // illegal
6  }
```

but then we can only instantiate `C` with ownership parameters that are definitively known to intersect. In the example in the previous section, `C<w1,p1,p2>` is legal (because `w1 intersects p2`) while `C<p1,p2,w1>` is illegal because `p1` does not intersect `p2`.

Where clauses can also be used to express disjointness constraints — a declaration such as:

```
1  class D<o, e>
2   where e disjoint o {
3  // ...
4  }
```

requires that the actual ownership parameters be disjoint. In the above example, `D<p1,p2>` is a legal ownership type because `p1 disjoint p2`, but `D<w1,p1>` is not, because those boxes are not disjoint. Note that a disjointness constraint also prevents both parameters being instantiated with the same actual ownership type, because `disjoint` is irreflexive, so `D<p1,p1>` and `D<this,this>` are also illegal.

In practice, we expect that many ownership parameters will use neither intersection or disjointness constraints. This gives maximal polymorphism: unconstrained parameters can be instantiated with either intersecting or disjoint boxes. A class which does not create objects with multiple owners will not need intersection constraints, and a class which is not susceptible to interference between parameters will not need disjointness constraints. Most collection classes, for example, will fall into this category, as will pairs, tuples, and many other generic classes.

Objects allowed to intersect, or guaranteed to be disjoint

$$\frac{\Gamma = ...q\ C\ q'....}{\Gamma \vdash q\ C\ q'} \qquad \frac{}{\Gamma \vdash q \circledcirc q} \qquad \frac{\Gamma \vdash q'\ C\ q}{\Gamma \vdash q\ C\ q'} \qquad \frac{}{\Gamma \vdash q \circledcirc ?}$$

$$\frac{\Gamma \vdash q \ll q'}{\Gamma \vdash q \circledcirc q'} \qquad \frac{\Gamma \vdash q' \ll q'' \qquad \Gamma \vdash q'' \between q}{\Gamma \vdash q \between q'}$$

$$\frac{Q = Q_1 \cap q \quad Q' = Q_2 \cap q' \quad \Gamma \vdash q \between q'}{\Gamma \vdash Q \between Q'} \qquad \frac{Q = Q_1 \cap q,\ Q' = Q_2 \cap q' \implies \Gamma \vdash q \circledcirc q'}{\Gamma \vdash Q \circledcirc Q'}$$

Well-formed types

$$\frac{q \in \mathcal{D}m(\Gamma)}{\Gamma \vdash q} \qquad \frac{}{\Gamma \vdash ?} \qquad \frac{\Gamma \vdash Q \qquad \Gamma \vdash q \qquad Q = Q' \cap q' \implies \Gamma \vdash q \circledcirc q'}{\Gamma \vdash Q \cap q}$$

$$\frac{\texttt{class}\ c < \overline{p} > \ ...\ \lhd ... \qquad |\overline{Q}| = |\overline{p}| \qquad \Gamma \vdash \overline{Q} \qquad Q_i\ C\ Q_j \in pCnstrs(c < \overline{Q} >) \implies \Gamma \vdash Q_i\ C\ Q_j}{\Gamma \vdash c < \overline{Q} >}$$

'Inside' relation for owner parameters

$$\frac{}{\Gamma \vdash q \ll q} \qquad \frac{\Gamma \vdash q \ll q'' \qquad \Gamma \vdash q'' \ll q'}{\Gamma \vdash q \ll q'} \qquad \frac{\Gamma(q) = c < q' \cap \overline{Q}, \overline{Q'} >}{\Gamma \vdash q \ll q'}$$

Figure 4.30: Well-formed types and the 'inside', intersects and disjoint relations for owner parameters

4.4.2 MOJO

In this section we present the MOJO language, a minimal object-oriented imperative language, in the Featherweight Java (FJ) [97] style with extensions for (multiple) ownership. It is closely related to JOE [57] and ODE [165].

The major change from FJ is that MOJO types and classes are parameterised by a sequence of owner parameters, the first of which is the owner of objects of that type. Actual ownership parameters may consist of multiple owners which may include the wildcard owner, "`?`". To support the topology of boxes described in Section 4.4.1, constraints on ownership parameters and final fields may be specified.

MOJO supports imperative features, including a heap and field assignment, and final fields that may be used as ownership parameters (non-final fields would be unsafe as ownership parameters as they may change during execution).

In comparison to the concrete, surface syntax described in Section 4.4.1, the formalism adopts a more succinct abstract syntax: class declarations use $\lhd$ instead of `extends`. Constraints on fields or ownership parameters, use $\circledcirc$ for `intersects` and $\between$ for `disjoint`. To emphasize the connection with set theory, multiple owners use $\cap$ rather than `&`. Actual ownership parameters consist of a set of formal parameters, `this`, final fields, method parameters, the `?` wildcard or, at runtime, addresses. The syntax is given in Figure 4.29.

Runtime model

Heaps (h) map addresses to objects. Objects are triples of a runtime type, a mapping from final field identifiers (Id^{fld}) to addresses, and a mapping from non-final field identifiers (Id^{fld}) to addresses. Runtime types consist of class identifiers and sequences of nonempty sets of addresses, representing actual owners,

including (?)[11]

$h \in Heap$	$=$	$\mathbb{N} \longrightarrow Object$	address to object
$Object$	$=$	$c < \overline{R} > \times$	runtime type
		$(\ Id^{ffld} \longrightarrow \mathbb{N})\ \times$	final field values
		$(\ Id^{fld} \longrightarrow \mathbb{N})$	non-final field values
$\iota \in \mathbb{N}$			object addresses

Execution

Execution is defined in terms of a large steps operational semantics, with format $e, h \rightsquigarrow v, h'$, which maps an expression and a heap to a result and a new heap.

The operational semantics for field assignment and field access is the obvious one and appears in appendix A. The semantics of object creation and method call is more intricate, and we discuss it here in more detail.

To create an object of type $c <\overline{R}>$, we first create a new object at a fresh address ι with temporary type $Object$[12]. We then initialize the final fields of c[13], and obtain objects $\overline{\iota'}$, and a heap h'_n. We then initialize the non-final fields[14], and obtain objects $\overline{\iota''}$, and a heap h'''_m. Finally, in h'''_m we update the class of the new object, and "connect" the final field identifiers to $\overline{\iota'}$, and the nonfinal field identifiers to $\overline{\iota''}$.

$$\frac{\begin{array}{c}\iota \text{ fresh in } h \qquad h_1 = h[\iota \mapsto (Object, \emptyset)] \\ fFields(c <\overline{R}>) = \overline{t\ ff} \qquad |\overline{ff}| = n \\ \texttt{new}\ t[\iota/\texttt{this}], h \rightsquigarrow \iota', h' \qquad \overline{h_{i+1}} = \overline{h'_i} \\ fields(c <\overline{R}>) = \overline{t'\ f} \\ |\overline{f}| = m \qquad h''_1 = h'_n \\ \texttt{new}\ t'[\overline{\iota'/ff}, \iota/\texttt{this}], h'' \rightsquigarrow \iota'', h''' \qquad \overline{h''_{i+1}} = \overline{h'''_i}\end{array}}{\texttt{new}\ c <\overline{R}>, h\ \rightsquigarrow\ \iota, h'''_m[\iota \mapsto (c <\overline{R}>, \overline{ff \mapsto \iota'}, \overline{f \mapsto \iota''})]}$$

Method calls evaluate the receiver and the argument, and look up the method body in the class as usual. More interestingly, in e_3, the method body, we replace the formal receiver by the actual one ($\iota/\texttt{this}$), the formal parameter by the actual one (ι'/x), and any appearance of the final fields in the types by the actual values of the final fields as found in the heap ($\overline{\iota/ff}$). The class's ownership parameters will have already been replaced by the corresponding sets of owners in the object's runtime type ($\overline{R/p}$) by the $mBody$ function.

