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 D2.1

Intermediate Report on Type Systems

Due date of deliverable: 2006-08-31 (T0+12)

Actual submission date: 2006-10-09

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

Organisation name of lead contractor for this deliverable: LMU

Revision — Final

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

Executive Summary:
Intermediate Report on Type Systems

This document summarises deliverable D2.1 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 intermediate results that were obtained during the first year 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). We have developed type systems for properties in all three domains, both at the level of bytecode and at higher language levels. We motivate the usage of type systems for facilitating proof generation as is required for a proof-carrying-code scenario. In order to prepare for the correspondong work in WP4, an important aspect of our work in the remaining time for the present tasks will be to formally relate the presented type systems to the MOBIUS logic and the Bicolano model of the Java Virtual Machine.

Contents

Chapter 1

Introduction

This deliverable reports on initial progress on type systems for Mechanisms for Safe Information Release (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, RWTH, CTH, IC, LMU, UEDIN, UPM, and ETH.

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.

Property-independent certification, on the other hand, is afforded by formalised program logics and operational semantics, see D3.1. Concrete certifications in program logic or proofs in a formalised meta logic about operational semantics are very cumbersome to produce directly. In MOBIUS, we therefore proposed to map typing derivations automatically into program logics or meta proofs about operational semantics and therefore combine the highest possible degree of trustworthiness and independent verifiability with state-of-the-art automation.

While these translations form the subject of WP4 we have asked the task leaders who have contributed to this deliverable to describe the respective type systems in such a way that the possibility of a translation into certifiable logic becomes apparent.

Chapter 2 describes a type system for flow-sensitive maintenance of level-based security policies for the JVM. While the central ideas of the type system were already known and have to a large extent been developed by members of the consortium which comprises leading players in this field, the main contribution consists of the adaptation and considerable extension to the JVM, notably objects and exceptions. In this way, type-based information security becomes applicable within MOBIUS.

Chapter 3 describes various type systems for controlling resource, notably execution time, heap space, peripheral resources, and access. Again, pre-existing work from within the consortium had to be considerably adapted and extended to fit the MOBIUS requirements, in particular the translatibility into program logic. The type system on access control has been designed from scratch for MOBIUS.

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. Accordingly, the issue of translatability into program logic is not yet covered in this document.

In a nutshell, alias control introduces the ownership relation between objects and guarantees that certain kinds of access 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

The Java security architecture combines static and dynamic mechanisms to enforce innocuity of applications; in particular, it features a bytecode verifier that guarantees statically safety properties such as the absence of arithmetic on references, and a stack inspection mechanism that performs access control verifications. However, it lacks of appropriate mechanisms to guarantee stronger confidentiality properties. One weakness of the model is that it only concentrates on who accesses sensitive information, but not how sensitive information flows through programs.

The purpose of this chapter is to present a type system to enforce confidentiality of object-oriented applications executing on a Java-like virtual machine, and to show that the type system enforces non-interference, a baseline information-flow policy that is increasingly considered in the context of language-based security.

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 (as a simplifying assumption, we assume that all methods, and in particular the main method of a program, return a value).

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 [9].

Virtual machine Our type system applies to a stack-based virtual machine that is similar to the Java Virtual Machine in several respects; in particular, our machine features 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 [74]. The most significant differences with the Java Virtual Machine are the assumption of an unbounded memory for the heap, and the lack of arrays, multi-threading, and subroutines. We will discuss these limitations when presenting the 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 [9] and [10], but solves a number

[1]Information flow security requirements relevant to global computing have been specified in MOBIUS deliverable 1.1 [66].

[2]Proofs can be consulted in a companion report [12]

of non-trivial difficulties due to the complexity of the language.

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 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 hang. 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 [74] 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 $k_{\mathrm{obs}} \in \mathcal{S}$ 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. $\mathsf{ft} f \leq k_{\mathrm{obs}}$; the formal definition of heap indistinguishability is rather involved and deferred to Section 2.3.2;
- 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 level of the method local variables (including methods 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 level of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, 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 $\mathbf{k_r}[n]$ 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,

[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.

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

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 hang; 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.

2.2 Type system

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

2.2.1 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 [**?**, 9]. For unstructured low level code, such as Java bytecode, implicit flows can be tracked using extra security annotations.

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 [13, 76]. 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).

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

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

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

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

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

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

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

Figure 2.1: Successor relation

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 $c \in \mathcal{C}$ for a step where an exception of class C 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.

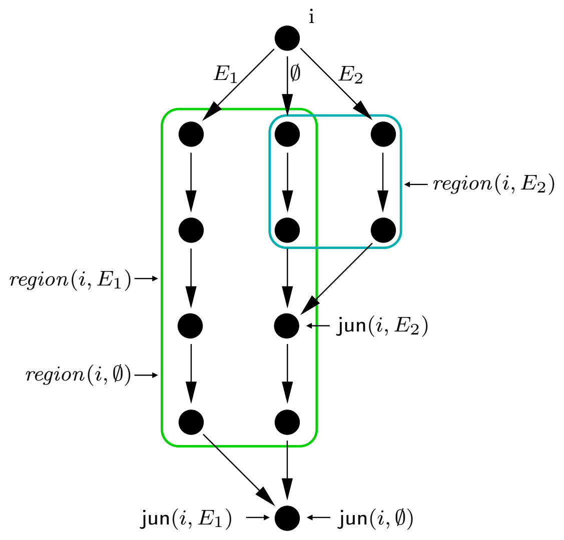

Figure 2.2: Example of cdr. Only relevant tags are presented here.

2.2.2 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, 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 [9] 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 [12].

Below we comment this selection of rules:

[5]Associating signatures with method identifier instead of method allows to enforce that overriding of a method preserve its declared security signatures.

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

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

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \mathbf{k'_a} \xrightarrow{k'_h} \mathbf{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \mathbf{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \mathbf{k'_a}[i+1] \\ k_e = \bigsqcup \{\ \mathbf{k'_r}[e] \mid \exists e \in \mathsf{excanalysis}(m_{\mathrm{ID}}), \exists t \in \mathcal{PP}, \mathsf{Handler}(i,e) = t\ \} \\ \forall j \in region(i,\emptyset),\ k \sqcup k_e \leq se(j) \end{array}}{\Gamma, region, se, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{\emptyset} st_1 :: k :: st_2 \Rightarrow \mathrm{lift}_{k \sqcup k_e}\ ((\mathbf{k'_r}[n] \sqcup se(i)) :: st_2)}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \mathbf{k'_a} \xrightarrow{k'_h} \mathbf{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \mathbf{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \mathbf{k'_a}[i+1] \\ e \in \mathsf{excanalysis}(m_{\mathrm{ID}}) \qquad \forall j \in region(i,e),\ k \sqcup \mathbf{k'_r}[e] \leq se(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, region, se, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \mathbf{k'_r}[e]) :: \varepsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \mathbf{k'_a} \xrightarrow{k'_h} \mathbf{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \mathbf{k'_a}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \mathbf{k'_a}[i+1] \\ e \in \mathsf{excanalysis}(m_{\mathrm{ID}}) \qquad k \sqcup \mathbf{k'_r}[e] \leq \mathbf{k_r}[e] \qquad \forall j \in region(i,e),\ k \sqcup \mathbf{k'_r}[e] \leq se(j) \qquad \mathsf{Handler}(i,e) \uparrow \end{array}}{\Gamma, region, se, \mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow}$$

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

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

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

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

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

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

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

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

Figure 2.3: Selected typing rules

- The typing rule for ifeq requires that the result stack type is lifted 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 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 $\mathbf{k_r}[n]$. In addition, the rule requires that the value on top of the operand stack has a security level above $\mathbf{k_r}[n]$, since it will be observed by the attacker.
- 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 ($k_h \leq k_{\mathrm{obs}}$), 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 $(k'_r[n] \sqcup se(i))$. $k'_r[n]$ correspond to the level of the return value in the context of th called method. $se(i)$ prevent 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.
- 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.
- 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.

2.2.3 Typable programs

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

- if $i \mapsto_m$ then $\Gamma, \mathit{region}, \mathit{se}, \mathrm{sign}, i \vdash^\tau S_i \Rightarrow$.

2.2.4 Typable example

The following method may throw two kind of exception: 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 because execution terminates normally only if it is not *null*.

```
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

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

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

i	$\mathit{region}(i,\cdot)$	$\mathsf{jun}(i,\cdot)$	S_i	$\mathit{se}(i)$
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

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 $\rightarrow 3$

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

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

with

$\mathcal{N}$	:	the set of numerical values
$\mathcal{L}$	:	the set of locations
$\mathcal{X}$	:	the set of variable names
$\mathcal{C}$	:	the set of class names

$\mathcal{F}$ the set of field names

Figure 2.4: MEMORY MODEL OF THE JVM

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:

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

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

2.3 Non interference theorem

2.3.1 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 $(\langle v\rangle_{\rm V}, h) \in \mathcal{V} \times$ Heap (normal termination), or a pair $(\langle l\rangle_{\rm e}, h) \in \mathcal{L} \times$ Heap (the method execution terminates because of an exception thrown on an object pointed by a location l, but not caught in this method).

2.3.2 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 [9]. 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 2.3.1 (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 $v_1 \sim^{\mathcal{V}}_{\beta} v_2$ is defined by the clauses:*

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

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

Value indistinguishability is extended point wise to local variable maps (for low variables, *i.e.* local with low security levels according to global policy ft).

Definition 2.3.2 (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 $x \in \mathcal{X}$, $\mathbf{k_v}(x) \le k_{\mathrm{obs}} \Rightarrow \rho_1(x) \sim^{\mathcal{V}}_{\beta} \rho_2(x)$. We denote this fact: $\rho_1 \sim^{\mathrm{LocalVar}}_{\beta,\mathbf{k_v}} \rho_2$*

The definition of object indistinguishability is similar.

Definition 2.3.3 (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 $f \in \mathsf{dom}(o_1)$, $\mathsf{ft}(f) \le k_{\mathrm{obs}} \Rightarrow o_1(f) \sim^{\mathcal{V}}_{\beta} o_2(f)$.*

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 2.3.4 (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 2.3.5 (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:*

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

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

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

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

2.3.3 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 [74].

Definition 2.3.6 (Secure method) *A method m (in a program p) is said* secure with respect to a signature $\mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}$ *if and only if for all arrays of local variables $\rho_1, \rho_2 \in$* LocalVar*, for all heaps $h_1, h_2 \in$* Heap*, for all final states $(ret_1, h'_1), (ret_2, h'_2) \in$* FinalState*, for all type annotation* ft *for fields and for all partial function $\beta \in \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\rho_1 \sim^{\rm LocalVar}_{\beta,\mathbf{k_r}} \rho_2$, $h_1 \sim^{\mathsf{Heap}}_{\beta,\mathsf{ft}} h_2$, $\langle h_1, 0, \rho_1, \varepsilon\rangle \Downarrow_m (ret_1, h'_1)$ 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 $\mathbf{k_r}$: $(ret_1, h'_1) \sim^{\rm FinalState}_{\beta',\mathsf{ft},\mathbf{k_r}} (ret_2, h'_2)$;*
- *no modification under level k_h is done on h_1 and h_2: for all field $f \in \mathcal{F}$ such that $k_h \not\le \mathsf{ft}(f)$,*
  - *for all location $l \in \mathcal{L}$ such that $h_1(loc).f$ is defined then $h'_1(loc).f$ is defined and equal to $h_1(loc).f$;*
  - *for all location $l \in \mathcal{L}$ 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 signature of a method m is defined as $\rm Policies_\Gamma(m) = \{\ \Gamma_m[k] \mid k \in \mathcal{S}\ \}$. We use it to define the notion of *safe program.*

Definition 2.3.7 (Non-interfering program) *A program is* safe *with respect to a table of method signature Γ if for all its method m, m is safe with respect to all policies in* $\rm Policies_\Gamma(m)$*.*

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 1 *Let m a method (in a program p) returning a numerical value and which is secure with respect to a signature $\mathbf{k_a} \xrightarrow{k_h} \mathbf{k_r}$. Let h_0, h_1, h_2 some heaps, ρ_1, ρ_2 two arrays of local variables such that for all variable x, $\mathsf{ft}(x) \le k_{\rm obs}$ implies $\rho_1(x) = \rho_1(x)$ (parameter 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 $\mathbf{k_r}[n] \le k_{\rm obs}$, both returned values are equals: $n_1 = n_2$.

2.3.4 Type system soundness

Theorem 2.3.8 *Let p a program and Γ a table of signature, 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 a companion report [12].

2.4 Related work

We refer to the survey article of Sabelfeld and Myers [81] 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. Then, we focus on issues of concurrency and information release that are not considered in the work presented. Ongoing work and future objectives are further discussed in Section 2.5.

2.4.1 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 [68] 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 [**?**], 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 [62] and by Yu and Islam [93] avoids the use of pseudo-instructions. In addition, Yu and Islam consider a richer assembly language and prove type-preserving compilation for an impreative language with procedures.

2.4.2 Higher-order low-level languages

Zdancewic and Myers [94] 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 [44] 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 [48] 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 [88]. 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.

2.4.3 JVM

Lanet *et al.* [22] 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 [18], 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 [10] 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 of this paper, and show type-preserving compilation for the imperative language and type system of [88]. Later, Barthe and Rezk [13] extend this work to a language with objects and a simplified treatment of exceptions, and Barthe, Naumann and Rezk [11] show type-preserving compilation for a Java-like language with objects and a simplified treatment of exceptions.

Genaim and Spoto [41] 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.

2.4.4 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 [9] 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 is 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 [72], and [86] present a type inference algorithm that dispenses users of writing all security annotations.

More recently, Hammer, Krinke and Snelting [43] 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 [70] and of [9] 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 unvalidated 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 its points to.

2.4.5 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 [29] 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 [35], who used an extension of the Krakatoa tool [61] 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 [29, 35] are application-oriented and do not attempt to provide a theoretical study of self-composition for Java. In a recent article, Naumann [73] 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 [5] 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.

2.4.6 Concurrency

Extending information flow type systems to concurrent languages is notoriously difficult because the parallel composition of secure sequential programs may itself not be secure [85]. The source of the problem is that races lead to non-determinism in a program's behavior, at least for intermediate states in an execution. Non-determinism, however, is in conflict with security definitions that require a program to lead to indistinguishable states given that the starting states are indistinguishable. Another problem is caused by the so-called internal timing leaks, which allow an attacker to exploit differences in the timing behavior of a program, even if he does not have access to a clock. An additional difficulty is that language definitions are usually parametric in the scheduler, while the presence of internal timing leaks closely depends on scheduling.

Work on type systems for information flow security often focuses on bisimulation-based notions of security that require indistinguishability for intermediate execution steps. The resulting security conditions prevent information leakage, permit concurrency, support a compositional analysis and are robust in the choice of the scheduler (e.g., [82, 80]), but they are over-restrictive in the sense that they reject many programs that are intuitively secure. Type systems for such security definitions reject even more intuitively secure programs, because being type correct constitutes a conservative approximation of the already restrictive security conditions. As a consequence, some secure programs are rejected solely because they, e.g., contain a loop with a high guard or perform a low assignment after a high branching statement.

Motivated by the desire to provide flexible and practical enforcement mechanisms for concurrent languages, recent work addressed the above limitations. One can identify three major research directions, depending on whether (1) the type system, (2) the programming language, or (3) the security condition is modified. The Mobius partners explored and advanced each of these directions:

1. In order to eliminate differences in the timing behavior between alternative execution paths, the cross-copying technique [?, 82] pads each branch with skip commands that emulate the timing behavior of the respective other branch. This was a first solution to permitting high guards in conditionals, without the protect statements needed in earlier type systems [89]. Köpf and Mantel further improve this approach by replacing the cross-copying technique with a unification-based transformation [50].

   Ultimately, the underlying security condition limits what one can achieve by solely improving the type system. Unless one were willing to sacrifice soundness, any intuitively secure program that does not satisfy the given security condition must also be rejected by the type system. Similarly, the scope of transformations is limited as they can only support the programmer in making programs type correct.

2. Since language definitions are usually parametric in the specification of the scheduler, it is desirable to make the information flow analysis robust with respect to the particular choice of a scheduler.

However, without making any assumptions about the class of possible schedulers, one arrives at rather restrictive security definitions like the strong security condition [82], as demonstrated in [79].

One can constrain scheduling decisions by adding explicit scheduling directives to a program, which can then be exploited in the security analysis in order to accept more programs that are intuitively secure. Russo and Sabelfeld introduce two commands to lift a thread from public to secret, or to make a temporarily secret thread public again, and require that the scheduler does not pick any public thread for execution as long as a temporarily secret thread is executing. They develop a sound information flow type system that enforces termination-insensitive non-interference in a concurrent setting [78].

3. Scheduler-independent security conditions for concurrent languages are usually either bisimulation based or determinism based. While the first approach leads to over-restrictive constraints on the flow of information (typically excluding, e.g., loops with high guards), the second approach limits concurrency by requiring that a program's behavior is deterministic in its low part [95].

   Mantel, Sudbrock and Araußer propose an approach to using security type systems following these traditions in combination when analyzing a given program [60]. The combining calculus provides typing rules for a compositional analysis and plugin rules for different information flow analyses.

2.4.7 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 [83], 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 [77].

2.5 Foreseen improvements

The work presented in Sections 2.1–2.3 does not yet address all aspects of the bytecode language. On-going and future efforts (in Task 2.1 and Task 2.2) aim at overcoming the most important, remaining limitations.

2.5.1 Multi-threading

An investigation of existing security type systems for multi-threaded languages revealed that the available approaches were not yet suitable for a practical information flow analysis of bytecode programs. As outlined in Section 2.4.6, three research directions were explored in order to determine an approach that is suited for the goals of the project. An outcome of these investigations is that it is most promising to accompany improvements to the security type system by improvements of the underlying security condition. This will be the approach taken to extend the security type system to concurrent programs. We plan to adapt the two solutions proposed in [78, 60] to the bytecode language. These solutions are complementary: the first achieves greater precision by exploiting scheduler directives in a program while the second is applicable also for the usual scheduler implementations that do not yet support such scheduler directives.

2.5.2 Distributedness and fault tolerance

The work will start by augmenting Java byte code with a primitive for RMI invocation and develop a typing system guaranteeing secure information flow. This work will include a basic notion of process failure. The uniform type structure for secure information flow developed in [48] will form the basis of this work, but needs to be extended to deal with failures. Basic notions of failure will be modelled following [16]. The key problem is to investigate the precise effect of failures on information flow.

2.5.3 Bounded memory, arrays and subroutines

Information flow analysis without bounded memory assumption requires to mix this work with a memory usage analysis. Resource analysis is the subject of the next chapter of this deliverable.

An extension of our type system to arrays is proposed in the companion report [12]. Special care has been taken to allow public arrays handling secret informations as it is done at source level in Jif. In their case study [?], Askarov and Sabelfeld show that such a mechanism is important to type realistic programs.

We do not plane to handle subroutines for our type-system. We have already worked on an extension, but without soundness proofs at the moment. We feel it is not necessary to continue in this direction because subroutines are currently disappearing in Java (and are already missing in MIDP).

2.5.4 Declassification

Security-critical systems usually rely on some possibility to intentionally release secrets. This makes the integration of declassification a key step to enabling technology transfer into standard security practice [81].

In Task 2.2, we plan to develop policies and mechanisms for safe information release that are adequate for expressing and enforcing intentional declassification. We will investigate the three dimensions of information release: *what* information is released, *who* controls information release, and *where* in the system information is released [59]. Most research has focused on one of these dimensions in isolation, but it appears essential to also study the relationship between the different dimensions and possibilities for enforcing them in combination. Currently, little is known about the relationship between different aspects of information release. This creates hazardous situations where policies provide only partial assurance that information release mechanisms cannot be compromised. For example, consider a policy for describing “what” information is released. This policy stipulates that a limited set of the digits of a credit card number might be released when a purchase is made (as often needed for logging purposes). This policy specifies “what” can be released but says nothing about ”who” controls which of the numbers are revealed. Leaving this information unspecified leads to an attack where the attacker launders the entire credit card number by asking to reveal different digits under different purchases. Beyond a flexible composition of security policies, the goal is also to automate their enforcement at the level of programming languages. Little has been done for enforcing information release policies for bytecode languages. It is the objective of Task 2.2 to enhance security type systems developed under Task 2.1 with possibilities for intentional information release.

Chapter 3

Types for Basic Resource Policies

In this section, we summarise work on type systems for resources. In the first year, work has concentrated on basic formalisms for heap space (Section 3.1), static analysis of resource managers (Section 3.3), access permissions of MIDlets (Section 3.2), and a framework which combines cost analysis with profiling techniques in order to infer functions which yield bounds on platform-dependent *execution times* of procedures (Section 3.4). Future work will aim to generalise and extend the presented work to more complex policies and patterns of resource access, and interprete selected type systems in the program logic and the verification infrastructure developed in WP3.

3.1 Analysis of heap space consumption

3.1.1 Type system for constant heap space

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 [23] 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 [23] is phrased as an abstract interpretation. In addition, Cachera et. al.'s approach involves the formalisation of the calculation of the program representation (control flow graph) and of the inference algorithm (fixed point iteration) in the theorem prover. In contrast, our presentation separates the algorithmic issues (type inference and checking) from semantic issues (the property expressed or guaranteed) as is typical for a type-based formulation. Depending on the verification infrastructure available at the code consumer side, the PCC certificate may either consist of (a digest of) the typing derivation or an expansion of the interpretation of the typing judgements into the MOBIUS logic. The latter approach was employed in our earlier work [17] and consists of understanding typing judgements as derived proof rules in the program logic and using syntax-directed proof tactics to apply the rules in an automatic fashion. In contrast to [17], 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. We prove that code resulting from compiling program written in this language into bytecode satisfies the bound asserted by a high-level type system: derivability in the high-level type system guarantees derivability in the bytecode level type system. The whole presentation is based on a representation in a theorem prover, for a variant of the MOBIUS program logic that differs from the one described in [67] in minor ways. More precisely,

- only a subset of bytecode instructions is considered, with a simplified syntax for jump operations. Apart from basic instructions (arithmetic, operand and object-manipulating instructions,. . .) we treat jumps and static method invocations with a single argument. The subset sufficies for translating our high-level language, but future work will seek to extend the formalism to the remaining instructions covered by Bicolano. This should not be too difficult: Array operations do not access heap. The rules for exception handlers will be similar to the rules for jumps. For method-internal recursion we expect that the context-based technique employed in the bytecode logic (see [67]) may be lifted to types, and corresponds to the traditional high-level mechanism for dealing with recursive program structures. Finally, the behavioural subtyping condition on virtual methods will amount to the requirement that overriding methods must not consume more memory than the overridden method.

- neither the operational model nor the program logic contain exceptions, i.e. the corresponding side conditions have been deleted from the proof rules. The incorporation of explicit exceptions into the type system will involve a rule for instruction Throw (which allocates one memory cell, for the construction of the exception object), and rules for handling exceptions. The latter rules will be similar to the rules for jump operations. However, the fact that almost all bytecode instructions may implicitly raise exceptions (e.g. NonNullException) will introduce further side conditions in may of the rules given below, unless additional program analysis is present that guarantees the absence of such exceptions.
- the last component of method invariants and local invariants are of type *heap* instead of type *state*. This reflects the fact that operand stacks and local variables of subframes are of little interest outside the frame, and simplifies the presentation of the type system's interpretation. This difference is a simple implementaion difference between the logics, and the conversion into the judgement form of the bytecode logic as presented in [67] is trivial.

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 judgements of the form $\vdash_\Sigma pc : n$, expressing that the segment of bytecode whose initial instruction is located at pc is guaranteed not to allocate more than n memory cells. Here, signature Σ assigns types (natural numbers n) to identifiers of (static) methods. The rules are as follows.

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

$$\text{C-Instr}\ \frac{basic(m,l) \quad \neg P(m,l) = \mathsf{New}\ C \quad \neg P(m,l) = \mathsf{Athrow} \quad \vdash_\Sigma m, suc_m(l) : n}{\vdash_\Sigma m, l : n}$$

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

$$\text{C-Invs}\frac{P(m,l) = \mathsf{Invokestatic}\ m' \quad m' \in \mathsf{dom} P \quad \vdash_\Sigma m, suc_m(l) : n \quad \Sigma(m') = k}{\vdash_\Sigma m, l : n+k} \qquad \text{C-Sub}\ \frac{\vdash_\Sigma pc : n \quad n \leq k}{\vdash_\Sigma pc : k}$$

The first rule, C-New, asserts that the memory consumption of a code fragment whose first instruction is New C is the increment of the remaining code. Rule C-Instr applies to all basic instructions (in the case of Goto l' we take $suc_m(l)$ to be l'), except for New C – the predicate $basic(m,l)$ is defined as

$$basic(m,l) \equiv P(m,l) \in \left\{ \begin{array}{l} \mathsf{Iload}\ x, \mathsf{Istore}\ x, \ldots \mathsf{Const}\ t\ z, \mathsf{Ibinop}\ o, \ldots, \mathsf{New}\ C, \\ \mathsf{Getfield}\ F, \mathsf{Putfield}\ F, \mathsf{Getstatic}\ F, \mathsf{Putstatic}\ F, \ldots, \mathsf{Athrow}, \mathsf{Goto}\ l' \end{array} \right\}.$$

in [67]. The memory effect of these instructions is zero, as is the case for return instructions, conditionals, and (static) method invocations.

We call P *well-typed* for Σ, notation $\vdash_\Sigma P$, if for all m and n, $\Sigma(m) = n$ implies $\vdash_\Sigma m, init_m : n$.

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 [67]. 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 invariant 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 $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$.

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,
3. I is a (local) invariant, i.e. a predicate $I(s, s')$ that relates the state s at pc to 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 $\mathsf{G}, \mathsf{Q} \vdash \{A_0\}\, m, init_m\, \{B_0\}\, (I_0)$ 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 satisifies 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.

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

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

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

Finally, we say that M satisfies Σ, notation $\mathsf{M} \models \Sigma$, if for all m and n, $\Sigma(m) = n$ holds exactly if $\mathsf{M}(m) = (\mathit{Spec}\ n, \emptyset, \emptyset)$. Thus, we require proof contexts G and 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 1 *Let* $\mathsf{M} \models \Sigma$, $m \in \mathsf{dom}\Sigma$ *and* $\vdash_\Sigma m, l : n$. *Then* $\emptyset, \emptyset \vdash m, l\ \{n\}$.

Based on this result, the fact that the bahavioural 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 3.1.1 *Let* $\mathsf{M} \models \Sigma$ *and* $\vdash_\Sigma P$. *Then* $\mathsf{M} \vdash P$.

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 [7]. The syntax is stratified into primitive expressions and general expressions similar to the A-normal form discipline [40]. 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.

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

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 3.1 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.

Definition 3.1.2 *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* $\Sigma \rhd e : \Sigma(m)$.

$$\textsc{T-int}\ \frac{}{\Sigma \rhd i : 0} \qquad \textsc{TP-un}\ \frac{}{\Sigma \rhd \mathsf{uop}\ u\ x : 0} \qquad \textsc{TP-bin}\ \frac{}{\Sigma \rhd \mathsf{bop}\ o\ x\ y : 0}$$

$$\textsc{TP-nil}\ \frac{}{\Sigma \rhd \mathsf{Nil} : 0} \qquad \textsc{T-cons}\ \frac{}{\Sigma \rhd \mathsf{Cons}(x, y) : 1} \qquad \textsc{T-call}\ \frac{\Sigma(m) = n}{\Sigma \rhd m(x) : n}$$

$$\textsc{T-prim}\ \frac{\Sigma \rhd p : n}{\Sigma \rhd \mathsf{prim}\ p : n} \qquad \textsc{T-let}\ \frac{\Sigma \rhd p : n \quad \Sigma \rhd e : k}{\Sigma \rhd \mathsf{let}\ x = p\ \mathsf{in}\ e : n + k} \qquad \textsc{T-sub}\ \frac{\Sigma \rhd e : k \quad k \leq n}{\Sigma \rhd e : n}$$

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

Figure 3.1: High-level typing rules

Figure 3.2 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 M 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 the $m \in \mathsf{dom}P$ and the implementation $P(m)$ is $([\![e]\!]_l^{[]}, init_m)$.

Type soundness for primitive expressions justifies the high-level typing judgements by showing the deriviability 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 2 *If* $\Sigma \rhd p : n$, $[\![p]\!]_l^C = (C_1, l_1)$, $\mathsf{dom}\Sigma \subseteq \mathsf{dom}P$, *and* $\vdash_\Sigma m, l_1 : k$, *then* $\vdash_\Sigma m, l : n + k$.

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

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

Combining this result with Theorem 3.1.1 yields the final soundness result.

Theorem 3.1.3 *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

$$
\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 & = & (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 & = & \left(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\right) \\
[\![m(x)]\!]_l^C & = & (C[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{Invokestatic}\ m], l+2) \\
[\![\mathsf{prim}\ p]\!]_l^C & = & let\ (C_1, l_1) = [\![p]\!]_l^C\ in\ (C_1[l_1 \mapsto \mathsf{Return}], l_1+1) \\
[\![\mathsf{let}\ x = p\ \mathsf{in}\ e]\!]_l^C & = & let\ (C_1, l_1) = [\![p]\!]_l^C, (C_2, l_2) = (C_1[l_1 \mapsto \mathsf{store}\ x], l_1+1) \\
 & & in\ [\![e]\!]_{l_2}^{C_2} \\
[\![\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_l^C & = & let\ (C_E, l_2) = [\![e_2]\!]_{l+2}^C, (C_T, l_1) = [\![e_1]\!]_{l_2}^{C_E} \\
 & & 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 & = & let\ (C_C, l_N) = [\![e_2]\!]_{l+9}^C, (C_N, l_1) = [\![e_1]\!]_{l_N}^{C_C}\ in \\
 & & \left(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\right)
\end{array}
$$

Figure 3.2: Translation into bytecode

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 [17] 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 soundenss 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 [17], 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 [23], 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.

3.1.2 General heap analysis for object-oriented programs

Building on earlier work on heap space inference for functional programs developed within MRG ([84]) Hofmann and Jost [47] have developed a framework for heap space analysis of imperative, class-based OOP which encompasses a number of design patterns and subsumes [?] and Section 3.1.1 above. In contrast to those works, however, the inference problem for [47] has not been studied yet and is likely to be very complicated in the general case. Future work will therefore aim at identifying tractable subsystems which will guide the development of further more general heap space analyses for objects within MOBIUS.

3.2 Access permissions in Midlets

Beyond the computational resources of memory space and CPU time, MOBIUS Deliverable 1.1 [66] identifies a number of other resources worth statically bounding on connected mobile devices, e.g., mobile phones. Among these resources are "billable events" like 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 [66] paragraph 3.3.1). Reducing the number of user interactions (as advocated in the resource scenarios in subsection 5.2 of [66]) 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 [21].) 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 tackles the same problem of certifying that programs do not abuse resources. Their approach is based on extending MIDP with *resource managers* (a.k.a. permission objects) which store the available resources at run time, decoupling resource access from interactive confirmation. UEDIN has developed a Java API for resource managers, which enables run-time checking of correct resource usage. This is supplemented by a type system (based on a Hoare logic) to statically certify that these run-time checks never fail.

3.2.1 Permission model

Central to our design is a rich model of resource permissions which, for a specific type of resource, relate a set of resources to a set of enabled actions with a *multiplicity* stating how many times the permission can be used. Permissions are updated by the following operations:

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

Formal definitions can be found in Figure 3.3

$$\begin{array}{rcl} \mathit{Mult} & \triangleq & (\mathbb{N} \cup \{\mathit{error}, \infty\}, \leq) \\ \mathit{Perm}_{rt} & \triangleq & (\mathcal{P}(\mathit{Res}_{rt}) \times \mathcal{P}(\mathit{Act}_{rt})) \cup \{\mathit{error}\} \\ \mathit{Perm} & \triangleq & \prod_{rt \in \mathit{ResType}} \mathit{Perm}_{rt} \times \mathit{Mult} \end{array}$$

$$\begin{array}{rcl} \mathit{consume}(p'_{rt})(\rho) & = & \begin{cases} \rho[rt \mapsto (p, m-1)] & \text{if } p' \sqsubseteq_{rt} p \wedge \rho(rt) = (p, m) \\ \rho[rt \mapsto (\mathit{error}, m-1)] & \text{otherwise} \end{cases} \\ \mathit{grant}(p_{rt}, m)(\rho) & = & \rho[rt \mapsto (p, m)] \end{array}$$

Figure 3.3: Permission model and operations

3.2.2 Secure programs

In MIDP, resource accesses are identified by API calls – typically a native one – and protected by the consumption of relevant permissions. In our model, secure programs, those which do not abuse resources, will only access resources if they have been granted the permission beforehand. Hence, secure programs exhibit execution traces for which permission consumption will never yield an error. (At consumption time, errors occur if there is no sufficient permission or if the resource multiplicity is zero).

Definition 3.2.1 (Safe trace) *A partial execution trace tr is* safe *if for all prefixes $tr' \in \mathit{prefix}(tr)$, the permission $p \in \mathit{Perm}$ held after running tr' is not an error ($\neg \mathit{Error}(p)$).*

3.2.3 Abstract program model

We have formally studied this security model on a control-flow graph model of programs. This model abstracts away any intra-procedural instruction except `consume` and `grant` operations but models method calls and exception handling. As a result, sound models of midlets can be constructed using standard class and exception analyses[1]. Formally, a control-flow graph is a 7-tuple

$$G = (\mathit{NO}, \mathit{EX}, \mathit{KD}, \mathit{TG}, \mathit{CG}, \mathit{EG}, n_0)$$

where:

- NO is the set of nodes of the graph;
- EX is the set of exceptions;
- $\mathit{KD} : \mathit{NO} \to \{\mathtt{grant}(p, m), \mathtt{consume}(p), \mathtt{call}, \mathtt{return}, \mathtt{throw}(ex)\}$, associates a kind to each node, indicating which instruction the node represents;
- $\mathit{TG} \subseteq \mathit{NO} \times \mathit{NO}$ is the set of intra-procedural edges;

[1]Dynamic class loading is not supported by midlets.

- $CG \subseteq NO \times NO$ 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.

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 $(ex, n, n') \in EG$.

Control flow graphs are equipped with a small-step operational semantics. The semantics state is a triple $(\mathit{Stack}, \mathit{Exception}?, \mathit{Perm})$. Semantics rules are given Figure 3.4.

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

$$\frac{KD(n) = \mathtt{call} \quad n \overset{CG}{\to} m}{n{:}s, \epsilon, \rho \twoheadrightarrow m{:}n{:}s, \epsilon, \rho} \qquad \frac{KD(r) = \mathtt{return} \quad n \overset{TG}{\to} n'}{r{:}n{:}s, \epsilon, \rho \twoheadrightarrow n'{:}s, \epsilon, \rho}$$

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

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

Figure 3.4: Small-step operational semantics

3.2.4 Static enforcement of secure permission usage

For this graph model of programs, a constraint-based analysis computes a safe (under)-approximation, denoted P_n, of the permissions that are guaranteed to be available at program point n. Because the analysis is inter-procedural, *summary functions*, denoted R, are used to model the effect on permissions of method calls. A representative set of constraint rules are presented Figure 3.5.

$$\frac{}{P_{n_0} \sqsubseteq p_{init}} \qquad \frac{KD(n) = \mathtt{grant}(p, m) \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq \mathit{grant}(p, m)(P_n)}$$

$$\frac{KD(n) = \mathtt{consume}(p) \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq \mathit{consume}(p)(P_n)} \qquad \frac{KD(n) = \mathtt{call} \quad n \overset{CG}{\to} m \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq R_m(P_n)}$$

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

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

Figure 3.5: Permission constraints

As established by Theorem 3.2.2, the analysis detects all the program that do not comply with the resource access policy.

Theorem 3.2.2 (Soundness) *Given a graph G, if we have $\neg Error(P_n)$ for all node n then all the partial traces of G are* safe*.*

Least solutions to the constraints can be computed by a combination of iterative and symbolic constraint solving. Hence, an effective resource analysis checker can be constructed from Theorem 3.2.2.

3.2.5 Toward a PCC resource checker for midlets

Previous works [4, 20] have shown that PCC architectures can be built upon the abstract interpretation framework. The methodology is applicable to the static analysis described above. The next paragraphs details the milestones leading to a resource checker proved correct w.r.t. the Bicolano semantics of Java bytecode [74].

- Permissions operations *i.e., grant* and *consume* have to be given a meaning at bytecode level. In the design we consider, a `Permission.grant` method is part of the *Trusted Computing Base.* This method takes as argument a permission; pops-up a security screen asking the user to grant the permission and either returns normally (upon acceptance) or throws a security exception. Because we enforce statically the absence of illegal resource accesses, the *consume* operation is not bound to an API call but annotates (native) API calls depending on the permissions they require to run.

- To construct a (possibly implicit) graph model of a bytecode program, auxiliary static analyses are needed. As stated in subsection 3.2.3, class and exception analyses can be used to build the control-flow graph of the program. To obtain the permission argument of grant nodes, we also need a data-flow analysis tracking down permissions values.

- On the proof side, we already have a Coq proof of Theorem 3.2.2 for our graph model of programs. We are confident it adapts easily to the full Bicolano formalisation. Indeed, if our graph model abstracts away intra-procedural instructions, its handling of method calls and exceptions is closed enough to the genuine Java bytecode semantics.

- To get an effective resource access checker, it remains to obtain an effective procedure for the premises of Theorem 3.2.2. This requires to provide algorithms for the constraint operators used by the analysis (see Figure 3.5). This is not always a trivial matter. For instance, to allow computability, *summary functions* must be encoded into concrete data-structures.

In a PCC setting, the code provider would send as program certificate an untrusted result of the analysis. The resource checker would check the certificate by verifying that all the constraints imposed by the analysis (see Figure 3.5) are satisfied. If so, it would conclude by Theorem 3.2.2 that the midlet is safe to run.

3.3 Explicit Accounting of External Resources

Abuse of external resources (like sending text messages) is a major concern on mobile devices, see MOBIUS Deliverable 1.1 [66] and the introduction to section 3.2. UEDIN develops a type system for bounding the usage of external resources. (A technical report [58] on this work is in preparation.)

This section introduces *resource managers* as the key device for bounding external resource usage. First, we sketch a resource manager API in Java which enables run-time checking of external resource usage. Then, a type system (based on a Hoare logic) is presented, statically certifying that these run-time checks never fail. Instead of developing the type system for the full Java language, we develop this initial version for a simpler language, an imperative procedural language without objects, heap and exceptions. Still, language and type system are expressive enough to type non-trivial programs.

3.3.1 Resource Managers in Java

We have developed an API for accounting of external resources, which may be integrated into MIDP. The API introduces special objects, called *resource managers*, which encapsulate multisets of resources that a MIDlet may legally use (according to the user's approval) and which are passed as arguments into the MIDP methods that actually use the resources. These MIDP methods, e. g., the one to send one text message, are instrumented to check the resource manager for permission first. The resource manager API consists of the following three classes.

`Resource` is an abstract class modelling a resource that resource managers may control; actual resources are subclasses. Since resources are merely names, the only public methods of this class admit comparing two instances for equality and printing an instance to a string. Subclasses which come with constructors must obey the contract that instances constructed by the same arguments be indistinguishable by the equality method.

`Multiset` is a final class holding a multiset (i. e., a bag) of resources. This class provides the standard constructors for multisets, including union, intersection and sum of two multisets, plus methods to query multisets, including an emptiness test. The class is final to avoid being tampered by subclassing.

`ResourceManager` is a final class (to avoid being tampered by subclassing) encapsulating a multiset of resources. All public methods are synchronised to avoid races in case different threads access the same resource manager. The public methods are divided into those that are to be called by the MIDlet (the constructor, `enable()`, and `clear()`) and those that are to be called by the instrumented MIDP methods (`use()`).

`ResourceManager()` is a constructor creating an empty resource manager.

`Multiset enable(Multiset r)` adds a sub-multiset `r'` of `r` to this resource manager and returns the complement of `r'`. The resource manager has to be empty when this method is called, otherwise it will throw an exception. The decision which sub-multiset `r'` to add is taken by a *black-box policy*, e. g., by the user who is ticking the resources in a pop-up dialogue. The MIDlet should check the returned multiset to learn which requested resources it is not allowed to use. In particular, if the returned multiset is empty then use of all requested resources has been approved.

`void clear()` empties this resource manager, removing all the resources it holds. In a sense, `clear()` is the inverse of `enable()`.

`void use(Multiset r)` checks whether this resource manager holds a super-multiset of `r`, and if so subtracts `r` from this manager's multiset, otherwise throws an exception.

We illustrate the use of resource managers by an example application. The code snipped below sends a text message (stored in `msg`) to a group of addresses (stored in `grp`). To simplify the presentation, we use simplified versions of the actual MIDP classes.