$$\frac{\begin{array}{c}e_1, h \rightsquigarrow \iota, h'' \qquad e_2, h'' \rightsquigarrow \iota'', h''' \\ h'''(\iota) = (c <\overline{R}>, \overline{ff \mapsto \iota}, \overline{f \mapsto ...}) \\ mBody(m, c <\overline{R}>) = (x, e_3) \\ e_3[\iota/\texttt{this}, \iota''/x, \overline{\iota/ff}], h''' \rightsquigarrow \iota', h'\end{array}}{e_1.m(e_2), h \rightsquigarrow \iota', h'}$$

Well-formed types

In Figure 4.30 we define the following five judgments:

$\Gamma \vdash q \ll q'$	q guaranteed to be inside q'
$\Gamma \vdash q \circledcirc q'$	q allowed to intersect q'
$\Gamma \vdash q \between q'$	q guaranteed disjoint with q'
$\Gamma \vdash Q$	Q consists of qs allowed to intersect
$\Gamma \vdash c <\overline{Q}>$	$c <\overline{Q}>$ well-formed type

[11] Allowing ? gives meaning to the expression `new Task<?,p>`. In MOJO we may want objects with unknown owners, in contrast to Java, where no object is instantiated with wildcard type.

[12] We do not give the newly created object the class $c\overline{<R>}$, in order to avoid objects with uninitialized final fields. We give ι the type $c\overline{<R>}$ only after the values for all new fields are available.

[13] We replace any occurrence of `this` in any type t_j by ι, the new object, and any occurrence of the source formal parameter p_i by the corresponding (set of) runtime actual parameters R_i.

[14] We replace any source actual ownership parameter ff_i by ι'_i. the corresponding value of that field.

$$\frac{Q_i = {Q_i}' \cap ? \implies p_i \text{ does not appear in } t}{t \vdash \overline{Q}/\overline{p}}$$

$$\frac{}{\Gamma \vdash q : \Gamma(q)} \qquad \frac{\Gamma \vdash t}{\Gamma \vdash \texttt{new}\ t : t}$$

$$\frac{\Gamma \vdash e : t' \quad t' <: t}{\Gamma \vdash e : t}$$

$$\frac{\Gamma \vdash e : c<\overline{Q}> \quad allFields(c<\overline{Q}>) = \overline{t\ f}}{\Gamma \vdash e.f_i : {t_i}^{\Gamma \cdot e}}$$

$$\frac{\Gamma \vdash e : c<\overline{Q}> \quad fields(c<\overline{p}>) = \overline{t\ f} \quad t_i \vdash \overline{Q}/\overline{p} \quad \Gamma \vdash e' : [\overline{Q/p}]t_i}{\Gamma \vdash e.f_i = e' : [\overline{Q/p}]{t_i}^{\Gamma \cdot e}}$$

$$\frac{\Gamma \vdash e : c<\overline{Q}> \quad mType(m, c<\overline{p}>) = t' \to t \quad t' \vdash \overline{Q}/\overline{p} \quad \Gamma \vdash e' : [\overline{Q/p}]t'}{\Gamma \vdash e.m(e') : [\overline{Q/p}]t^{\Gamma \cdot e}}$$

$$\frac{}{q \sqsubseteq q} \qquad \frac{}{q \sqsubseteq ?} \qquad \frac{\forall q' \in Q' : \exists q \in Q \ st\ q \sqsubseteq q'}{Q \sqsubseteq Q'}$$

$$\frac{\texttt{class}\ c<\overline{p}> ... \lhd c'<\overline{Q'}> ...}{c<\overline{Q}> <: c'<[\overline{Q/p}]\overline{Q'}>} \qquad \frac{\overline{Q \sqsubseteq Q'}}{c<\overline{Q}> <: c<\overline{Q'}>} \qquad \frac{t <: t'' \quad t'' <: t'}{t <: t'}$$

Figure 4.31: Typing and subtyping rules

An environment, Γ, maps `this`, x and ι to types, and contains a set of formal ownership parameters (p) and intersects and disjoints relationships declared in the class of the receiver.

The operator $\cap$ is associative and commutative, and the empty sequence ϵ is neutral, *i.e.* $\epsilon \cap Q = Q$.

An object is inside another, if its box (that is, the set of objects it owns) is a subset of the box of the other.

The relations ∞ and $\wr$ extend the declared intersections and disjointness of owner parameters and fields. The `disjoint` relation makes use of the inside ($\ll$) relation for owner parameters.

A type $c<\overline{Q}>$ is well-formed in the context of an environment Γ, iff: a) there is a Q for each formal parameter p; b) each Q is well-formed (*i.e.* consists of ownership parameters which are allowed to intersect); and c) if two parameters are declared to intersect or be disjoint in the class declaration, then the environment Γ will allow the parameters to intersect or guarantee them to be disjoint, respectively.

Subtypes

In Figure 4.31 we define the subtyping relation $t' <: t$. The auxiliary judgment $Q \sqsubseteq Q'$ guarantees that Q' is the same as Q except that some of the contents of Q may be replaced by `?`. Note that $\_ \sqsubseteq \_$ is reflexive and transitive, but *not* symmetric.

For types $c<\overline{Q}>$ and $c<\overline{Q'}>$, if no `?` appears in $\overline{Q}$ or $\overline{Q'}$, subtyping is invariant with respect to the ownership parameters. For example, $\texttt{C}<\texttt{o}_1 \cap \texttt{o}_2>$ is not a subtype of $\texttt{C}<\texttt{o}_1>$ — to allow such a

relation would be unsound. The `?` owner introduces variance, not only with respect to the owners, but also to the *number of owners*. For example, as well as the obvious relationship `C <o> <: C <?>`, we also have `C <`$\mathtt{o}_1 \cap \mathtt{o}_2$`> <: C <?>`. This gives the equivalence $? \cap \overline{Q} \equiv ? \cap ? \cap \overline{Q}$.

Types of expressions

The type of an expression e depends on an environment Γ and is given by the judgment $\Gamma \vdash e : t$ defined in Figure 4.31. The rules are as expected for an ownership type system, with some special care taken for field assignment and parameter passing when the types involve ?, where we require substitutions to be good, *i.e.* to introduce no more `?`s than in the original field or method return type. Consider the following classes:

```
1  class B<b1>{ ... }
2  class C<c1>{ B<c1> f1; B<?> f2; }
```

in the example:

```
1  class Test<t1,t2>{
2    void m1(C<t1> x, C<?> y){
3      x.f2 = new B<t2>; // type correct
4      x.f2 = new B<?>; // type correct
5      y.f1 = new B<t2>; // type error
6      y.f1 = new B<?>; // type error
7      y.f2 = new B<?>; // type correct
8    }
9  }
```

the assignments to `x.f2` are type correct because from the point of view of `x` its field `f2` may contain a `D<`Q`>`, for *any* actual owners Q. On the other hand, any assignment to `y.f1` is type-incorrect, because from the point of view of `y` its field `f1` must contain a `D<`Q`>`, for some *fixed* actual owners Q, which are unknown in the current context. In terms of our formal description, the first two and the last assignment are type correct, because for all Q, it holds that `B<any>` $\vdash Q$ `/c1`; the next two assignments are type incorrect, because `B<c1>` $\not\vdash$ `any /c1`.

Furthermore, the types of fields and methods needs to treat the actual ownership parameter `this` specially; ie it replaces `this` by the expression whose field or method is being selected, provided that e denotes a constant value. This is described through $t^{\Gamma \cdot e}$, defined as follows, where $x \in t$ means that x appears in t:

$$t^{\Gamma \cdot e} = \begin{cases} t, & \text{if } \mathtt{this} \notin t, \textit{or } e = \mathtt{this}; \\ [e/\mathtt{this}]t & \text{if } \mathtt{this} \in t, e \in \{\iota, x, p\}; \\ [\mathit{ff}/\mathtt{this}]t, & \text{if } \mathtt{this} \in t, e = \mathtt{this}.\mathit{ff}; \\ [\iota'/\mathtt{this}]t, & \text{if } \mathtt{this} \in t, e = \iota.\mathit{ff}, \Gamma(\iota.\mathit{ff}) = \iota'; \\ \bot, & \text{otherwise.} \end{cases}$$

Well-formed class and program

A class is well-formed, if it has same owner as the superclass, the superclass type is well-formed, types mentioned in fields and methods are well-formed, and method bodies are well typed. For checking well-formedness of the superclass, constraints between ownership parameters are taken into account. For checking types of final fields, `this` is allowed to appear in $\overline{t}$. For checking types of fields and method bodies, constraints between final fields are also taken into account. Fields must not overlap with those from the superclass. Finally, the constraints on ownership parameters and final fields must be well-formed.

$$\frac{\begin{array}{c}\texttt{class}\ c<\overline{p}>\ \overline{pCnstr} \lhd c'<\overline{Q}> \\ \{\ \overline{\texttt{fin}\ t\ \mathit{ff}}\ \overline{fCnstr}\ \overline{t'\ f}\ \overline{mth}\ \} \\ first(\overline{p}) = first(\overline{Q}) \\ \Gamma = \overline{p}, pCnstrs(c<\overline{p}>) \qquad \Gamma \vdash c' < \overline{Q} > \\ \Gamma' = \Gamma, \texttt{this} \qquad \Gamma' \vdash \overline{t} \\ \Gamma'' = \Gamma', fCnstrs(c<\overline{p}>) \qquad \Gamma' \vdash \overline{t'} \\ \Gamma''' = \Gamma'', c<\overline{p}>\ \texttt{this} \qquad \Gamma''' \vdash \overline{mth} \\ fFields(c'<\overline{Q}>) = \overline{t''\ \mathit{ff}'} \qquad \overline{\mathit{ff}'} \cap \overline{\mathit{ff}} = \emptyset \\ fields(c'<\overline{Q}>) = \overline{t'''\ f'} \qquad \overline{f'} \cap \overline{f} = \emptyset \\ \vdash \Gamma'''\diamond \end{array}}{c<\overline{p}>\ \text{well formed}}$$

The constraints on owner parameters and fields are well-formed if they contain no contradictions:

$$\frac{\Gamma \vdash p \circledcirc p' \Rightarrow \Gamma \nvdash p \between p' \qquad \Gamma \vdash p \between p' \Rightarrow \Gamma \nvdash p \circledcirc p'}{\vdash \Gamma\diamond}$$

A method body is well typed if it contains an expression of the same type as the return type of the method, and if overriding is legal.

$$\frac{\begin{array}{c}\Gamma(this) = c<\overline{p}> \\ \texttt{class}\ c<\overline{p}>\ \overline{pCnstr} \lhd c' < \overline{Q} > ... \\ \Gamma \vdash t \qquad \Gamma \vdash t' \qquad \Gamma, t'\ x \vdash e : t \\ override(m, c' < \overline{Q} >, t' \to t) \end{array}}{\Gamma \vdash t\ m(t'\ x)\{e\}}$$

Runtime types

The function *env* maps heaps to typing environments, enriching these with information about the values of final fields:

Definition 4.4.1. *We define env as follows:*

$$\begin{array}{rl} env(\overline{\iota \mapsto obj}) = & \overline{env(\iota \mapsto obj)} \\ env(\iota \mapsto obj) = & \{\ \iota : c<\overline{R}>\ \} \cup \{\ (\iota_i\ C\ \iota_j) \mid \\ & \mathit{ff}_i\ C\ \mathit{ff}_j \in fCnstrs(c<\overline{R}>)\} \\ & \cup \{\ \iota.\overline{\mathit{ff}} \mapsto \overline{\iota}\ \} \\ & \textit{where}\ \ obj = (c<\overline{R}>, \overline{\mathit{ff} \mapsto \iota}, ...) \end{array}$$

Wherever an environment gives a judgment, a corresponding heap gives the same judgment:

$$env(h) = \Gamma \qquad \Gamma \vdash jud\_x \implies h \vdash judg\_x$$

Thus we obtain judgements for typing expressions, well-formed types, the inside relation, etc:

$h \vdash \iota \ll \iota'$ $\quad h \vdash Q \circledcirc Q'$ $\quad h \vdash Q \between Q'$ $\quad \vdash h\diamond$

$h \vdash Q$ $\quad h \vdash c<Q>$ $\quad h \vdash e : t$

Well-formed heap

In Figure 4.32 we define well-formed objects and heaps. An object ι in the heap is well-formed, expressed by $h \vdash \iota$, if its type is well-formed, and all its fields have types according to their static types where the ownership parameters have been substituted according to the runtime class of ι, and its static fields. The heap is well-formed if all objects in the heap are well-formed; where the boxes of objects intersect in the heap, the heap can show that these objects are in a $\circledcirc$ relationship; if the heap can show that two objects are in a $\between$ relationship then, their boxes do not overlap; finally, the heap must contain no contradictions, *i.e.* there exist no objects ι, ι' such that $h \vdash \iota \circledcirc \iota'$ and $h \vdash \iota \between \iota'$.

$$[\![\ \iota\ ]\!]_h = \{\iota' \mid h \vdash \iota' \ll \iota\ \}$$

$$\frac{\begin{array}{c} h(\iota) = (c < \overline{R} >, \overline{ff \mapsto \iota}, \overline{f \mapsto \iota'}) \qquad h \vdash c < \overline{R} > \\ fFields(c < \overline{R} >) = \overline{t\ ff} \qquad h \vdash \overline{\iota} : [\iota/\texttt{this}]\overline{t} \\ fields(c < \overline{R} >) = \overline{t'\ f} \qquad h \vdash \overline{\iota'} : [\iota/\texttt{this}, \overline{\iota/ff}]\overline{t'} \end{array}}{h \vdash \iota}$$

$$\frac{\begin{array}{c} \forall \iota \in \mathcal{D}m(h) :\ h \vdash \iota \\ [\![\ \iota\ ]\!]_h \cap [\![\ \iota'\ ]\!]_h \neq \emptyset \implies h \vdash \iota \circledcirc \iota' \\ h \vdash \iota \otimes \iota' \implies [\![\ \iota\ ]\!]_h \cap [\![\ \iota'\ ]\!]_h = \emptyset \\ \vdash h \diamond \end{array}}{\vdash h}$$

Figure 4.32: Well-formed objects and heaps

Soundness of the type system

We first prove that runtime types, and the inside relation are invariant, while disjointness and possible intersection of objects are monotonic with execution:

Lemma 4.4.2. *In a well-formed program, if $\iota, \iota' \in \mathcal{D}m(h)$, and $e, h \rightsquigarrow \iota'', h'$, and $h \vdash e : t'$, then*

1. $h \vdash \iota\ :\ t$ *if and only if* $h' \vdash \iota\ :\ t$
2. $h \vdash \iota \ll \iota'$ *if and only if* $h' \vdash \iota \ll \iota'$
3. $h \vdash \iota\ C\ \iota'$ *implies* $h' \vdash \iota\ C\ \iota'$

As usual in soundness proofs, we need a substitution lemma; in our particular setting, the substitution needs to be aware of ownership and allowed/forbidden intersections.

For a substitution σ which maps q to q, we define its expansion, σ_h, so that it also maps final fields (ff), or formal ownership parameters. We then define the concept of an appropriate substitution $\Gamma, h \vdash \sigma$, as one which preserves all constraints implied in Γ:

Definition 4.4.3. *Given a $\sigma : \{\ this, x\} \longrightarrow \mathbb{N}$, we define:*

- $\sigma_h\ :\ Q \longrightarrow \mathcal{P}wr(\mathbb{N})$ *as follows:*
  1. $\sigma_h(\texttt{this}) = \sigma(this),\ \ \sigma_h(x) = \sigma(x)$.