```
ResourceManager mgr = new ResourceManager();
Multiset r = new Multiset();
for (address in grp) {
   r.add(new MessageResource(address.get_mobile()), 1);
}
if (mgr.enable(r).is_empty()) {
   for (address in grp) {
     msg.send_rm(mgr, address.get_mobile());
   }
}
mgr.clear();
```

The above code first builds up a multiset of resources `r` by iterating over the group of addresses and for each address, extracting the mobile phone number, converting it into resource by constructing a `MessageResource` object, and adding one occurrence of the resource to the multiset `r`. Here, `MessageResource` is a subclass of `Resource`, modelling the resource of sending one text message to the phone number supplied in the constructor. Next, the code enables the resource manager `mgr` to use the multiset `r`. This will pop up a confirmation screen where the user can approve or deny the planned resource usage. Only if the user approves of all messages to be sent, i. e., if `enable()` returns an empty multiset, the code proceeds to the actual send loop. The send loop again iterates over the group of addresses, extracting for each address the mobile phone number and sending the message using the instrumented send method `send_rm()`. In the end, `clear()` is called to destroy any left-over resources so as to prevent later unintended use.

The code for the instrumented send method `send_rm()` is shown below. It constructs a multiset containing a single occurrence of the resource corresponding to the phone number `num`. Then it calls the `use()` method of the resource manager to check whether it may really send the message. If this call succeeds then as a side effect the resource is deduced from the manager. Finally, the standard MIDP `send()` method is called. Note the explicit synchronisation on `mgr` which is necessary to avoid races in case another thread could access the same resource manager.