  2. $\sigma_h(p) = R_i$ *if* $h(\sigma(\texttt{this})) = (c < \overline{R}>, ..., ...)$ *and* $\mathcal{D}m(c) = \overline{p}$ *and* $p = p_i$*; undefined otherwise.*
  3. $\sigma_h(ff) = \iota_i$ *if* $h(\sigma(\texttt{this})) = (..., \overline{ff \mapsto \iota}, ...)$ *and* $ff = ff_i$*; undefined otherwise.*
- $\sigma_h \circ t$ *indicates the application of* σ_h *on the type* t.
- $\sigma_h \circ e$ *indicates the application of* σ_h *on the expression* e.
- $\Gamma, h \vdash \sigma$ *iff for any constraint* C*:*
  1. $q\ C\ q' \in \Gamma \implies h \vdash \sigma_h(q)\ C\ \sigma_h(q')$,
  2. $h \vdash \iota : \sigma_h \circ \Gamma(\texttt{this})$,
  3. $h \vdash \sigma(x) : \sigma_h \circ \Gamma(x)$,
  4. $p \in \mathcal{D}m(\Gamma) \implies \sigma_h(p) \subseteq \mathcal{D}m(h)$,
  5. $p \in \mathcal{D}m(\Gamma),\ \iota, \iota' \in \sigma_h(p) \implies h \vdash \iota \circledcirc \iota'$.

We can now prove the substitution lemma:

Lemma 4.4.4. *If $\Gamma, h \vdash \sigma$ then:*

1. $\Gamma \vdash t$ *implies* $h \vdash \sigma_h \circ t$.

2. $\Gamma \vdash t' <: t$ *implies* $h \vdash \sigma_h \circ t' <: \sigma_h \circ t$.

3. $\Gamma \vdash e : t$ *implies* $h \vdash \sigma_h \circ e : \sigma_h \circ t$.

We define $t^{[c \lhd c']}$ the "projection" of a type t as seen from a class c to the way it is seen from a subclass c', and similarly, of an environment or an expression:

Definition 4.4.5. *For environment* Γ*, classes* c*,* c'*, type* t*, expression* e*,* $\overline{p} = \mathcal{D}m(c)$*,* $\overline{p'} = \mathcal{D}m(c')$*, we define:*

$$
\begin{array}{ll}
e^{[c \lhd c']} = & [\overline{Q/p}]e, \text{ if } c' <\overline{p'}> <: \ c <\overline{Q}> \\
& \text{undefined,} \quad \text{otherwise.} \\
t^{[c \lhd c']} = & [\overline{Q/p}]t, \text{ if } c' <\overline{p'}> <: \ c <\overline{Q}> \\
& \text{undefined,} \quad \text{otherwise.} \\
\Gamma^{[c \lhd c']} = & [\overline{Q/p}]t' \ x, c' <\overline{p'}> \ this, \overline{p'}, \overline{fConstr}, \overline{pConstr'} \\
& \text{if } c' <\overline{p'}> <: \ c <\overline{Q}> \text{ for some } \overline{Q}, \text{ and} \\
& \Gamma = t' \ x, c <\overline{p}> \ this, \overline{p}, \overline{fConstr}, \overline{pConstr}, \\
& \text{and } \overline{pConstr'} = [\overline{Q/p}]\overline{pConstr}. \\
& \text{undefined,} \quad \text{otherwise.}
\end{array}
$$

We then prove that projection to subclasses preserves typing. In other words, if a type t is well formed in an environment from class c, then the projection of t onto the subclass c' is well-formed in the environment as defined in the subclass c'.

Lemma 4.4.6. *For classes* c*, and* c'*, environments* Γ *so that* $\Gamma^{[c \lhd c']}$ *is defined:*

- $\Gamma \vdash t$ *implies* $\Gamma^{[c \lhd c']} \vdash t^{[c \lhd c']}$.
- $\Gamma \vdash t <: t'$ *implies* $\Gamma^{[c \lhd c']} \vdash t^{[c \lhd c']} <: t'^{[c \lhd c']}$.
- $\Gamma \vdash e : t$ *implies* $\Gamma^{[c \lhd c']} \vdash e^{[c \lhd c']} : t^{[c \lhd c']}$.

Theorem 4.4.7. *For a well formed program, if* $h \vdash e : t$ *and* $\vdash h$ *and* $e, h \rightsquigarrow \iota, h'$*, then* $h' \vdash \iota : t$*, and* $\vdash h'$.

Proof. By structural induction, and using Lemmas 4.4.2, and 4.4.4 and 4.4.6.

4.4.3 Effects

Effects are used to give a conservative estimate of the area of the heap read or written by an expression. We describe these areas through one or more boxes, where the "$\cup$" operator describes the union of such boxes. The first part of an effect is the area being read; the second is the area being written:

$$
\begin{array}{lll}
\phi & ::= \ \epsilon \ | \ Q \ \cup \ \phi & \textit{boxes} \\
\psi \in Effect & ::= \ \phi/\phi & \textit{effect}
\end{array}
$$

We expect programs to come equipped with a function to give us the effects of a method[15]:
$\mathcal{M}_{\mathit{eff}}(\_,\_) : Id^{class} \times Id^{mth} \longrightarrow Effect$

The example with effects

We now revisit the example from Section 4.4.1, and give the values for the function $\mathcal{M}_{\mathit{eff}}(\_,\_)$ through comments in the code.

```
1  class Duration<d1> {
2      Date<this> start;  Date<this> end;
3      void delay() {...}      // EFF: this / this
4  }
```

[15]Through the lookup function we skip the requirement for the definition of syntax.

```
1  class Task<t1> {
2      Duration<this> duration;
3      void delay() {...}      // EFF: this / this
4  }
```

```
1  class Worker<w1>{
2      TaskList<this, this & ?> tasks;
3      void insert (Task<this & ?> t) {...}
4                              // EFF: this / this
5      void delay() {...}      // EFF: this & ? / this & ?
6  }
```

```
1  class TaskList<l1,l2>{
2      TaskList<l1,l2> next;
3      Task<l2> task;
4      void insert (Task<l2> t)     // EFF: l1 / l1
5      void delay() {...}           // EFF: l2 / l2
6  }
```

```
1  class Project<p>{ //
2      TaskList<this, this & ?> tasks;
3      void insert (Task<this & ?> t){...} //
4                             // EFF: this / this
5      void delay() {...}     // EFF: this & ? / this & ?
6  }
```

Effects of Expressions

In this and the following sections we introduce effects for expressions, and the disjointness and inside relations for effects. We go on to prove soundness of the effect system (Theorem 4.4.15): that is, if the effects of two expressions are disjoint, then the order of their execution is unimportant.

The effects of expressions are defined through the judgment $\Gamma \vdash_{\mathsf{e}} e \ : \ \phi \ / \ \phi'$, given in Figure 4.33. The rules are fairly straightforward, with effects of sub-expressions propagated to the enclosing expression; reading or writing a field causing a read or write effect. Method invocation is more interesting: care must be taken to substitute the owners of the receiver into the effects of the method body correctly. In our example:

```
1   final Worker<this> w1 = new Worker<this>;
2   final Worker<this> w2 = new Worker<this>;
3   w1 disjoint w2;
4
5   w1.delay();      // EFF: w1 & ? / w1 & ?
6   w2.delay();      // EFF: w2 & ? / w2 & ?
7
8   final Project<this> p1=new Project<this>;
9   p1.delay();      // EFF: p1 & ? / p1 & ?
```

The inside relation for effects ($effll$) is given in Figure 4.33. Intuitively, an effect is inside another if it covers a smaller part of the heap than the other. We can prove that the write effect is always inside the read effect for any expression:

Lemma 4.4.8. *If* $\Gamma \vdash_{\mathsf{e}} e \ : \ \phi \ / \ \phi'$, *then* $\Gamma \vdash \phi' \ll_{\mathsf{e}} \phi$.

Proof. Straightforward induction on $\Gamma \vdash_{\mathsf{e}} e \ : \ \phi \ / \ \phi'$.