```
void send_rm(ResourceManager mgr; String num) {
   Multiset r = new Multiset(new MessageResource(num), 1);
   synchronized (mgr) {
     mgr.use(r);
     send(num);
   }
}
```

3.3.2 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. Note that this 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 [8].

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

- extensible array types (with the usual query and update operators),
- a type of (names of) *resources* (which can only be compared for equality),
- a type of *multisets* of resources (with the usual operator, including the multiset sum $\oplus$), and
- a type of *resource managers* (with no operators at all).

Since there are no operators on resource managers, the language provides three built-in procedures to access them.

- **clear**$(\mid m')$ creates an empty manager m'.

- **enable**$(r, m \mid r', m')$ tries to top up an empty manager m with the multiset of resources r (e. g., by asking the user to confirm). It returns the new manager m' holding the granted multiset of resources and the multiset r' of resources that have not been granted, i. e., the multisets in m' and r' sum up to r. Triggers a run-time error if m is not empty.

- **use**$(r, m \mid m')$ deduces the multiset of resources r from the manager m and returns the new manager m'. Triggers a run-time error if m does not hold enough resources.

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 $p(x_1, \dots, x_m \mid y_1, \dots, y_n)$, i. e., the variables x_i and y_j are all different. This restriction simplifies the effect type system in subsection 3.3.3, for assignments behave like let bindings in a functional language (e. g., GRAIL [8]).

- 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 deduce resources r from m twice by calling **use**$(r, m \mid m')$ and **use**$(r, m \mid m'')$.

Logical semantics of expressions. In section 3.3.3, we will introduce a type system that ascribes preconditions and effects to statements and procedures. Preconditions and effects are logical formulae or *constraints*, so we need to define a constraint language that admits reasonable assertions about statements and procedures.

We translate expressions into terms of a many-sorted first-order constraint language with equality (denoted by $\approx$). This technique is standard in program verification (e. g., for generating verification conditions), so we describe it only cursorily here.

The translation straightforwardly maps the constants and operators of expressions to the logical constants and operators of the corresponding theories. Therefore, the constraint language must host the theories of the Booleans, integer arithmetic, arrays, resource constants, and multisets of resource constants. Resource managers are translated into the theory of multisets, so on the level of constraints we identify a resource manager with the multiset of resources it holds.

Terms in the constraint language are total (i. e., defined for all values of their free variables) whereas expressions can be partial (e. g., accessing an array at a negative index is not defined). Therefore, we actually translate an expression e into a pair $\langle \phi_e, t_e \rangle$ consisting of a formula ϕ_e and a term t_e. The term t_e is the semantics of e in case e is defined, and the formula ϕ_e characterises when e is defined.

We introduce notation for valuations of free variables in the constraints. Given a valuation α, a variable x and a value a' (compatible with the sort of x), by $x \mathbin{@} \alpha$ we denote the value a which α assigns to x, and by $\alpha[x \mapsto a']$ we denote for the valuation α' which assigns a' to x and $y \mathbin{@} \alpha$ to all variables $y \neq x$. Given a term t and a valuation α, we denote the evaluation of t under α by $t \mathbin{@} \alpha$. We write $\alpha \models \phi$ if the valuation α makes the formula ϕ true, and we write $\models \phi$ if ϕ is valid.

Operational semantics of statements. We present a big-step operational semantics of our programming language. The semantics judgement takes the form $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$, where Π is a program, Γ is a variable environment for the variables occurring in the statement s, and β and β' are the pre- resp. post-states of

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

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

$$(\text{OS-assign-}\bot)\ \frac{\alpha \not\models \phi_e}{\Pi, \Gamma \vdash y := e \triangleright \alpha \rightsquigarrow \bot} \qquad (\text{OS-assign})\ \frac{\alpha \models \phi_e \qquad \alpha' = \alpha[y \mapsto t_e \mathbin{@} \alpha]}{\Pi, \Gamma \vdash y := e \triangleright \alpha \rightsquigarrow \alpha'}$$

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

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

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

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

$\Pi(q)$	Φ_q	Ψ_q
$\mathbf{clear}(\mid \mathbf{m}' : \mathbf{mgr})$	*true*	$\mathbf{m}' \approx \emptyset$
$\mathbf{use}(\mathbf{r} : \mathbf{mres}, \mathbf{m} : \mathbf{mgr} \mid \mathbf{m}' : \mathbf{mgr})$	$\mathbf{r} \subseteq \mathbf{m}$	$\mathbf{m} \approx \mathbf{m}' \oplus \mathbf{r}$
$\mathbf{enable}(\mathbf{r} : \mathbf{mres}, \mathbf{m} : \mathbf{mgr} \mid \mathbf{r}' : \mathbf{mres}, \mathbf{m}' : \mathbf{mgr})$	$\mathbf{m} \approx \emptyset$	$\mathbf{r} \approx \mathbf{r}' \oplus \mathbf{m}'$

Figure 3.6: Operational semantics.

the execution of s. We write

$$\begin{aligned} &\Pi(p) = p(x_1 : \tau_1, \ldots, x_m : \tau_m \mid y_1 : \sigma_1, \ldots, y_n : \sigma_n)\{z_1 : \kappa_1, \ldots, z_l : \kappa_l;\ s\} \\ \text{and } &\Pi(q) = q(x_1 : \tau_1, \ldots, x_m : \tau_m \mid y_1 : \sigma_1, \ldots, y_n : \sigma_n)\end{aligned}$$

to expose the declarations of input parameters x_i, output parameters y_j, local variables z_k and body s of the procedure p or the built-in q in Π. We write Γ_p resp. Γ_q for the variable environment (mapping input, output and local variables to their declared types) associated with these procedure resp. built-in declarations. Given a variable environment Γ, a Γ-*state* (or *state* if Γ is understood) is either the error state $\bot$ or a valuation α supplying values for all variables declared in Γ. For procedures p and built-ins q, the terms *p-state* and *q-state* mean Γ_p-state and Γ_q-state, respectively. We denote states by the letters α and β, with the convention that a state denoted by α cannot be the error state.

Figure 3.6 displays the rules[2] defining the big-step semantics. Most of the rules are quite standard, safe for the rule schemas (OS-q) and (OS-q-$\bot$) dealing with the built-ins $q =$ **clear**, **use**, **enable**. Rule (OS-q) executes a call to the built-in q by creating a local state α_q, copying the values of the input variables to the input parameters in α_q, checking the precondition Φ_q, checking the effect Ψ_q, and copying the values of the output parameters in α_q back to the output variables. 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 $y_1, \ldots, y_n$ such that the effect is

[2]The standard rules for sequencing and conditionals have been omitted.

satisfied. The choice may be non-deterministic, e. g., in the case of **enable** there are indeed many values for $\mathbf{r'}$ for $\mathbf{m'}$ such that their sum yields $\mathbf{r}$. Rule (OS-q-$\bot$), which is deterministic, covers the case when the precondition of the built-in q fails to hold.

Monotonicity of resource usage. To justify our modelling of stateful resource manager objects as 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 variable environment for statement s and let β and β' be Γ-states (with the intention that β is the pre-state of an execution of s and β' is the post-state). By $\mathit{pre}^s_\Gamma(\beta)$, we denote the 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 $\mathit{post}^s_\Gamma(\beta')$, 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 variable environment for a statement s, and let β and β' be Γ-states. If Δ_{OS} is a derivation of $\Pi, \Gamma \vdash s \triangleright \beta \rightsquigarrow \beta'$ which does not call* **enable**, *i. e., Δ_{OS} does not use rule (OS-***enable***), then $\mathit{pre}^s_\Gamma(\beta) \supseteq \mathit{post}^s_\Gamma(\beta')$.*

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

3.3.3 Effect Types

Run-time errors occur because some precondition fails to hold, either the definedness condition of an expression in an assignment, see rule (OS-assign-$\bot$), or a precondition of a built-in, see rules (OS-**enable**-$\bot$) and (OS-**use**-$\bot$). 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.

Types and typing judgement. We use $\bar{x}$, $\bar{y}$ and $\bar{z}$ as abbreviations for sequences of variables, e. g., $\bar{x}$ stands for $x_1, \ldots, x_m$. Sequences can be concatenated by comma. We view these sequences as sets rather than as lists, i. e., their order does not matter and there are no duplicate variable occurrences.

Given two sequences of variables $\bar{x}$ and $\bar{y}$ and two formulae Φ and Ψ, we call $\Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x},\bar{y}\rangle$ an *effect type* if $\{\bar{x}\} \cap \{\bar{y}\} = \emptyset$, $\mathit{free}(\Phi) \subseteq \{\bar{x}\}$, and $\mathit{free}(\Psi) \subseteq \{\bar{x},\bar{y}\}$. We call Φ and Ψ *precondition* and *effect*, and we call $\bar{x}$ and $\bar{y}$ *input* and *output* variables, respectively.

Let Π be a program. In order to type procedure calls, we need access to an effect type of the callee. Therefore, we introduce *effect environments* Θ, which map the procedures and built-ins in Π to effect types.

Let Π be a program and Θ and effect environment. Let Γ be a variable environment for a statement s, and let $\Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x},\bar{y}\rangle$ be an effect type. The *effect typing judgement* $\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x},\bar{y}\rangle$ may be interpreted informally in the following ways.

1. During every (terminating) execution of s whose initial state satisfies the precondition Φ, the preconditions of all procedures and built-ins called will be satisfied.

2. For every terminating execution of s whose initial state satisfies the precondition Φ, the terminal state will satisfy the effect Ψ. Note that the values of the input variables in the terminal state are the same as in the initial state, thanks to s being in SSA form.

Typing of statement effects $\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}\rangle$

$$\text{(T-precond)}\ \frac{\begin{array}{c}\models \forall\bar{x}(\hat{\Phi} \Rightarrow \Phi)\\ \Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}\rangle\end{array}}{\Pi, \Theta, \Gamma \vdash s : \hat{\Phi}\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}\rangle} \qquad \text{(T-effect)}\ \frac{\begin{array}{c}\models \forall\bar{x}\forall\bar{y}(\Phi \wedge \Psi \Rightarrow \hat{\Psi})\\ \Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}\rangle\end{array}}{\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \hat{\Psi}\langle\bar{x}, \bar{y}\rangle}$$

$$\text{(T-intro)}\ \frac{\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}\rangle \qquad z \notin \mathit{def}(s)}{\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}, z\rangle \to \Psi\langle\bar{x}, z, \bar{y}\rangle} \qquad \text{(T-elim)}\ \frac{\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}, z\rangle}{\Pi, \Theta, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \exists z \Psi\langle\bar{x}, \bar{y}\rangle}$$

$$\text{(T-skip)}\ \frac{}{\Pi, \Theta, \Gamma \vdash \mathbf{skip} : \mathit{true}\langle\rangle \to \mathit{true}\langle\rangle} \qquad \text{(T-assign)}\ \frac{\{\bar{x}\} = \mathit{free}(t_e)}{\Pi, \Theta, \Gamma \vdash y := e : \phi_e\langle\bar{x}\rangle \to y \approx t_e\langle\bar{x}, y\rangle}$$

$$\text{(T-call)}\ \frac{\begin{array}{c}\Pi(p) = p(x_1 : \tau_1, \ldots, x_m : \tau_m \mid y_1 : \sigma_1, \ldots, y_n : \sigma_n)[\{z_1 : \kappa_1, \ldots, z_l : \kappa_l;\ s\}]\\ \Theta(p) = \Phi\langle x_1, \ldots, x_m\rangle \to \Psi\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle\end{array}}{\Pi, \Theta, \Gamma \vdash p(x'_1, \ldots, x'_m \mid y'_1, \ldots, y'_n) : \Phi'\langle x'_1, \ldots, x'_m\rangle \to \Psi'\langle x'_1, \ldots, x'_m, y'_1, \ldots, y'_n\rangle}$$

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

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

Typing of procedure and built-in effects $\Pi, \Theta \vdash p$

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

$$\text{(T-builtin)}\ \frac{\begin{array}{c}\Pi(q) = q(x_1 : \tau_1, \ldots, x_m : \tau_m \mid y_1 : \sigma_1, \ldots, y_n : \sigma_n)\\ \Theta(q) = \Phi_q\langle x_1, \ldots, x_m\rangle \to \Psi_q\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle\end{array}}{\Pi, \Theta \vdash q}$$

Typing of effect environments $\Pi \vdash \Theta$

$$\text{(T-prog)}\ \frac{\begin{array}{c}\Pi\ =\ p_1(\ \ldots\ )[\{\ \ldots\ \}]\ \ \ldots\ \ p_n(\ \ldots\ )[\{\ \ldots\ \}]\\ \Theta(p_1) = \Phi_1\langle\ldots\rangle \to \Psi_1\langle\ldots\rangle \quad \ldots \quad \Theta(p_n) = \Phi_n\langle\ldots\rangle \to \Psi_n\langle\ldots\rangle\\ \Pi, \Theta \vdash p_1 \qquad \ldots \qquad \Pi, \Theta \vdash p_n\end{array}}{\Pi \vdash \Theta}$$

Figure 3.7: Typing rules for effect types.

Typing rules. Figure 3.7 displays the typing rules for effect types. Note that these rules assume well-formedness of the effect types occurring in premises and conclusion, which may impose additional restrictions. For instance, well-formedness of the conclusion of rule (T-intro) imposes that the newly introduced input variable z does not occur among the output variables $\bar{y}$.

The typing rules for statement effects fall into three groups. First, the logical rules (T-precond) and (T-effect) allow to strengthen the precondition or weaken the effect, respectively. Second, the variable rules (T-intro) and (T-elim) allow to introduce a new input variable or eliminate an output variable, respectively. Note that (T-intro) must make sure that the newly introduced input variable z is not being defined by the statement s (i. e., z neither occurs on the left-hand side of an assignment nor as an output parameter of a procedure call), hence the premise $z \notin \mathit{def}(s)$. Finally, each statement constructor has its rule. Rules (T-skip), (T-seq) and (T-if) are self-explanatory (but recall that z in (T-if) is a Boolean variable and note that the restrictions on effect types force z to occur among the input variables $\bar{x}$). Rule (T-assign) synthesises an effect type for an assignment using the translation of expressions e into pairs of formulae ϕ_e and terms t_e. Rule (T-call) looks up the effect type $\Theta(p) = \Phi\langle x_1, \ldots, x_m\rangle \to \Psi\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$ of the callee p (which could be a procedure or a built-in) and converts it to the effect type for the call $p(x'_1, \ldots, x'_m \mid y'_1, \ldots, y'_n)$ by substituting the actual input and output parameters x'_i and y'_j for the formal parameters x_i and y_j in precondition Φ and effect Ψ.

To justify the effect types assumed by the environment Θ, we have to prove the judgement $\Pi \vdash \Theta$. Rule (T-prog) does this by proving $\Pi, \Theta \vdash p$ for every procedure or built-in p in the program Π. In turn, the judgement $\Pi, \Theta \vdash p$ means that the environment's assumption $\Theta(p)$ about the effect type of a procedure or built-in p is correct. This is proven by rule (T-proc) if p is a procedure (by checking the body of p against the conjectured effect type), and by rule (T-builtin) if p is a builtin (by constructing the effect type from the operational semantics, see figure 3.6).

Type soundness. Theorem 3.3.2 below states that for a statement s of effect type $\Phi\langle x_1, \ldots, x_m\rangle \to \Psi\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$, 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, (2) the input variables x_i have preserved their values, and (3) the final state satisfies the effect Ψ. Mostly, (1) is important to us as it means that typable statements are safe from run-time errors (as long as the initial state satisfies the precondition), in particular safe from lack-of-resources errors raised by rule (OS-**use**-$\bot$). (3) associates the effect Ψ with a Hoare-style postcondition, however, note that both output and input variables (which by (2) have kept their initial values) may occur in Ψ, so Ψ may specify a input-output relation (or effect). (2) really is a consequence of the statement s being in SSA form. Note that 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 variable environment for a statement s, let $\Phi\langle x_1, \ldots, x_m\rangle \to \Psi\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$ be an effect type, and let β and β' be Γ-states. If*

1. $\Pi \vdash \Theta$,
2. $\Pi, \Theta, \Gamma \vdash s : \Phi\langle x_1, \ldots, x_m\rangle \to \Psi\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$,
3. $\Pi, \Gamma \vdash s \rhd \beta \rightsquigarrow \beta'$,
4. $\beta \neq \bot$, *and*
5. $\beta \models \Phi$

then

1. $\beta' \neq \bot$,
2. *for all* i, $x_i \mathbin{@} \beta = x_i \mathbin{@} \beta'$, *and*

3. $\beta' \models \Psi$.

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).

3.3.4 Related Work

Nanevski et al. [71] 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. [24] 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 [24] only deals with one fixed resource.

3.3.5 Future Work

Integration of our type system with the MOBIUS logic [67] will be done in the style of the MRG project. This requires translating our effect type judgements $\Theta, \Pi, \Gamma \vdash s : \Phi\langle\bar{x}\rangle \to \Psi\langle\bar{x}, \bar{y}\rangle$ 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. Work on type checking will already commence in Task 2.3. Since type checking involves proving validity of first-order formulae (rules T-precond and T-effect), 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 be helpful. At least in the case of non-recursive procedures, static analysis should gather enough information for synthesising effect types. To deal with recursion, we will investigate how to approximate the effects of recursive calls, e.g. by computing *procedure summaries*.

3.4 Execution Time Estimation

Execution time has been identified as one of the resource-oriented properties of programs which are fundamental and meaningful in real MIDP applications for MOBIUS (Deliverable 1.1 [66], Resource and Information Flow Security Requirements). Execution times obviously depend in practice on the sizes of certain inputs or their values. In this section we present a method which is aimed at statically inferring functions that yield upper- and lower-bounds on actual (i.e., platform-dependent) *execution times* for procedures. These functions take as input the sizes or value ranges of the input data to the procedure. The method builds on previous work on inferring similar functions providing upper- and lower-bounds on the number of *execution steps* performed by procedures (based again on the sizes or value ranges of input) [31, 30, 54, 32, 46]. This method is based heavily on the use of abstract interpretation. As mentioned before, previous work [4, 20, 45] has shown that PCC architectures can be built upon the abstract interpretation framework (ACC). The proposed method is directly relevant to the ACC approach and can be used in particular for resource consumption certification of mobile code.

3.4.1 Overview of the Approach

Our approach essentially extends the previously developed compile-time analysis for upper and lower bounds on number of execution steps: similar platform-independent cost functions are first inferred, but these functions are now parameterized by certain constants, each of them related to the actual execution cost on each platform of a certain operation or class of operations that the compiler translates the code to. These operations may be intermediate canonical code, operations at the RTL level, bytecode instructions, etc. For each execution platform, the value of such constants is determined experimentally once and for all by running a set of synthetic benchmarks and measuring their running times. These parameters essentially calibrate a cost model which, from then on, is able to compute statically execution time bound functions for procedures and to predict with a significant degree of accuracy the execution times of such procedures in the given platform.

We have implemented our approach within the `CiaoPP` system, which can analyze and generate ACC-based certificates for different programming languages, including Java (source code), functional, and (constraint) logic programing languages. We expect to extend the system to deal with Java bytecode in the next period. In the `CiaoPP` system, assertions can be added to the program describing desired bounds on the efficiency of the program as a specification which the system will try to verify or falsify. The cost function (or complexity order) in the specification is compared with the upper- or lower-bound cost functions inferred by analysis. In this context, upper bounds are used to prove that upper-bound cost specifications are met (i.e., when the upper-bound computed by analysis is lower or equal than the upper-bound stated by the user in the specification). Lower bounds are used conversely for disproving upper bounds on resource usage, e.g., proving that a program will use more resources that those stated in the specification or safety policy. This is why we are also interested in lower bounds.

We have studied a number of cost models, involving different sets of constants in order to explore experimentally their precision and to see which models produce the most precise results, i.e., which parameters model and predict best the actual execution times of procedures. In doing this we have taken into account the trade-off between simplicity of the cost models (which implies efficiency of the cost analysis and also simpler profiling) and the precision of their results. With this aim, we have started with a simple model and explored several possible refinements.

3.4.2 Platform-Dependent Static Cost Analysis

Since the work done by a call to a recursive procedure often depends on the "size" of its input, such work cannot in general be statically estimated in any reasonable way without knowledge of (or bounds on) the actual input data sizes. Thus, our basic approach is as follows: given a call p, an expression $\Phi_p(n)$ is computed that a) is relatively simple to evaluate, and b) it approximates $\texttt{Time}_p(n)$, where $\texttt{Time}_p(n)$ denotes the cost (in time units) of computing p for an input of size n. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. The idea is that $\Phi_p(n)$ is determined at compile time. It is then evaluated at run-time, when the size of the input is known, yielding an estimate of the cost of the call. In the following we will refer to the compile-time computed expressions $\Phi_p(n)$ as *cost functions*. Types, modes, and size measures are first automatically inferred by other (abstract interpretation-based) analyzers which are part of `CiaoPP` and then used in the size and cost analysis.

Platform-Independent Static Cost Analysis

As mentioned before, our static cost analysis approach is based on that of [31, 30] (for estimation of upper bounds on execution steps) and further extended in [32] (for lower bounds). In these approaches the cost of a procedure definition (generally taken as a number of execution steps) can be bounded by the cost of the basic operations in the body of the procedure (including the parameter passing cost), combined with bounds on the cost of each of the procedure calls in the body. For simplicity, the discussion that follows is focused on the estimation of upper bounds. We refer the reader to [32] for details on lower-bounds analysis. Consider a procedure definition `p` given as "$\mathtt{H}\ \{\mathtt{L_1}, ..., \mathtt{L_m}\}$" where $\mathtt{L_i}$ is either a procedure call or a basic operation of the language (which we will refer to as "builtins"). We assume that the source language has been desugared so that the program is made up of a number of flat procedure definitions, and loops are converted to tail recursions. Also, conditionals and case statements are expressed as alternative procedure definitions (clauses) of a given procedure (i.e., in Horn clause style).

Our model deals both with deterministic languages (such as Java) and non-deterministic languages in which procedure calls can return several solutions, and the number of times a procedure call will be executed depends on the number of solutions that the procedure calls preceding it can generate. Assume that $\overline{n}$ is a vector such that each element corresponds to the size of an input argument to procedure definition `p` and that each $\overline{n}_i$, $i = 1 \dots m$, is a vector such that each element corresponds to the size of an input argument to procedure call $\mathtt{L}_i$, τ is the cost of the basic operations in the body of the procedure (including the parameter passing cost), and $\mathtt{Sols}_{\mathtt{L}_j}$ is the number of solutions procedure call $\mathtt{L}_j$ can generate. Then, an upper bound on the cost of procedure definition `p` (assuming all solutions are required), $\mathtt{Cost}_\mathtt{p}(\overline{n})$, can be expressed as:

$$\mathtt{Cost}_\mathtt{p}(\overline{n}) \leq \tau + \sum_{i=1}^{m} (\prod_{j \prec i} \mathtt{Sols}_{\mathtt{L}_j}(\overline{n}_j)) \mathtt{Cost}_{\mathtt{L}_i}(\overline{n}_i), \tag{3.1}$$

Here we use $j \prec i$ to denote that $\mathtt{L}_j$ precedes $\mathtt{L}_i$ in the procedure definition.

In order to further simplify the discussion that follows, and given that in the MOBIUS applications we are dealing with a deterministic language (Java or Java bytecode), we restrict ourselves to the simple case where each procedure call is determinate, i.e., produces at most one solution. In this case, equation (3.1) simplifies to:

$$\mathtt{Cost}_\mathtt{p}(\overline{n}) \leq \tau + \sum_{i=1}^{m} \mathtt{Cost}_{\mathtt{L}_i}(\overline{n}_i). \tag{3.2}$$

(However, it is important to note that our implementation is not limited to deterministic programs: our system handles non determinism, i.e., presence of several solutions for a given call.)

A difference equation is set up for each recursive procedure definition, whose solution (using as boundary conditions the cost of non-recursive procedure definitions) is a function that yields the cost of a procedure definition. The cost of cases where multiple definitions occur (as in conditionals and case statements) is computed from the cost of each of its branches. If mutual exclusion of the branches can be established the cost of the block can be bounded by the maximum of the costs of mutually exclusive clusters of branches. In languages where procedures can produce multiple solutions the number of solutions generated by a procedure that will be demanded is generally not known in advance. In these cases a conservative upper bound on the computational cost of a procedure can be obtained by assuming that all solutions are needed.

As mentioned before, previous analyses were primarily aimed at estimating execution steps. However, the basic metric is open and can be tailored to alternative metrics as the unit of cost in the analysis, so that instead of the number of steps, for example, the number of low-level instructions executed could be counted. In the rest of this section we explore this issue and study how to extended the cost analysis in order to infer cost functions using more refined parametric cost models, which in turn will allow achieving accurate execution time bound analysis.

Proposed Platform-Dependent Cost Analysis Models

Since the cost metric which we want to use in our approach is execution time, we take τ (in expression 3.2) to include the time taken by parameter passing, the basic operations in the body of the procedure, and the cost involved in setting up the body procedure calls for execution. In the following, we will refer to τ as the *procedure local cost function*, which takes all these costs into account. We will consider different values for τ, each of them yielding a different cost model. These cost models make use of a vector of platform-dependent constants, together with a vector of platform-independent metrics, each one corresponding to a particular low-level operation related to program execution. Examples of such low-level operations considered by the cost models may be passing a parameter to a procedure or returning it, pushing and popping frames to and from the stack, creating new data structures on the heap, creating choice-points, etc. Thus, we assume that τ is a function parameterized by the cost model, so that:

$$\tau(\Omega) = \mathit{time}(\Omega) \tag{3.3}$$

where $\mathit{time}(\Omega)$ is a function that gives the time associated with all these costs for the cost model named Ω. We study a family of cost models such that $\mathit{time}(\Omega)$ is a function defined as follows:

$$\mathit{time}(\Omega) = \mathit{time}(\omega_1) + \cdots + \mathit{time}(\omega_v),\ v > 0 \tag{3.4}$$

where each $\mathit{time}(\omega_i)$ provides that part of the execution time which depends on the metric ω_i. We assume that:

$$\mathit{time}(\omega_i) = K_{\omega_i} \times I(\omega_i) \tag{3.5}$$

where K_{ω_i} is a platform-dependent constant, and $I(\omega_i)$ is a platform-independent cost function.
Since $\mathit{time}(\Omega)$ is a linear combination of platform-independent cost functions, we can write equation (3.4) as:

$$\mathit{time}(\Omega) = \overline{K_\Omega} \bullet \overline{I(\Omega)} \tag{3.6}$$

where $\overline{K_\Omega}$ is a vector of platform-dependent constants, $\overline{I(\Omega)}$ is a vector of platform-independent cost functions, and $\bullet$ is the dot product. Accordingly, we generalize the definition of equation (3.2) introducing this cost function τ as a parameter:

$$\mathtt{Cost_p}(\tau, \overline{n}) \leq \tau + \sum_{i=1}^{m} \mathtt{Cost_{L_i}}(\overline{n}_i). \tag{3.7}$$

Using equations 3.3–3.6, we have:

$$\mathtt{Cost_p}(\mathit{time}(\Omega), \overline{n}) = \overline{K_\Omega} \bullet \overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n}) \tag{3.8}$$

where $\overline{K_\Omega}$, $\overline{I(\Omega)}$ and $\overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n})$ are vectors of the form:
$\overline{K_\Omega} = (K_{\omega_1}, \ldots, K_{\omega_v})$,
$\overline{I(\Omega)} = (I(\omega_i), \ldots, I(\omega_v))$, and
$\overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n}) = (\mathtt{Cost_p}(I(\omega_1), \overline{n}), \ldots, \mathtt{Cost_p}(I(\omega_v), \overline{n}))$

A particular definition of $\overline{I(\Omega)}$ yields a cost model. We have tried several cost models, by using different vectors $\overline{I(\Omega)}$ constructed by choosing different $I(\omega_i)$ cost functions. Equation (3.8) gives the basis for computing values for constants K_{ω_i} via profiling (as explained in Section 3.4.4). Also, it provides a way to obtain the cost of a procedure expressed in a platform-dependent cost metric from another cost expressed in a platform-independent cost metric.

3.4.3 Dealing with the Builtin and External Operations in the Language

We assume that there is a cost function (expressed via trust assertions [46]) for each builtin operation in the language and for each procedure whose code is not available to the analyzer (procedures in binary libraries, calls to other languages, etc.). In some cases, this cost function will be given by a constant value, and in others by a function that depends on properties of the (input) arguments of the procedure.

Going into more detail, we assume that each builtin contributes with a new component to the execution time as expressed in Equation (3.4), that is, our cost model will have a new component $time(\omega_i)$ for each builtin. Let $\odot/n$ be an arithmetic operator (an example of such a language builtin). The execution time due to the total number of times that such operator is evaluated is given by:

$$time(\odot/n) = K_{\odot/n} \times I(\odot/n)$$

where $K_{\odot/n}$ is a platform-dependent constant, and $I(\odot/n)$ is a platform-independent cost function. $K_{\odot/n}$ approximates the cost (in units of time) of evaluating the arithmetic operator $\odot/n$. $I(\odot/n)$ could be the number of times that the arithmetic operator is evaluated. Alternatively, it can be a cost function defined as:

$$I(\odot/n) = \sum_{a \in S} \texttt{EvCost}(\odot/n, a)$$

where S is the set of arithmetic expressions appearing in the procedure definition body which will be evaluated and $\texttt{EvCost}(\odot/n, a)$ represents the cost corresponding to the operator $\odot/n$ in the evaluation of the arithmetic term a, i.e.:

$$\texttt{EvCost}(\odot/n, A) = \begin{cases} 0 & \text{if } A \text{ is a constant} \\ & \quad \text{or a variable} \\ 1 + \sum_{i=1}^{n} \texttt{EvCost}(\odot/n, A_i) & \text{if } A = \odot(A_1, ..., A_n) \\ \sum_{i=1}^{m} \texttt{EvCost}(\odot/n, A_i) & \text{if } A \neq \odot(A_1, ..., A_n) \\ & \wedge\ A = \hat{\odot}(A_1, ..., A_m) \\ & \text{for some operator } \hat{\odot}/m \end{cases}$$

For simplicity, we assume that the cost of evaluating the arithmetic term t to which a variable appearing in A will be bound at execution time is zero (i.e., we ignore the cost of evaluating t). This is a good approximation if in most cases t is a number and thus no evaluation is needed for it. However, a more refined cost model could assume that this cost is a function on the size of t.

Note that this model ignores the possible optimizations that the compiler might perform. We can take into account those performed by source-to-source transformation by placing our analyses in the last stage of the front-end, but at some point the language the compiler works with would be different enough as to require different considerations in the cost model.

However, experimental results show that our simplified cost model gives a good approximation of the execution times for arithmetic builtin procedures. With these assumptions, equation (3.8) (in Section 3.4.2) also holds for programs that perform calls to builtin procedures, say, for example, a builtin b/n, by introducing b/n and $\odot/n$ as new cost components of Ω.

A similar approach can be used for other (non-arithmetic) builtins b/n using the formula:

$$time(b/n) = K_{b/n} \times I(b/n)$$

3.4.4 Calibrating Constants via Profiling

In order to compute values for the platform-dependent constants which appear in the different cost models proposed in Section 3.4.2, our calibration schema takes advantage of the relationship between the platform-dependent and -independent cost metrics expressed in Equation (3.8). In this sense, the calibration of the constants appearing in $\overline{K}_\Omega$ is performed by solving systems of linear equations (in which such constants are treated as variables).
Based on this expression, the calibration procedure consists of:

1. Using a selected set of calibration programs which aim at isolating specific aspects that affect execution time of programs in general. For these calibration programs it holds that $\mathtt{Cost_p}(I(\omega_i), \overline{n})$ is known for all $1 \le i \le v$. This can be done by using any of the following methods:

   - The analyzers integrated in the `CiaoPP` system infer the exact cost function, i.e.:

     $$\mathtt{Cost_p}^l(I(\omega_i), \overline{n}) = \mathtt{Cost_p}^u(I(\omega_i), \overline{n}) = \mathtt{Cost_p}(I(\omega_i), \overline{n})$$

     where $\mathtt{Cost_p}^l(I(\omega_i), \overline{n})$ and $\mathtt{Cost_p}^u(I(\omega_i)$, are functions that yield lower and upper bounds on the cost of procedure `p` respectively,
   - $\mathtt{Cost_p}(I(\omega_i), \overline{n})$ is computed by a profiler tool, or
   - $\mathtt{Cost_p}(I(\omega_i), \overline{n})$ is supplied by the user together with the code of program `p` (i.e., the cost function is not the result from any automatic analysis but rather `p` is well known and its cost function can be supplied in a trust assertion).

2. For each benchmark p in this set, automatically generating a significant amount m of input data for it. This can be achieved by associating with each calibration program a data generation rule.

3. For each generated input data d_j, computing a pair $(\overline{C}_{p_j}, T_{p_j})$, $1 \le j \le m$, where:

   - T_{p_j} is the j-th observed execution time of program p with this generated input data.
   - $\overline{C}_{p_j} = \overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n_j})$, where $\overline{n_j}$ is the size of the j-th input data d_j.

4. Using the set of pairs $(\overline{C}_{p_j}, T_{p_j})$ for setting up the equation:

   $$\overline{C}_{p_j} \bullet \overline{K}_\Omega = T_{p_j} \tag{3.9}$$

   where $\overline{K}_\Omega$ is considered a vector of variables.

5. Setting up the (overdetermined) system of equations composed by putting together all the equations (3.9) corresponding to all the calibration programs.

6. Solving the above system of equations using the least square method (see, e.g., [90]). A solution to this system gives values to the vector $\overline{K}_\Omega$ and hence, to the constants K_{ω_i} which are the elements composing it.

7. Calculating the constants for builtins and arithmetic operators by performing repeated tests in which only the builtin being tested is called, accumulating the time, and dividing the accumulated time by the number of times the repeated test has been performed.

3.4.5 Assessment of the Calibration of Constants

We have assessed both the constant calibration process and the prediction of execution times using a number of cost models in two different platforms:

- "intel" platform: Dell Optiplex, Pentium 4 (Hyper threading), 2GHz, 512MB RAM memory, Fedora Core 4 operating System with Kernel 2.6.
- "ppc" platform: Apple iMac, PowerPC G4 (1.1) 1.5GHz, 1GB RAM memory, with Mac OS X 10.4.5 Tiger.

In section 3.4.4 we presented equation 3.9, and we mentioned that it can be solved using the least squares method. We used the householder algorithm, which consists in decomposing the matrix $C = \{\overline{C}_{p_j}\}$, which has m rows and n columns into the product of two matrices Q and U (denoted $\bullet$ or without any symbol) such that $C = Q \bullet U$, where Q is an orthonormal matrix (i.e., $Q^T \bullet Q = I$, the $m \times m$ identity matrix) and U an upper triangular $m \times n$ matrix. Then, multiplying both sides of the equation 3.9 by Q^T and simplifying we can get:

$$U \bullet K = Q^T \bullet T = B$$

where, for clarity, we denote $K = \overline{K}_\Omega$, $T = T_{p_j}$ and $Q^T \bullet T = B$. We can take advantage of the structure of U and define V as the first n rows of U, n being the number of columns of C and b the first n rows of B, then K can be estimated solving the following upper triangular system, where $\hat{K}$ stands for the estimate for K:

$$V \bullet \hat{K} = Q^T \bullet T = b$$

Since this method is being used to find an approximate solution, we define the residual of the system as the value

$$R = T - C\hat{K}$$

Let

$$RSS = R \bullet R$$

be the residual square sum, and let

$$MRSS = \frac{RSS}{m - n}$$

be the mean of residual square sum, where m and n are the number of rows and columns of the matrix C respectively, and finally let

$$S = \sqrt{MRSS}$$

be the estimation of the model standard error, S. In order to experimentally evaluate which models better approximate the observed time in practice, we have compared the values of $MRSS$ (or S) for several proposed models. Table 3.1 shows the estimated values for the vector K using a number of calibration programs as well as the standard error of the model, sorted from the best to the worst model. For example, the first row in the table shows the model that has as components step, nargs, instruct, outstruct, inptr, outptr for the intel platform. It has a standard error of 6.2475 μs and the values for each of the constants are 21.27, 9.96, 10.30, 8.23, 6.46, and 5.69 nanoseconds, respectively.

Note that the estimation of K is done just once per platform. In the case of the intel platform it took 15.62 seconds and in ppc 17.84 seconds, repeating the experiment 250 times for each program.

Plat.	Model	S (μs)	$\overline{K}_\Omega$
intel	step nargs instruct outstruct inptr outptr	6.2475	(21.27, 9.96, 10.30, 8.23, 6.46, 5.69)
	step instruct outstruct inptr outptr	9.3715	(26.56, 10.81, 8.60, 6.17, 6.39)
	step instruct outstruct outptr	13.7277	(27.95, 11.09, 8.77, 7.40)
	step	68.3088	108.90
ppc	step nargs instruct outstruct inptr outptr	4.7167	(41.06, 5.21, 16.85, 15.14, 9.58, 9.92)
	step instruct outstruct inptr outptr	5.9676	(43.83, 17.12, 15.33, 9.43, 10.29)
	step instruct outstruct outptr	16.4511	(45.95, 17.55, 15.59, 11.82)
	step	116.0289	183.83

Table 3.1: Global values for vector constants in several cost models (in nanoseconds), sorted by S, the standard error of the model.

Platform	Model	Error (%)
intel	step nargs instruct outstruct inptr outptr	53.17
	step instruct outstruct inptr outptr	31.06
	step instruct outstruct outptr	21.48
	step	58.45
ppc	step nargs instruct outstruct inptr outptr	18.72
	step instruct outstruct inptr outptr	14.66
	step instruct outstruct outptr	19.44
	step	43.04

Table 3.2: Global comparative of the accuracy of cost models.

3.4.6 Assessment of the Prediction of Execution Times

We have tested the implementation of the proposed cost models in order to assess how well they predict the execution time of other programs (not used in the calibration process) statically, without performing any runtime profiling with them. We have performed experiments with a large number of modes. We show results for the three most accurate cost models (according to a global accuracy comparison that will be presented later) plus the step model, which has special interest as we will also see later. More details about the experimental results can be found in [63, 64].

Table 3.2 compares the overall accuracy of the four cost models mentioned previously, for the two considered platforms. The last column shows the global error and it is an indicator of the amount of deviation of the execution times estimated by each cost model with respect to the observed values. As global error we take the square mean of the errors in each example being considered. By considering both platforms in combination we can conclude that the more accurate cost model is the one consisting of steps, instruct, outstruct, inptr, and outptr. This cost model has an overall error of 14.66 % in platform “PPC” and 31.06 % in “Intel”. In “Intel” (obviously a more challenging platform) the model consisting of steps, instruct, outstruct, and outptr appears to be the best. This coincides with our intuition that taking into account a comparatively large number of lower-level operations should improve accuracy. However, such components should contribute significantly to the model in order to avoid noise introduction. It is also interesting to see that including *nargs* in the cost model does not further improve accuracy, as expected, since nargs is not independent from the four components instruct, outstruct, inptr, outptr. In fact, including this component results in a less precise model in both platforms, due to the noise introduced in the model. Also, the cost model step deserves special mention, since it is the simplest one and, at least for the given examples, the error is smaller than we expected and better than more complex cost models not shown in the tables.

Overall we believe that the results are very encouraging in the sense that our combined framework predicts with an acceptable degree of accuracy the execution times of programs and paves the way for even more accurate analyses by including additional parameters. We believe this is an encouraging result, since using a one-time profiling for estimating execution times of other, unrelated programs is clearly a challenging goal.

3.4.7 Related Work

As mentioned before, [31, 30] presented a method for automatically inferring functions which capture an upper bound on the number of *execution steps* that a procedure will execute as a function of the size of its input data. In [54, 53] the method of [31, 53] was fully automated in the context of a practical compiler and in [32, 53] a similar approach was applied in order to also obtain lower bounds. Our work extends these approaches from the inference of number of execution steps. We introduce models parametrized by an arbitrary number of constants or functions bounding the cost of basic operations, proposing a method for adjusting the constants via calibration programs, and showing how the approach can be used for inferring execution time.

Worst case execution time (WCET) estimation have been studied for imperative languages and for different application domains. Thiele and Wilhelm [87] described threats to time-predictability of safety-critical embedded systems which have to satisfy hard real-time constraints and proposed design principles that support time predictability for such systems. In [92, 91] the principles of their Timing-Analysis method, which uses Abstract Interpretation to predict the system's behavior on the underlying processors components and Integer Linear Programming to determine a worst-case path through the program. Berg et al. [15] introduce a set of design principles that aim to make processor architectures amenable to static timing analysis. Bel-Hadj-Aissa et al. [3, 2], described the main issues when computing WCET on very constrained devices in terms of memory and time, and proposed an scheme to compute WCET in the CAMILLE operating system for smart cards. Eisinger et al. [36] presented an automated timing anomaly identification approach. They also validated the method by applying it to a simplified microprocessor using a commercial model checking tool.

However these methods do not infer cost functions of input data sizes but rather absolute maximum execution times. To this end, each loop has to be hand annotated indicating the maximum number of iterations that it will execute. In contrast in our approach this is inferred as a function of input data metrics.

In the particular context of Java, Bate et al. [14] proposed a framework for providing a portable (i.e. hardware and language independent) WCET analysis for the Java platform. It is achieved by separating the WCET analysis process in three separated stages and by analyzing the Java Byte Code, not the high level source code. The three stages are: a Java virtual machine platform-dependent (low-level) analysis, a software dependent (high-level) analysis and an on-line integration step. However, this approach again requires that the code has to include annotations on the worst case behavior of its constructs (i.e. maximum loop bounds) in order for the code to be analyzable. Bernat et. al. [19] addressed two issues to take under consideration when analyzing Java Byte Code: how to extract data and control flow information from Java Byte Code programs and how to provide a compiler/language independent mechanism to introduce WCET annotations in the source code.

Chapter 4

Alias Control Types

4.1 Universe Type System

Alias characterisations simplify reasoning about programs [34]: they enable modular verification, facilitate thread synchronisation, and allow programmers to exchange internal representations of data structures. Ownership types [26, 25] and Universe types [69] are mechanisms for *characterising* aliasing in object oriented programming languages. They organise the heap into a hierarchical structure of nested non-overlapping *contexts* where every object is contained in one such context. Each context is characterised by an *object*, which is said to *own* all the objects contained directly in that context.

Example 1 *Figure 4.1 shows one possible hierarchical organisation of eight objects located at addresses* 1 *to* 8. *Each rounded box indicates the address and class of an object, whereas the adjoining dotted lines forming a box denote the context (ownership) that object characterises. For instance, the topmost round box denotes an object at address* 1 *of class* D*, the round box beneath it denotes an object at address* 2 *of class* C*, etc. Moreover, the dotted lines originating from the topmost round box denote that the object at address* 1 *owns the object at address* 2. *Similarly, the dotted box (context) of the object at address* 2 *denotes that this object owns the three objects at addresses* 3*,* 4 *and* 5.

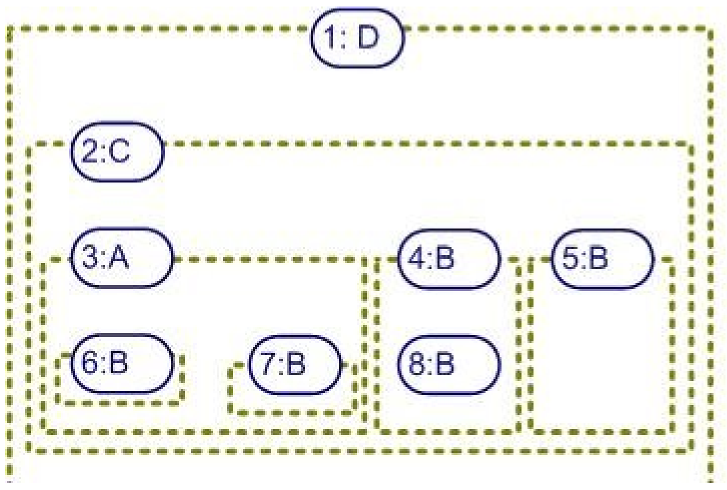

Figure 4.1: Ownership Hierarchical Structure

In the Universe Type System [69, 34, 52], a context hierarchy such as the one in Figure 4.1 is induced by extending types with Universe annotations, which range over `rep`, `peer` and `any`. A field typed with a Universe modifier `rep` denotes that the object referenced by it must be within the context of the current object; a field typed with a Universe modifier `peer` denotes that the object referenced by it must be within the context that also contains the current object; a field typed with a Universe modifier `any` is agnostic about the context containing the object referenced by the field.

Example 2 *The following is an example class definition in the Universe Type System.*