Projecting effects onto the heap

Based on the $\ll$ relation for objects (from Figure 4.30), we define the projection of an effect ϕ to a heap through $[\![\phi]\!]_h$:

$$\frac{q \in \{\texttt{this}, x\}}{\Gamma \vdash_{\mathsf{e}} q \,:\, \epsilon \,/\, \epsilon} \qquad \frac{\Gamma \vdash t}{\Gamma \vdash_{\mathsf{e}} \texttt{new}\ t \,:\, \epsilon \,/\, \epsilon} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi_1 \,/\, \phi_2 \quad \Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_3 \quad \Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_4 \quad \Gamma \vdash \phi_3 \ll_{\mathsf{e}} \phi_4}{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi_3 \,/\, \phi_4}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi \,/\, \phi' \qquad \Gamma \vdash e : c < Q, \overline{Q} >}{\Gamma \vdash_{\mathsf{e}} e.f \,:\, \phi \cup Q \,/\, \phi'} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi_1 \,/\, \phi_2 \qquad \Gamma \vdash_{\mathsf{e}} e' \,:\, \phi_3 \,/\, \phi_4 \quad \Gamma \vdash e : c < Q, \overline{Q} >}{\Gamma \vdash_{\mathsf{e}} e.f = e' \,:\, \phi_1 \cup \phi_3 \cup Q \,/\, \phi_2 \cup \phi_4 \cup Q}$$

$$\frac{\Gamma \vdash e : c < \overline{Q} > \qquad \phi/\phi' = \mathcal{M}_{\mathit{eff}}(c < \overline{p} >, m) \qquad \Gamma \vdash_{\mathsf{e}} e \,:\, \phi_1 \,/\, \phi_2 \quad \Gamma \vdash_{\mathsf{e}} e' \,:\, \phi_3 \,/\, \phi_4 \quad \Gamma \vdash e' : [\overline{Q}/\overline{p}]\, t'}{\Gamma \vdash_{\mathsf{e}} e.m(e') \,:\, \phi_1 \cup \phi_3 \cup [\overline{Q}/\overline{p}]\,\phi \;/\; \phi_2 \cup \phi_4 \cup [\overline{Q}/\overline{p}]\,\phi'}$$

$$\frac{}{\Gamma \vdash \epsilon \ll_{\mathsf{e}} \phi} \qquad \frac{\Gamma \vdash q \ll q'}{\Gamma \vdash q \ll_{\mathsf{e}} q'} \qquad \frac{\Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_2 \qquad \Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_3}{\Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_3}$$

$$\frac{\Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_3 \qquad \Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_4}{\begin{array}{c}\Gamma \vdash \phi_1 \cup \phi_2 \ll_{\mathsf{e}} \phi_3 \cup \phi_4 \cup \phi_5 \\ \Gamma \vdash \phi_1 \cap \phi_2 \ll_{\mathsf{e}} \phi_3 \cap \phi_4\end{array}}$$

$$\frac{}{\Gamma \vdash \epsilon \# \phi} \qquad \frac{\Gamma \vdash \phi \# \phi'}{\Gamma \vdash \phi' \# \phi} \qquad \frac{\Gamma \vdash \phi \# \phi' \quad \Gamma \vdash \phi \# \phi''}{\Gamma \vdash \phi \# \phi' \cup \phi''} \qquad \frac{\Gamma \vdash q \between q'}{\Gamma \vdash q \cap Q \# q' \cap Q'}$$

Figure 4.33: Effect rules and 'inside' and disjointness relations for expressions.

Definition 4.4.9.

$$\begin{array}{lcl} [\![r]\!]_h & = & \{\ \iota \mid h \vdash \iota \ll r\} \\ [\![r \cap R]\!]_h & = & [\![r]\!]_h \cap [\![R]\!]_h \\ [\![\phi \cup \phi']\!]_h & = & [\![\phi]\!]_h \cup [\![\phi']\!]_h \end{array}$$

We can prove that the type of an expression describes the boxes to which its evaluation will belong:

Lemma 4.4.10. *If $h \vdash e : c < R, \overline{R} >$ and $e, h \rightsquigarrow \iota, h'$, then $\iota \in [\![R]\!]_{h'}$.*

Proof. Straightforward application of the definitions (Definitions 4.4.9, and $\ll$ from Figure 4.30), and Theorem 4.4.7.

We give rules for judging the disjointness relation ($\Gamma \vdash \phi \# \phi'$) in Figure 4.33. The rules state that the empty effect is disjoint from all effects; that the disjoint relation is symmetric and distributive with respect to the union of effects; and that if any pair of owners in a pair of sets of multiple owners are disjoint (by the $\between$ relation), then the effects denoted by this pair of sets is disjoint (by the # relation).

In the following lemma, the first two assertion guarantee soundness of the inside and disjointness judgments are sound wrt. the projection of effects. The last assertion is the counterpart to Lemma 4.4.4.

Lemma 4.4.11. *For any effects ϕ, ϕ', environment Γ, substitution σ with $\Gamma, h \vdash \sigma$, and $\vdash h$, we have*

- *If $\Gamma \vdash \phi \ll_{\mathsf{e}} \phi'$, then $[\![\sigma_h \circ \phi]\!]_h \subseteq [\![\sigma_h \circ \phi']\!]_h$.*
- *If $\Gamma \vdash \phi \# \phi'$, then $[\![\sigma \circ \phi]\!]_h \cap [\![\sigma \circ \phi']\!]_h = \emptyset$.*
- *If $\Gamma \vdash_{\mathsf{e}} e \,:\, \phi \,/\, \phi'$ and $\Gamma, h \vdash \sigma$, then $h \vdash_{\mathsf{e}} \sigma_h \circ e \,:\, \sigma_h \circ \phi \,/\, \sigma_h \circ \phi'$.*

Proof. by induction on the derivation of $\Gamma \vdash \phi \ll_{\mathsf{e}} \phi'$, respectively of $\Gamma \vdash \phi \# \phi'$, respectively of $\Gamma \vdash_{\mathsf{e}} e \,:\, \phi \,/\, \phi'$, and using Definition 4.4.3.

Well-formed Programs with effects

A program is well formed if, in addition to the requirements from Section 4.4.2, a) each method body has read/write effect inside its declared effect, and b) the effect of an overriding method is inside the effect of any overridden method. Formally, we require that a) $t\ m(t'\ x)\{e\}$ in $c < \overline{p} >$ implies that $\Gamma \vdash_{eff} e : \mathcal{M}_{\mathit{eff}}(c, m)$, and b) $\mathcal{M}_{\mathit{eff}}(c, m) = \phi_1/\phi_2$ and $\mathcal{M}_{\mathit{eff}}(c', m) = \phi_3/\phi_4$ and $c<\overline{p}><: c' < \overline{Q} >$ implies that $\Gamma \vdash \phi_1 \ll_{\mathsf{e}} [\overline{Q}/\overline{p}]\phi_3$ and $\Gamma \vdash \phi_2 \ll_{\mathsf{e}} [\overline{Q}/\overline{p}]\phi_4$, where $\Gamma = \overline{p}, c < \overline{p} >$ `this`.

The counterpart to Lemma 4.4.6, guarantees that the effect of an expression is preserved in a subclass modulo the necessary renamings for ownership parameters:

Definition 4.4.12. *For environment Γ, classes c and c', where $\overline{p} = \mathcal{D}m(c)$, $\overline{p'} = \mathcal{D}m(c')$, and effect ϕ, we define:*

$$\phi^{[c \lhd c']} = \begin{array}{ll} [\overline{Q/p}]\phi, & \textit{if } c' <\overline{p'}> <: \ c <\overline{Q}> \\ \textit{undefined}, & \textit{otherwise.} \end{array}$$

Lemma 4.4.13. *For classes c, and c', and environments Γ so that $\Gamma^{[c \lhd c']}$ is defined:*

- $\Gamma \vdash \phi \ll_{\mathsf{e}} \phi'$ *implies* $\Gamma^{[c \lhd c']} \vdash \phi^{[c \lhd c']} \ll_{\mathsf{e}} \phi'^{[c \lhd c']}$.