```
class A {
    rep B br;
    peer B bp;
    any B ba;
}
```

Class A contains three fields of class B, but each field has a different Universe modifier. Field `br` *has modifier* `rep` *which means it can only point to an object in the context of the current object; field* `bp` *has modifier* `peer` *which means it can only point to an object that is a peer of the current object; field* `ba` *has modifier* `any` *which means it can point to any object of class B in the Universe hierarchy. Thus, if this class definition referred to the object at address* 3 *in Figure 4.1, then the field* `br` *could only point to the objects at addresses* 6 *and* 7*, the field* `bp` *could only point to the objects at addresses* 4 *and* 5 *and field* `ba` *could point to any object in the hierarchy. We here note how Universe contexts* characterise *aliasing but do not restrict it. More specifically, the field* `ba` *is allowed to* cross into *the context of the object at address* 4 *so as to point to the object at address* 8*.*

So far, we have concentrated on the following three areas:

Universe Java The formalisation and proof of soundness of a minimal object oriented language with Universe Types.

Generic Universe Java The extension of Universe Java to Generic Java.

Concurrent Universe Java The use of Universe Types to administer race conditions and atomicity in a concurrent version of Universe Java.

The full reports on these tasks containing complete formalisations, examples and proofs will appear shortly at [27, 33, 28].

4.2 UJ - Universe Java

We introduce Universe Types into a subset of the Java programming language and obtain what we call Universe Java, or simply UJ [27]. For the purposes of this discussion, we limit ourselves to the basic constructs of Java such as field access, field update and method invocation.

Preliminaries We assume a set of class names $c, c_1, c_2, ... \in C$, a subclass relation on class names $c \leq c$ and a set of addresses $\iota, \iota_1, \iota_2 \in A$. We also assume the variables $u, u_1, u_2, \ldots$ which range over Universe annotations $\mathtt{U} = \{\mathtt{rep}, \mathtt{peer}, \mathtt{any}\}$. We sometimes refer to the set of *extended* Universes $\mathtt{EU} = \mathtt{U} \cup \{\mathtt{this}\}$. Static types $t_s \in T_s$ are written as $u\,c$, where u is the Universe annotation and c is the class name. On the other hand, dynamic types $t_d \in T_d$ are written as $\iota\,c$ where ι is the address of the owner and c is the name of the class of the object at ι.

In UJ, executions need to hold the additional owner information for every object. Thus, the heap h would be the partial function

$$h:\ A \rightharpoonup T_d \times FM \tag{4.1}$$

where the dynamic type T_d holds the additional owner information and $fm \in FM$ are partial functions mapping field names $f \in F$ to addresses

$$fm\colon F \rightharpoonup A \tag{4.2}$$

Objects in a heap, denoted by the address where they are located, are thus assigned dynamic types; we defined the following judgement

$$\frac{h(\iota) = t_d, fm}{h \vdash \iota : t_d}$$

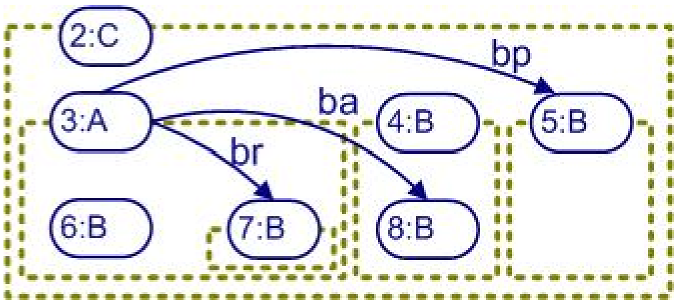

Figure 4.2: A Typical Heap with a Universes Hierarchical Structure

Example 3 *Assuming that our program defines the class names* `A`,`B` *and* `C`*, then in heap* h_1*, represented graphically in Figure 4.2 we would have*

$$h_1(3) \;=\; \underbrace{\overbrace{2}^{\textit{owner}}\;\texttt{A}}_{\textit{dynamic type}}\;,\overbrace{\{\texttt{br} \mapsto 7,\ \texttt{bp} \mapsto 5,\ \texttt{ba} \mapsto 8\}}^{\textit{field map}}$$

We also use the judgement

$$h_1 \vdash 3 :\; 2\,\texttt{A}$$

to assert that the dynamic type to the object residing at address 3 *in heap* h_1 *is* $2\,\texttt{A}$*.*

View Dependent Static Types A central aspect of UJ is that static types depend on the point of view of the address from where they are accessed. This is formalised by the judgement

$$h, \iota \vdash \iota' : t_s \tag{4.3}$$

meaning that from ι in heap h, the address ι' is viewed to have type t_s. The definition for this judgement is based on a function $\uparrow_{h,\iota}(-)$ which lifts a dynamic type to a static type, relative to the current view denoted by the tuple (h, ι). In essence, this function converts the owner information ι' in the dynamic type to Universe modifiers u, according to the view (h, ι). This function is defined as:

$$\uparrow_{(h,\iota)}(\iota'\,c) \overset{\textbf{def}}{=} \begin{cases} \texttt{rep}\,c & \text{if } \iota' = \iota \\ \texttt{peer}\,c & \text{if } h \vdash \iota : (\iota', c') \text{ for some class } c' \\ \texttt{any}\,c & \text{otherwise} \end{cases} \tag{4.4}$$

With the above function we can thus define our view dependent static type judgement using these two rules

$$\frac{h \vdash \iota' : t_d}{h, \iota \vdash \iota' : \uparrow_{(h,\iota)}(t_d)} \qquad \frac{h, \iota \vdash \iota' : u\,c \quad c \leq c'}{\begin{array}{c} h, \iota \vdash \iota' : u\,c' \\ h, \iota \vdash \iota' : \texttt{any}\,c' \end{array}}$$

Example 4 *For the heap* h_1 *of Figure 4.2 we can deduce the following static types from the point of view of the object at address* 3*:*

$$h_1, 3 \vdash 4 : \texttt{peer}\,B$$

	$u \rhd u'$	peer	rep	any
u	this	peer	rep	any
	peer	peer	any	any
	rep	rep	any	any
	any	any	any	any

	$u \blacktriangleright u'$	peer	rep	any
u	this	peer	rep	any
	peer	peer	any	any
	rep	any	peer	any
	any	any	any	any

Figure 4.3: The Universe composition and decomposition operators

that is, the object at address 4 *is a* ***peer*** *object (to the object at address* 3*) of class* B*. Similarly, we deduce that*

$$h_1, 3 \vdash 7 : \textit{rep}\, B \quad \textit{and} \quad h_1, 3 \vdash 8 : \textit{any}\, B$$

Assuming another class E *such that* $B \leq E$*, we can also weaken the former two judgements as:*

$$h_1, 3 \vdash 7 : \textit{any}\, E \qquad\qquad h_1, 3 \vdash 8 : \textit{any}\, E$$

Based on the definition of view dependent static types, we can state what it means for a heap h to be well-formed in UJ, denoted as $\models h$. Intuitively, a heap is well-formed if the field values of every object in the heap respect (are a subtype of) the field type declared for the class of that object.

Definition 4.2.1 (Well Formed Heap)

$$\models h \overset{def}{=} \begin{cases} \forall \iota \in \textbf{\textit{dom}}(h). & \textit{if}\, h(\iota) = ((\iota'\, c), \textit{fm}) \\ & \textit{and class } c \textit{ has field } f \textit{ with type } t_s \\ & \textit{then}\, h, \iota \vdash \textit{fm}(f) : t_s \end{cases}$$

4.2.1 Types, and Type Soundness

The typing rules of UJ, omitted from this document, make use of Universe modifier composition ($\rhd$) and decomposition ($\blacktriangleright$) operations with the following signatures:

$$\begin{aligned} \rhd &: \ \mathtt{EU} \times \mathtt{U} \to \mathtt{U} \\ \blacktriangleright &: \ \mathtt{EU} \times \mathtt{U} \to \mathtt{U} \end{aligned}$$

The table on the left hand side of Figure 4.3 defines the operation $\rhd$, which "composes" two viewpoints, in the sense that if some object considers a second object to be at at u, and, if the second object considers a third object to be at u', then the first object considers the third object to be at $u \rhd u'$. This is expressed by the following lemma, where we use $h, \iota \vdash \iota' : u$ as a shorthand for $h, \iota \vdash \iota' : u\, c$ for some c:

Lemma 2 (Universe Modifier Composition) $h, \iota \vdash \iota' : u,\ h, \iota' \vdash \iota'' : u' \implies h, \iota \vdash \iota'' : u \rhd u'$.

On the other hand, the decomposition operator $\blacktriangleright$, defined on the right hand side of figure 4.3, is, in some sense, the opposite of $\rhd$. Namely, $u \blacktriangleright u'$ returns a universe modifier u'' such that $u \rhd u'' = u'$ if it exists and is unique (formally, if $u \blacktriangleright u' = u''$ then $u \rhd u'' = u'$ and $u'' \neq$ any or for all u''': $u \rhd u''' = u'$ implies $u''' = u''$).

The operation $\blacktriangleright$ "decomposes" viewpoints, in the sense that if some object considers a second object to be at u, and the first object considers a third object to be at u', then the second object considers the third object to be $u \blacktriangleright u'$. This is expressed by the following lemma:

Lemma 3 (Universe Modifier Decomposition) $h, \iota \vdash \iota' : u,\ h, \iota \vdash \iota'' : u' \implies h, \iota' \vdash \iota'' : u \blacktriangleright u'$.

Example 5 *In the heap* h_1 *of Figure 4.2, the following judgements* α, β, γ *and* δ *hold:*
(α) $h_1, 2 \vdash 3$: ***rep*** (β) $h_1, 3 \vdash 5$: ***peer*** (γ) $h_1, 2 \vdash 5$: ***rep*** (δ) $h_1, 3 \vdash 5$: ***peer***
Judgment (γ) *also follows from* (α) *and* (β) *and Lemma 2; judgement* (δ) *also follows from* (α) *and* (β) *and Lemma 3.*

Typing has the format

$$\Gamma \vdash e : t_s$$

where Γ is the standard typing context assigning types to `this` and method parameters. The type rules are shown in detail in [27]; here we only show the type rules for field reading and field assignment

$$\text{(fld-acc)}\quad \frac{\Gamma \vdash e :\ u\,c \qquad \text{class } c \text{ has a field } f \text{ of type } u'\,c'}{\Gamma \vdash e.f :\ u \triangleright u'\,c'} \qquad\qquad \text{(fld-ass)}\quad \frac{\Gamma \vdash e :\ u\,c \qquad \text{class } c \text{ has a field } f \text{ of type } u'\,c' \qquad \Gamma \vdash e' :\ u'\,c'}{\Gamma \vdash e.f\ =\ e' :\ u \blacktriangleright u'\,c'}$$

Rule (`fld-acc`) is justified by lemma 2, while rule (`fld-ass`) is justified by lemma 3.

Execution is described in terms of large-step semantics of the form

$$e, s, h \ \rightsquigarrow \iota, h'$$

where s is a stack mapping the identifier `this` and method parameters to addresses. We define well-formed stacks, denoted as $\Gamma, h \vdash s$, to mean that the objects mapped to by the identifiers in the stack correspond to the types assigned to those identifiers in Γ. We then prove that the execution of a well-typed expression e with respect to a well formed heap and stack preserves both the type of the expression and the well-formedness of the heap.

Theorem 4.2.2 (Type Soundness for UJ) *For any h, s, Γ, e, t_s :*

$$\models h, \quad \Gamma, h \vdash s, \quad \Gamma \vdash e : t_s, \quad e, s, h \ \rightsquigarrow \iota, h' \qquad \Longrightarrow \qquad \models h', \quad h', s(\textit{this}) \vdash \iota : t_s.$$

4.2.2 Encapsulation

In Universe Types, encapsulation is interpreted to be the *owner-as-modifier* property, meaning that the fields of an object can only be modified through method calls made on the owner of that object. This restriction however does not apply to field access, which can be performed by circumventing the owner.

To guarantee this encapsulation property in UJ, we define a more restrictive type system,

$$\Gamma \vdash_{\mathsf{enc}} e : t_s$$

which typechecks the expression e if e observes the owner as modifier restriction, and if $\Gamma \vdash e : t_s$ (i. e., if it typechecks in the previous type system).

To be able to demonstrate that type safety implies encapsulation, we extend our operational semantics for UJ to keep track of the receivers of any methods called during execution. This is reflected in the new format of the operational semantics,

$$e, s, h \ \overset{\mu}{\rightsquigarrow}\ v, h'$$

where μ is a set of addresses $\{\iota_1, \dots, \iota_n\} \subset A$. When a well-typed expression reduces to some value, after having invoked methods on the objects contained in μ, the owners of any objects whose fields were modified during execution must be μ. This is stated in the following Theorem.

Theorem 4.2.3 (Encapsulation for UJ) *For all h, s, Γ, e, μ, t_s such that $\models h$, and $\Gamma, h \vdash s$, $\Gamma \vdash e : t_s$ and $e, s, h \ \overset{\mu}{\rightsquigarrow}\ \iota, h'$*

$$h(\iota).f \textit{ is defined, and } h(\iota).f \neq h'(\iota).f \quad \Longrightarrow \quad \exists \iota'' \in \mu, \textit{with } h \vdash \iota :\ \iota''\ \_.$$

4.2.3 Universe Types and the Bicolano Logic

We extend the Bicolano Logic [74] to accommodate the additional heap structure introduced by universes, in a style similar to [51]. In particular, we specify an additional function on heaps called `owner` with signature

```
Parameter owner : t -> Location -> option Owner.
```

where `Owner` is defined as:

```
Inductive Owner : Set :=
  | Ownr: Location -> Owner
  | World : Owner
```

The function `owner` extends heaps to hold owner information for any existing location: `owner(h,l)` returns the owner of the object at location `l` in heap `h`, if `l` exists in `h`, and `None`, otherwise. Owners are defined through the datatype `Owner` above, which is either another location, `Ownr l`, or the topmost owner, `World`. We also extend the signature of the heap operation `new` to

```
Parameter new : t -> Program -> LocationType -> Owner -> option(Location * t)
```

so that it requires an additional parameter of type `Owner` when creating a fresh location, which determines the owner of the fresh location. The extended signature of `new` entails cosmetic adaptations to existing invariants of this operation; one instance would be

```
Parameter new_fresh_location :
  forall (h:t) (p:Program) (lt:LocationType) (o:Owner) (loc:Location)  (h':t),
     new h p lt o = Some (loc1 ,h') -> typeof h loc1 = None.
```

The redefined `new` operation also requires two new invariants. The first invariant states that if a location is created successfully, then its owner is the one specified.

```
Parameter new_fresh_location_owner :
  forall (h:t) (p:Program) (lt:LocationType) (o:Owner) (loc:Location)  (h':t),
     new h p lt o = Some (loc ,h') -> onwer h' loc = o.
```

The second invariant states that a fresh location in a heap `h` owned by a location `l` can only be created if `l` already exists in `h`.

```
Parameter new_fresh_location_owner_exists :
  forall (h:t) (p:Program) (lt:LocationType)  (loc, loc':Location)  (h':t),
     new h p lt Ownr(loc) = Some (loc' ,h') -> typeof h loc <> None.
```

Finally, we specify invariants ensuring that the owner of an existing location does not change as a result of modifications to the heap. We thus need to ensure that the heap operations `update` and `new` do not alter the `owner` function:

```
Parameter new_onwer_old :
  forall (h:t) (p:Program) (lt:LocationType) (o:Owner) (loc loc':Location) (h':t),
     new h p lt o = Some (loc,h') -> loc <> loc' -> onwer h' loc' = onwer h loc'.

Parameter owner_update_same :
  forall (h:t) (loc:Location) (am:AddressingMode) (v:value),
     owner (update h am v) loc = owner h loc.
```

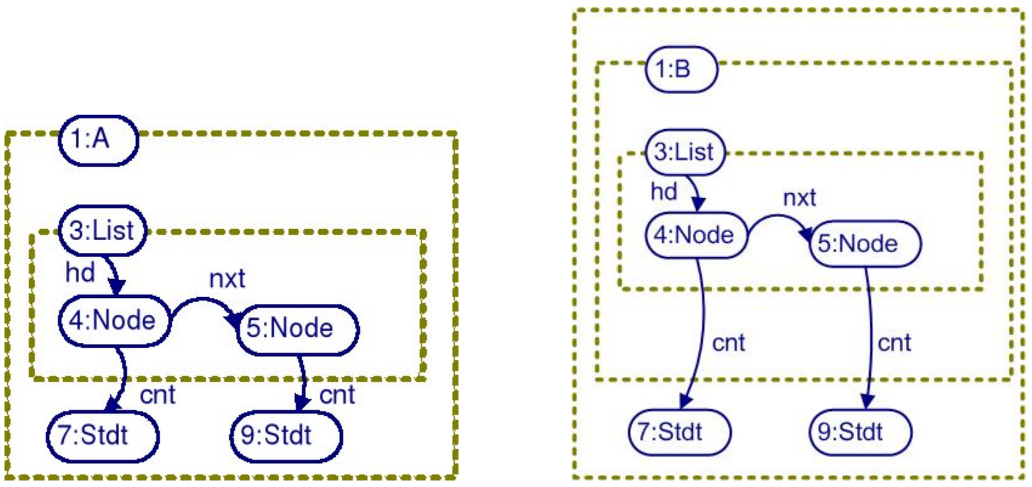

Figure 4.4: Heap Examples in Generic Universe Java

4.3 GUJ - Generic Universe Java

We extend Universe Java to handle generics, which now form part of the official release of Java 1.5 [49, 42], and refer to this extension as Generic Universe Java (GUJ) [33]. In Generic Java, classes have parameters which can be bound by types: since in Universe Java, types are made up of a Universe modifier and a class, GUJ class parameters in generic class definitions are bound by Universe modifiers *and* classes.

Example 6 *Consider the class definitions with* generic *Universe modifiers below. The class parameter S in both the generic classes List < S > and Node < S > is bound by the type* `any Object`*.*

```
class List <S extends any Object> { rep Node<S> hd;}

class Node <S extends any Object> {
    peer Node<S> nxt;
    S cnt; }

class Student {...}
```

In class A defined below, we instantiate the class parameter S of List < S > (and, as a consequence, the type parameter S of Node < S > as well) by the type `rep` *Student. This allows heaps with ownership structure such as the one on the left hand side of Figure 4.4, where objects of class Student stored in the list are owned by the same owner of the list.*

```
class A { rep List<rep Student> z ...}
```

In class B defined below, we instantiate the class parameter of List < S > with the type `peer` *Student, which gives us heaps where any objects of class Student stored in the list are peers of the owner of the list; the right hand side diagram of Figure 4.4 depicts one such case of these heaps.*