- $\Gamma \vdash_{\mathsf{e}} e : \phi \ / \ \phi'$ *implies* $\Gamma^{[c \lhd c']} \vdash_{\mathsf{e}} e^{[c \lhd c']} : \phi^{[c \lhd c']} \ / \ \phi'^{[c \lhd c']}$.

Soundness of the Effects System

Soundness of the effects system guarantees that the read and write effects completely describe the areas of the heap read and written during some execution. In the following, we use the * operator, inspired by separation logic notation for "concatenation" of functions with disjoint domains. Thus, the construction $h * h'$ implicitly guarantees disjointness of h and h'.

Theorem 4.4.14. *In a well formed program, if $\Gamma \vdash_{\mathsf{e}} e : \phi \ / \ \phi'$, and $\Gamma, h \vdash \sigma$ and $\sigma \circ e, h \rightsquigarrow \iota, h'$, then there exist heaps h_1, h_2, h_3, h_4 and h'_2 so that:*

- $h = h_1 * h_2 * h_3$, *and* $[\![\sigma \circ \phi]\!]_h = dom(h_1 * h_2)$, *and* $[\![\sigma \circ \phi']\!]_h = dom(h_2)$,
- $e, h_1 * h_2 \rightsquigarrow \iota, h_1 * h'_2 * h_4$,
- $h' = h_1 * h'_2 * h_3 * h_4$, *and* $[\![\sigma \circ \phi]\!]_{h'} = dom(h_1 * h'_2)$, *and* $[\![\sigma \circ \phi']\!]_{h'} = dom(h'_2)$.

Proof. By induction on the derivation of $e, h \rightsquigarrow \iota, h'$. We use a "generation lemma for effects", *e.g.* that $\Gamma \vdash_{\mathsf{e}} e.f : \phi_1 \ / \ \phi_3$ implies that for some ϕ_3 and ϕ_4, we have that $\Gamma \vdash_{\mathsf{e}} e : \phi_3 \ / \ \phi_4$, $\Gamma \vdash e : c < Q, \overline{Q} >$, and $\Gamma \vdash \phi_3, Q \ll_{\mathsf{e}} \phi_1$, and $\Gamma \vdash \phi_2, Q \ll_{\mathsf{e}} \phi_4$, and $\Gamma \vdash \phi_3, Q \ll_{\mathsf{e}} \phi_4$. We also use the fact that $e, h \rightsquigarrow \iota, h'$ implies that if h' and h'' are disjoint, then $e, h * h'' \rightsquigarrow \iota, h' * h''$

We now prove that execution of two expressions with disjoint effects is independent, in the sense that the order of their execution is immaterial:

Theorem 4.4.15. *In a well formed program, if $\Gamma, h \vdash \sigma$, and $\vdash h$ and $\Gamma \vdash_{\mathsf{e}} e_1 : \phi_1 \ / \ \phi_2$, and $\Gamma \vdash_{\mathsf{e}} e_2 : \phi_3 \ / \ \phi_4$, and $\Gamma \vdash \phi_1 \# \phi_4$ and $\Gamma \vdash \phi_2 \# \phi_3$, then*

$$\begin{array}{c} \sigma \circ e_1, h \rightsquigarrow \iota', h'', \quad \sigma \circ e_2, h'' \rightsquigarrow \iota, h', \\ \textit{implies} \\ \sigma \circ e_2, h \rightsquigarrow \iota, h''', \quad \sigma \circ e_1, h''' \rightsquigarrow \iota', h' \end{array}$$

Proof. The proof is based on Matthew Smith's thesis [165], which develops an abstract model of independence of expressions based on disjointness of effects for any languages satisfying a set of basic requirements. Theorem 3.5.2 from [165] guarantees the assertion of our theorem provided that the heap satisfies basic composition and decomposition properties (**SH1**-**SH6** in [165]), that execution also satisfies basic decomposition properties (**LL2**,**L1**-**L5** in [165]), and that effects also satisfy decomposition properties (**LS1**-**LS5**). Property **LS4** corresponds to Theorem 4.4.14. All the other properties can be easily proven for MOJO.

In terms of our example

```
1  w1 disjoint w2;
2     w1.delay();   // EFF: w1 & ? / w1 & ?
3     w2.delay();   // EFF: w2 & ? / w2 & ?
4     p1.delay();   // EFF: p1 & ? / p1 & ?
```

From `e1#e2` we obtain that `e1&?#e2&?` and therefore `e1.delay()` and `e2.delay()` are independent of each other in the sense of the above theorem. On the other hand, `e1.delay()` and `p1.delay()` are not necessarily independent as we have no information regarding the disjointness of `w1` and `p1`.

4.4.4 Conclusion

Multiple ownership does not impose an ownership tree onto the objects in a program: rather, it allows DAGs, and places objects into boxes — sets — that may intersect or remain disjoint as best serves the program's design. This allows more flexibility in expressing the architecture of a program, and fits well with recent trends in aspect oriented programming.

As we said earlier, we developed MOJO as an extension of traditional ownership types, because of the availability of explicit ownership parameters, and the possibility to name the owners. We now plan to extend GUJ with multiple owners; this will require the use of a simple form of path dependent types, *e.g.* `x`.rep & y.rep A a to express that `a` has class `A` and is owned by `x` and `y`.

We have shown how multiple ownership can be used to demonstrate independence of expressions from each other, and in future work we plan to adapt these ideas for modular verification (Task 3.4).

Chapter 5

Conclusions

We developed new type systems that guarantee adherence to security-related properties of mobile code. Hereby, we addressed the enhanced security requirements of global computing as identified in MOBIUS Deliverable D1.1 [123]. We successfully lifted the scope of type-based, static analysis techniques for bytecode from basic soundness properties to the semantic properties of information flow (Sections 2.1.4, 2.2.5, 2.3.1), resource consumption (Sections 3.1, 3.2.2,3.3.2,3.4.5), and aliasing (Sections 4.1.5, 4.3.4, 4.4.3).

We achieved the objectives of Tasks 2.1 (see Chapter 2), 2.3 (see Chapter 3) and 2.5 (see Chapter 4) within MOBIUS. For information flow security, we defined noninterference-like security properties, and showed that these properties are enforced by the type systems we developed. The type system for information flow control is the first that can be applied to a realistic, low-level language such as Java bytecode that includes features such as objects, exceptions, methods and concurrency (Chapter 2). For resource control, we managed to statically enforce bounds on the amount of heap space allocated by bytecode programs, to statically enforce access policies for external resources, and to statically determine the overall costs of executing bytecode programs (Chapter 3). We related a resource type system to the MOBIUS-logic, we showed how to handle external policies on MIDP-API-calls and how to analyse costs generically. For alias control, we managed to extend the support of modular reasoning by applying Universe types systems to languages with generics and by permitting multiple owners (Chapter 4).

In summary the contributions of this deliverable improve property coverage, language coverage and flexibility as well as scalability of type systems with respect to Java bytecode.

This deliverable shall serve as a basis for further tasks within MOBIUS. Task 2.2 will build on the typing rules for information flow security by enhancing them to permit declassification. Task 2.4 will build on the typing rules for resources by scaling them up to larger programs and implementing more advanced resource policies. All typing rules will serve as the basis for prototype implementation in Task 2.6 which in turn will be evaluated on the case studies in WP5. Moreover, the typing rules for information flow will be integrated with the logical analysis in Task 3.5, and the alias analysis will be exploited in Task 3.4 for modular verification.

Bibliography

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Principles of Programming Languages*, pages 147–160. ACM Press, 1999.

[2] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Principles of Programming Languages*, pages 104–115. ACM Press, 2001.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[4] J. Agat. Transforming out timing leaks. In *Principles of Programming Languages*, pages 40–53, January 2000.

[5] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation.* PhD thesis, Chalmers University of Technology and Gothenburg University, December 2000.

[6] A. Ahern and N. Yoshida. Formalising Java RMI with explicit code mobility. In R. Johnson and R. P. Gabriel, editors, *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 403–422. ACM Press, 2005. A full version will appear in *Theoretical Computer Science*, special issue of *Global Computing.*

[7] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java bytecode. In *Workshop on Termination*, June 2007.

[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *ESOP 2007* [76], pages 157–172.

[9] 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.

[10] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. A generic framework for the cost analysis of Java bytecode. In *Spanish Conference on Programming and Computer Languages*, September 2007.

[11] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code: A model for mobile code safety. *New Generation Computing*, 2007.

[12] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *ECOOP*, number 3086 in Lecture Notes in Computer Science, pages 1–25. Springer-Verlag, 2004.

[13] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*, 1997.

[14] A. Almeida Matos. *Typing secure information flow: declassification and mobility.* PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2006.

[15] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In G. Morrisett and S. Peyton Jones, editors, *Principles of Programming Languages*, pages 91–102. ACM, 2006.

[16] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java. In *European Conference on Object-Oriented Programming*, pages 124–147, 2006.

[17] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[18] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4), 1998.

[19] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Theorem Proving in Higher-Order Logics*, volume 3223 of *Lecture Notes in Computer Science*, pages 34–49, Berlin, September 2004. Springer-Verlag.

[20] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In Barthe et al. [26], pages 1–26.

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

[22] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

[23] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[24] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [26], pages 151–171.

[25] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Verification, Model Checking and Abstract Interpretation*, number 2934 in Lecture Notes in Computer Science, pages 2–15. Springer-Verlag, 2004.

[26] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[27] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.

[28] G. Barthe, D. Pichardie, and T. Rezk. Non-interference for low level languages. Technical report, INRIA, 2006.

[29] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *ESOP 2007* [76], pages 125–140.

[30] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Types in Language Design and Implementation*, pages 103–112. ACM Press, 2005.

[31] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *European Symposium On Research In Computer Security*, Lecture Notes in Computer Science. Springer-Verlag, September 2007.

[32] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. Technical report, Chalmers University of Technology, 2007. `http://www.cse.chalmers.se/~russo/tissecfull.pdf`.

[33] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Static analysis for stack inspection. *Electronic Notes in Theoretical Computer Science*, 54, 2001.

[34] G. Baxter, M. R. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. D. Tempero. Understanding the shape of Java software. In Peri L. Tarr and William R. Cook, editors, *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 397–412. ACM, 2006.

[35] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2), 2004.

[36] M. Berger. *Towards Abstractions for Distributed Systems.* PhD thesis, Imperial College, Dept. of Computing, 2002.

[37] M. Berger and K. Honda. The two-phase commit protocol in an extended π-calculus. *Electronic Notes in Theoretical Computer Science*, 39, 2000.

[38] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. *J. Funct. Program.*, 17(4-5):473–546, 2007.

[39] M. Berger and N. Yoshida. Timed, distributed, probabilistic, typed processes. In *APLAS*, Lecture Notes in Computer Science, pages 158–174. Springer-Verlag, 2007.

[40] L. Beringer and Martin Hofmann. A bytecode logic for JML and types. In N. Kobayashi, editor, *Programming Languages and Systems: Proceedings of the 4th Asian Symposium, APLAS 2006*, volume 4279 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 2006.

[41] L. Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452, pages 347–362. Springer-Verlag, 2005.

[42] C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In A. Cortesi, editor, *Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2002.

[43] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[44] Fréderic Besson, Guillaume Dufay, and Thomas Jensen. A formal model of access control for mobile interactive devices. In *ESORICS 2006* [77].

[45] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.

[46] W. Binder, J. Hulaas, and A. Villazón. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 139–155, 2001.

[47] E. Bonelli, A. B. Compagnoni, and R. Medel. Information flow analysis for a typed assembly language with polymorphic stacks. In G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3956 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 2005.

[48] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. In *Theoretical Computer Science*, volume 281, pages 109–130, 2002.

[49] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[50] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–230. ACM Press, November 2002.

[51] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages*, pages 213–223, New York, NY, USA, 2003. ACM Press.

[52] M. Bugliesi and M. Giunti. Secure implementations of typed channel abstractions. In *Principles of Programming Languages*, pages 251–262, 2007.

[53] D. Cachera, Thomas P. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Formal Methods Europe*, pages 91–106, 2005.

[54] D. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 2007.

[55] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming*, Lecture Notes in Computer Science, pages 311–325. Springer-Verlag, 2005.

[56] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[57] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–310. ACM Press, 2002.

[58] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 1998. ACM Press.

[59] K. Crary and S. Weirich. Resource bound certification. In *Principles of Programming Languages*, pages 184–198. ACM Press, 2000.

[60] D. Cunningham, S. Drossopoulou, and S. Eisenbach. CUJ: Universe types for race safety. In *1st Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP'2007)*, 2007.

[61] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

[62] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 21–35, 1998.

[63] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.

[64] 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.

[65] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[66] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A distributed object oriented language with session types. In *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 299–318, 2005.

[67] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of generic universe types. Technical Report 532, ETH Zurich, 2006.

[68] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In E. Ernst, editor, *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 28 – 53. Springer-Verlag, 2007.

[69] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In *ACM Workshop on Foundations of Object-Oriented Languages*, January 2007.

[70] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Workshop on Formal Techniques for Java Programs*, pages 49–54, 2004.

[71] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[72] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[73] S. Drossopoulou, A. Francalanza, and P. Müller. A unified framework for verification techniques for object invariants (full paper). `http://research.microsoft.com/~mueller/publications.html`, 2007.

[74] G. Dufay, A. P. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, 2005.

[75] B. Emir, A. J. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In Dave Thomas, editor, *European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer-Verlag, 2006.

[76] *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.

[77] *Computer Security — ESORICS 2006, Proceedings of the 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18–20, 2006*, number 4189 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[78] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Languages Design and Implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[79] C. Flanagan and S. Qadeer. Types for atomicity. In *Types in Language Design and Implementation*. ACM Press, 2003.

[80] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Programming Languages Design and Implementation*, pages 237–247, 1993.

[81] 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.

[82] 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.

[83] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer-Verlag, 1999.

[84] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *Symposium on Secure Software Engineering*. IEEE Press, 2006.

[85] C. Hankin, F. Nielson, and H. R. Nielson. *Principles of Program Analysis*. Springer-Verlag, 2005. Second Ed.

[86] D. Hedin and D. Sands. Timing aware information flow security for a Java Card-like bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science, 2005.

[87] N. Heintze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Principles of Programming Languages*, pages 365–377. ACM Press, 1998.

[88] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction carrying code and resource-awareness. In *Principle and Practice of Declarative Programming*. ACM Press, July 2005.

[89] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program development using abstract interpretation (and the ciao system preprocessor). In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 127–152. Springer-Verlag, 2003.

[90] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[91] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Principles of Programming Languages*, pages 81–92. ACM Press, 2002.

[92] K. Honda and N. Yoshida. A uniform type structure for secure information flow. *ACM Transactions on Programming Languages and Systems*, 29(6):101 pages, 2007.

[93] K. Honda, N. Yoshida, and M. Berger. Control in the π-calculus. In *ACM SIGPLAN Continuation Workshop*. ACM Press, 2004.

[94] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Logic in Computer Science*, pages 270–279, 2005.

[95] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Computer Security Foundations Workshop*, 2006.

[96] J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 169–177, 2004.

[97] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 34(10), pages 132–146, 1999.

[98] Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of control flow based security properties. In *Proceedings of the 20th IEEE Symp. on Security and Privacy*, pages 89–103. New York: IEEE Computer Society, 1999.

[99] JSR 205 Expert Group. Wireless messaging API (version 2.0). Java specification request, Java Community Process, June 2003.

[100] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *Programming Languages Design and Implementation*, pages 1–12, 2001.

[101] J. Knudsen. Networking, user experience, and threads. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/articles/threading/`, 2002.

[102] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[103] B. Köpf and H. Mantel. Eliminating Implicit Information Leaks by Transformational Typing and Unification. In T. Dimitrakos, F. M. elli, P. Y. A. Ryan, and S. Schneider, editors, *Workshop on Formal Aspects in Security and Trust*, Lecture Notes in Computer Science, pages 47–62, Newcastle, UK, July 2006. Springer-Verlag.

[104] B. Köpf and H. Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security*, 2007.

[105] T. Kraußer, H. Mantel, and H. Sudbrock. A probabilistic justification of the combining calculus under the uniform scheduler assumption. Technical Report 2007-09, RWTH Aachen, May 2007.

[106] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004. Available from `www.sct.inf.ethz.ch/publications/index.html`.

[107] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 144–153, 1998.

[108] Yi Lu and J. Potter. Protecting representation with effect encapsulation. In *Principles of Programming Languages*, pages 359–371, 2006.

[109] Q. H. Mahmoud. Preventing screen lockups of blocking operations. Sun Technical Articles and Tips `http://developers.sun.com/techtopics/mobility/midp/ttips/screenlock/`, 2004.

[110] P. Maier, D. Aspinall, and I. Stark. Explicit accounting of resources using resource managers. Technical Report EDI-INF-RR-0859, The University of Edinburgh, October 2006.

[111] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *ESOP 2007* [76], pages 141–156.

[112] H. Mantel and A. Sabelfeld. A Generic Approach to the Security of Multi-threaded Programs. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 126–142, Cape Breton, Nova Scotia, Canada, 2001. IEEE Press.

[113] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-threaded Programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[114] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Asian Programming Languages and Systems Symposium*, LNCS 3303, pages 129–145, Taipei, Taiwan, November 2004. Springer-Verlag.

[115] H. Mantel, H. Sudbrock, and T. Kraußer. Combining Different Proof Techniques for Verifying Information Flow Security. In G. Puebla, editor, *Logic-based Program Synthesis and Transformation*, number 4407 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[116] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[117] R. Medel, A. B. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In *Italian Conference on Theoretical Computer Science*, volume 3701 of *Lecture Notes in Computer Science*, pages 360–374. Springer-Verlag, 2005.

[118] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Towards execution time estimation for logic programs via static analysis and profiling. In *Workshop on Logic Programming Environments*, page 16, August 2006.

[119] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Using combined static analysis and profiling for logic program execution time estimation. In *International Conference on Logic Programming*, number 4079 in Lecture Notes in Computer Science. Springer-Verlag, August 2006.

[120] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining static analysis and profiling for estimating execution times. In M. Hanus, editor, *Symposium on Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, January 2007.

[121] J. K. Millen. A resource allocation model for denial of service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 137–147. IEEE Computer Society Press, 1992.

[122] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming*, pages 74–98, 2006.

[123] MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements, 2006. Available online from `http://mobius.inria.fr`.

[124] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[125] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, November 1999.

[126] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, 1993.

[127] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[128] P. Müller. Reasoning about object structures using ownership. In *Verified Software: Theories, Tools, Experiements*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[129] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999. Technical Report 263, Available from `sct.inf.ethz.ch/publications`.

[130] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[131] P. Müller and A. Rudich. Formalization of ownership transfer in universe types. Technical Report 556, ETH Zurich, 2007.

[132] P. Müller and A. Rudich. Ownership transfer in universe types. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007. To appear.

[133] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages*, pages 228–241. ACM Press, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.

[134] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *International Conference on Functional Programming*, pages 62–73. ACM Press, 2006.

[135] D. Naumann. Verifying a secure information flow analyzer. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher-Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 211–226. Springer-Verlag, 2005. Preliminary version appears as Report CS-2004-10, Stevens Institute of Technology, 2003.

[136] David A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS 2006* [77], pages 279–296.

[137] 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.

[138] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[139] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In Eric Jul, editor, *European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 158–185. Springer-Verlag, 1998.

[140] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conference*, pages 1–20, 2006.

[141] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [124], 2006.

[142] A. Potanin, J. Noble, D. Clarke, and Robert Biddle. Generic ownership for generic java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2006. ACM Press.

[143] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in object-oriented programs. *Communications of the ACM*, May 2005.

[144] 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.

[145] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In Mooly Sagiv, editor, *European Symposium on Programming*, pages 77–93, 2005.

[146] A. Reinhard. Analyse nebenläufiger programme unter intransitiven sicherheitspolitiken. Master's thesis, RWTH Aachen, May 2006.

[147] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[148] M. Rosendahl. Automatic complexity analysis. In *Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. Association of Computing Machinery, 1989.

[149] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Computing Science Conference*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[150] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Computer Security Foundations Workshop*, pages 177–189. IEEE Press, 2006.

[151] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Andrei Ershov International Conference on Perspectives of System Informatics*, volume 4378 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2006.

[152] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *Journal of Logic and Algebraic Programming*, 2007. Special Issue dedicated to the Nordic Workshop on Programming Theory.

[153] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 225–239. Springer-Verlag, 2001.

[154] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, 2003.

[155] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 376–394. Springer-Verlag, 2002.

[156] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, 2003.

[157] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In D. Swiestra, editor, *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 40–58, 1999.

[158] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop*, pages 200–215. IEEE Press, 2000.

[159] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations Workshop*, pages 255–269. IEEE Press, 2005.

[160] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking and Abstract Interpretation*, pages 199–215, 2005.

[161] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[162] G. Smith. A new type system for secure information flow. In *Computer Security Foundations Workshop*, pages 115–125, June 2001.

[163] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Computer Security Foundations Workshop*, pages 3–13, 2003.

[164] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Principles of Programming Languages*, pages 355–364, 1998.

[165] M. Smith. *A Model of Effects with an application to Ownership Types.* PhD thesis, Imperial College, 2007.

[166] F. Spoto. JULIA: A generic static analyser for the Java bytecode. In *Workshop on Formal Techniques for Java Programs*, 2005.

[167] F. Spoto, P. M. Hill, and E. Payet. Path-length analysis for object-oriented programs. In *Emerging Applications of Abstract Interpretation*, 2006.

[168] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 2004.

[169] Sun Microsystems, Inc., Palo Alto/CA, USA. *Mobile Information Device Profile (MIDP) Specification for Java 2 Micro Edition, Version 2.0*, 2002.

[170] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[171] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *IEEE Computer Security Foundations Symposium*, July 2007. `http://www.cse.chalmers.se/~russo/`.

[172] J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

[173] D. Volpano and G. Smith. A type-based approach to program security. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.

[174] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Computer Security Foundations Workshop*, pages 34–43, Rockport, Massachusetts, June 1998. IEEE Press.

[175] H. S. Wilf. *Algorithmics and Complexity.* A. K. Peters Ltd, 2002.

[176] R. Wilhelm. Timing analysis and timing predictability. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323. Springer-Verlag, 2004.

[177] T. Wrigstad and D. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4):141–159, 2007.

[178] N. Yoshida, K. Honda, and M. Berger. Linearity and bisimulation. *Journal of Logic and Algebraic Programming*, 72:207–238, 2007.

[179] Nobuko Yoshida. Type-based security for mobile computing integrity, secrecy and liveness. *ENTCS*, 162:333–340, 2006.

[180] Dachuan Yu and Nayeem Islam. A typed assembly language for confidentiality. In *Programming Languages and Systems: Proceedings of the 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, 2006.

[181] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

[182] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, USA, June 2003. IEEE Press.