```
class B { rep List<peer Student> z ...}
```

Formalisation To accommodate generics, we extend the structure of static types $t_s \in T_s$ and dynamic types $t_d \in T_d$ to

$$
\begin{aligned}
t_s &::= u\,c\,\sigma \\
t_d &::= \iota\,c\,\rho
\end{aligned}
$$

where σ and ρ are substitutions from type variables to static types and dynamic types respectively.

$$\begin{array}{rcl} \sigma \in \texttt{Sub} & ::= & \texttt{Var} \rightharpoonup T_s \\ \rho \in \texttt{DSub} & ::= & \texttt{Var} \rightharpoonup T_d \end{array}$$

We retain the former definition of field maps (4.2) and heaps (4.1) from Section 4.2. Nevertheless, we note that, with our reformulation of dynamic types T_d, the heap now keeps additional information regarding type variable substitution during executions.

$$h : A \rightharpoonup T_d \times FM$$

Example 7 *The type* **rep** *List <* **rep** *Student >, used in Example 6, is represented by the static type triple*

$$\underbrace{\texttt{rep}}_{u}\ \underbrace{List}_{c}\ \underbrace{\{S \mapsto \texttt{rep}\, Student\, \emptyset\}}_{\sigma}$$

Then, in the leftmost heap of Figure 4.4, the dynamic type

$$\underbrace{1}_{\iota}\ \underbrace{List}_{c}\ \underbrace{\{S \mapsto 1\, Student\, \emptyset\}}_{\rho}$$

can be assigned to an object referenced by a field with the static type above.

Similar to Section 4.2, we also have a judgement assigning static types to addresses with respect to a particular view $h, \iota \vdash \iota' : t_s$, based on a function $\uparrow_{h,\iota}(-)$ lifting dynamic types to static types relative to the view (h, ι). Since types in GUJ now contain type variable substitutions, we overload $\uparrow_{h,\iota}(-)$ so that it lifts

- dynamic types T_d to static types T_s (as before),
- substitutions from type variables to dynamic types DSub to substitutions from type variables to static types Sub.

Thus, the function has the respective two signatures

$$\begin{array}{lll} \uparrow \ : & H \times A \times T_d \to T_s & \textit{(from Dynamic types to Static types)} \\ & H \times A \times \texttt{DSub} \to \texttt{Sub} & \textit{(from Var}\rightharpoonup\texttt{DSub} \textit{ to Var}\rightharpoonup\texttt{Sub)} \end{array}$$

and is defined through the the mutually recursive definitions

$$\uparrow_{h,\iota}(\rho) \overset{\textbf{def}}{=} \{S \mapsto \uparrow_{h,\iota}(t_d) \quad | \ S \in \textbf{dom}(\rho) \text{ and } t_d = \rho(S)\}$$

$$\uparrow_{h,\iota}(\iota', c, \rho) \overset{\textbf{def}}{=} \begin{cases} \texttt{rep}\ c\ \uparrow_{h,\iota}(\rho) & \text{if } \iota' = \iota \\ \texttt{peer}\ c\ \uparrow_{h,\iota}(\rho) & \text{if } h \vdash \iota :\ \iota' c' \text{ for some class } c' \\ \texttt{any}\ c\ \uparrow_{h,\iota}(\rho) & \text{otherwise} \end{cases}$$

As a result, the view dependent static type judgement for GUJ is defined using the rules

$$\frac{h \vdash \iota' : t_d}{h, \iota \vdash \iota' : \uparrow_{(h,\iota)}(t_d)} \qquad \frac{h, \iota \vdash \iota' :\ u\, c\, \rho \quad c \le c'}{\begin{array}{c} h, \iota \vdash \iota' :\ u\, c'\, \rho \\ h, \iota \vdash \iota' :\ \texttt{any}\, c'\, \rho \end{array}}$$

Example 8 *With h_3 we refer to the leftmost heap in Figure 4.4. Then, the following judgements, assigning addresses to dynamic types,*

$$\begin{array}{lcl} h_3 & \vdash & 3 :\ 1\ List\ \{S \mapsto 1\, Student\ \emptyset\} \\ h_3 & \vdash & 4 :\ 3\ Node\ \{S \mapsto 1\, Student\ \emptyset\} \\ h_3 & \vdash & 7 :\ 1\ Student\ \emptyset \end{array}$$

hold. Using the above judgements, and the definition of the view dependent static judgments, we obtain

$$\begin{aligned} h_3, 3 &\vdash 4 : \texttt{rep}\ Node\ \{S \mapsto \texttt{peer}\ Student\ \emptyset\} \\ h_3, 3 &\vdash 7 : \texttt{peer}\ Student\ \emptyset \end{aligned}$$

Despite the reformulation of types in GUJ, which now also hold class parameter substitutions, our judgements still allow us to inherit Definition 4.2.1 for well formed heaps, i. e., $\models h$. We also inherit the notion of a well formed stack s, denoted as $\Gamma, h \vdash s$. This permits us to state, in standard fashion, the main theorem for our formalisation of GUJ.

Theorem 4.3.1 (Type Soundness for GUJ) *For any h, s, Γ, e, t_s :*

$$\models h, \quad \Gamma, h \vdash s, \quad \Gamma \vdash e : t_s, \quad e, s, h \rightsquigarrow \iota, h' \quad \Longrightarrow \quad \models h', \quad h', s(\texttt{this}) \vdash \iota : t_s.$$

4.4 UJ and Concurrency

The Universe ownership relation in UJ provides a natural way to characterise non-overlapping nested groups of objects in a heap. We therefore exploit this structure in a Java with multiple concurrent threads [28] to guarantee certain properties during execution, namely

- the prevention of data races.
- the atomicity of execution of atomic blocks of code.

Ownership and Locking Based on previous work [39, **?**] we explore the use of *locking* groups of objects so as to gain *exclusive access* to such objects and thus avoid interference from other threads of execution. By simply locking a particular object in a well-formed heap in UJ, a thread would gain exclusive access to all of the objects in the representation of that object (it however would not give exclusive access to the owner itself).

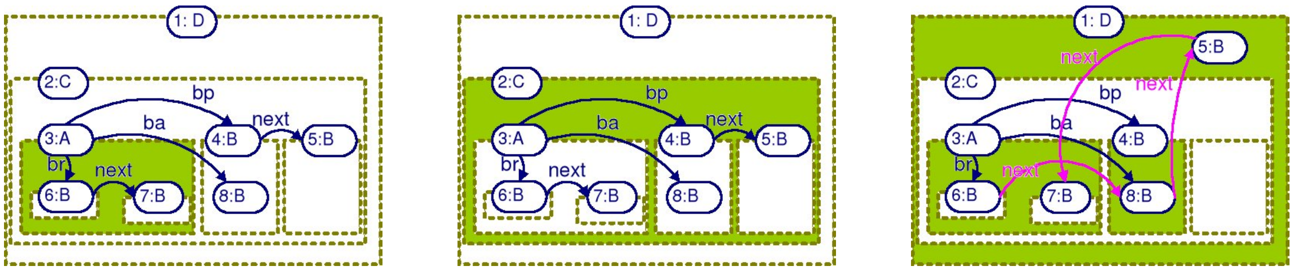

Figure 4.5: Using the Universes Hierarchical Heap Structure for Locking

Example 9 *Consider the class definitions with Universe modifiers below. If we execute method* $m1()$ *on object* 3 *in the leftmost runtime heap depicted in Figure 4.5, then locking* **this** *would lock all the objects owned by the object* 3*; this is represented by the shaded area. Since the local variable* br1 *is tagged as* **rep** *and the local field* next *in class* B *is tagged as* **peer***, we can statically infer that this single lock suffices to guarantee that there will be no data races interfering with the objects accesses during the iteration in method* $m1()$ *and that every execution of this method is atomic.*

```
class B { peer B next;}
class A {
    rep B br; peer B bp; any B ba;

    void m1(){
        rep B br1 = br;
        lock (this){
            while (br1){
                br1 = br1.next;} } }

    void m2(){
        peer B bp1 = bp;
        lock (bp1.owner){
            while (bp1){
                bp1 = bp1.next;} } }

    void m3(){
        any B ba1 = ba;
        while (ba1){
            lock (ba1.owner){
                ba1 = ba1.next;} } }
```

Similarly for method $m2()$*, locking* bp1.*owner would lock all the objects in the representation of the object of class C at address* 3*, that is the objects at addresses* 3*,* 4 *and* 5 *(refer to middle heap in Figure 4.5). Similar to the previous method, since the local field we iterate on in method* $m2()$ *has modifier* **peer** *(and the local field* next *in class B is tagged as* **peer**)*, we can statically infer that this single lock is sufficient to guarantee no data races and also guarantee atomicity during the iteration of the method.*

Unfortunately, similar conclusions can not be statically determined for the iteration in method $m3()$ *since there are heap instances where the locking procedure in* $m3()$ *does not suffice to guarantee atomicity. For instance, if we execute* $m3()$ *on the leftmost and middle heap diagram in Figure 4.5, we happen to obtain no data races as well as atomicity. However, for the rightmost heap depicted in Figure 4.5, the execution of* $m3()$ *would result in three successive lockings of different objects, that guarantees the absence of data races but* does not *guarantee atomicity.*

Type Systems for Races and Atomicity The type system of Section 4.2 is extended so that when we typecheck an expression

$$\Gamma \vdash_{\mathbf{race}} e : t$$

we can statically guarantee that e locks objects appropriately so as to ensure that there are no data races when e executes in parallel with other expressions. This can be formalised as the following theorem, where $\rightsquigarrow$ now denotes *small step reductions* from *threads* (sets of expressions) to *threads.*

Theorem 4.4.1 (Data Race Soundness) *For any heap h, expressions $e_1, \ldots, e_n$, any $i, j \in \{1, ..., m\}$ if $\models h$, and $\forall i{=}1..n$: $\Gamma \vdash_{race} e_i : t_i$ then*

$$e_1, \ldots, e_n, h \;\rightsquigarrow^*\; e'_1, \ldots, e'_m, h', \quad e'_i \textit{ contains } \iota.f, \quad e'_j \textit{ contains } \iota.f \qquad \Longrightarrow \qquad i = j.$$

We also extend the type system for UJ to guarantee atomicity of execution for an expression e. Exploiting the local reasoning allowed by Universe Types, we devise typing rules so as to typecheck an expression as

$$\Gamma \vdash_{\mathbf{atom}} e : t$$

This would then give us a static guarantee that whenever e executes in parallel with other expressions, it locks objects appropriately so that any execution interleaving with other threads would still yield the same result as if e was executed atomically (without interleaving). This property can be formalised as the following Theorem.

Theorem 4.4.2 (Atomicity Soundness) *For any well formed heap $\models h$, if $\Gamma \vdash_{\mathit{atom}} e : t$, and if $\Gamma \vdash e_i : t_i$ for all $i \in \{1, ..., n\}$, then*

$$e, e_1, \ldots, e_n, h \leadsto^\star v, e'_1, \ldots, e'_m, h' \quad \Longrightarrow \quad \exists e''_1, ..., e''_k,\ h'',\ h'''. \begin{cases} e, e_1, \ldots, e_n, h \leadsto^\star e, e''_1, \ldots, e''_k, h'' \\ e, h'' \leadsto^\star v, h''' \\ e''_1, \ldots, e''_k, h''' \leadsto^\star e'_1, \ldots, e'_m, h' \end{cases}$$

4.5 Future Work

We plan to extend our type systems to larger fragments of Java, including constructs such as exceptions and exception handling. More importantly however, 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.

4.6 Related Work

The closest work to Universe Types and UJ, adopting the *owner-as-modifier* principle is perhaps [57]. Through what they call Effective Ownership Types, they approach the problem of representation exposure by restricting side effects rather than aliasing. Similar to the Universe modifier `any`, objects allow non-mutating accesses of their fields through side effect free methods calls on other objects. Subsequently, [56] takes this work one step further, breaking the owner-as-modifier property by separating the properties of object accessibility and reference capability, allowing more refined static constraints. They use context variance mechanisms to abstract over object ownership while extending methods and fields types with accessibility parameters, permitting mutations up to those accessibility levels. [55], which predates the above, attempts to offer more flexibility than the single-owner restriction dictated by ownership types by defining a class-based region-parametric type system which assigns objects to *regions* organised as a directed acyclic graph. The partial ordering of regions imposes constraints on region reachability and reference cycles are limited to objects belonging to the same region. In contrast to work in [57, 56] the main aim of this work is to guarantee reference acyclicity rather than encapsulation.

Generic Ownership [75] relates closely to GUJ in that it provides an abstraction that adds ownership on top of genericity. Through examples, they argue that the additional ownership information fits smoothly in a language with type genericity and is not too taxing in terms of additional annotation. They also provide an implementation, integrating Generic Ownership into Java and call it OGJ.

The STARS programming model [6] use packages, gate classes and scoped classes in Real-Time Java to partition the heap into distinct logical blocks of memory called scoped memory. Such a partitioning enables static checks for controlled sharing of references in multithreading and memory management. Elsewhere, [65] employ the dominator relationship and ownership structures to great effect to analyse the shapes of large heaps in real-world applications and generate meaningful summaries of such heaps.

Chapter 5

Conclusion

In this deliverable we have described program analyses for a security, resource consumption and access, and alias control. In view of the remit of MOBIUS, it is important that these formalisms can be related to the mobile code format JVML, and the verification structure developed for bytecode in WP3 (see deliverable D3.1). This connection can be achieved in various ways:

- by a formulation of the analysis on the level of bytecode, as has been carried out for the information flow type system. In this case, the connection to the formalised bytecode verification infrastructure is most naturally achieved by formalising the soundness proof w.r.t. the Bicolano operational semantics.
- by a a formulation for bytecode program phrases, as carried out in the section on heap space analysis. Complementing a possible verification in Bicolano, we have presented a formalised soundness proof with respect to the MOBIUS bytecode logic.
- as a set of derived assertions in the sense of MRG for bytecode phrases where the granularity of the rules is determined by the compilation strategy and a formulation of the analysis at a higher language level. Although neither the compilation of the program nor the translation of the typing derivations needs to be formalised, this approach is difficult to adapt to arbitrary bytecode as it makes use of the bytecode structure afforded by the compilation from high-level code. For example, no null pointer exceptions can occur in the getfield operations resulting from the compilation of a pattern match. For a more complex example consider [17], where linearity conditions at the high level ensure the absence of aliasing at the low-level.
- by an implementation of the analysis (with or without inference) in the theorem prover, including selected parts of the analysis framework (e.g. properties of fixed points and lattices in the case of abstract interpretation). Our work on using abstract interpretation to certify constant bounds on heap consumption [23] is an example of this approach.

Common to these four approaches is the formalisation of the soundness property afforded by each type system with respect to Bicolano and/or the MOBIUS base logic. This has been achieved already or can readily be extracted from the hand-written soundness proof for all the type systems presented in this deliverable. Work in the next nine months will then involve translation of typing derivations into proofs about Bicolano. In many cases the type systems address Bytecode directly (secure information flow, constant heap usage, type system for access permission), and representations in the MOBIUS tool suite (Bicolano, MOBIUS base logic) have been completed or are planned to be completed within the next months.

As discussed in Section 3.3.5, the integration of the type system for resource managers with the Bicolano/base logic infrastructure will take the form of derived judgements, and will include work towards type inference.

The type system for input-dependent heap space from Section 3.1.2 will need to be refined and downsized to fit the requirements of MOBIUS. This will be explored in Task 2.4 (advanced resource type systems). We anticipate that a potential formalisation will take the form of derived assertions.

The static analysis of Section 3.4 will be improved in precision by developing lower-level models and applying the approach to Java bytecode. The techniques will be generalized in Task 2.4 to infer upper and lower bound functions on parametrized resource descriptions. This will allow taking into account different classes of advanced resources with a common tool base and guaranteeing for example that mobile code will not exceed different kinds of resource bounds when running on a given device.

In the remaining time of this task, we will thus seek to further develop the link to the verification infrastructure for selected systems described in the present deliverable, and to develop further analyses as described in DIP2. The result of this work will be reported in the final deliverable for the present task (deliverable 2.4), the deliverables relating to certificate generation (WP4), and be an integral component of the software infrastructure developed in Task 2.6 and WP4.

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] N. Bel Hadj Aissa, C. Rippertand D. Deville, and G. Grimaud. A distributed WCET computation scheme for smart card operating systems. In *Workshop on Worst Case Execution Time Analysis*, Catania, Sicily, Italy, 2004.

[3] N. Bel Hadj Aissa, C. Rippert, and G. Grimaud. Distributing the WCET computation for embedded operating systems. In *Real-Time Systems Symposium*, Lisbone, Portugal, 2004.

[4] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In F. and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 380–397. Springer-Verlag, 2005.

[5] 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.

[6] 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.

[7] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] G. Barthe, D. Pichardie, and T. Rezk. Non-interference for low level languages. Technical report, INRIA, 2006.

[13] 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.

[14] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *IEEE Symposium on Object-oriented Real-time distributed Computing*, April 2002.

[15] C. Berg, J. Engblom, and R. Wilhelm. Requirements for and design of a processor with predictable timing. In Lothar Thiele and Reinhard Wilhelm, editors, *Workshop on Design of Systems with Predictable Behaviour*, volume 03471 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2004.

[16] M. Berger. *Towards Abstractions for Distributed Systems.* PhD thesis, Imperial College, Dept. of Computing, 2002.

[17] 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.

[18] 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.

[19] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using java byte code, 2000.

[20] F. Besson, T. Jensen, and D. Pichardie. A PCC architecture based on certified abstract interpretation. In *Emerging Applications of Abstract Interpretation.* Elsevier, 2006.

[21] Fréderic Besson, Guillaume Dufay, and Thomas Jensen. A formal model of access control for mobile interactive devices. In *ESORICS 2006* [38].

[22] 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.

[23] D. Cachera, Thomas P. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Formal Methods Europe*, pages 91–106, 2005.

[24] 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.

[25] 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.

[26] 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.

[27] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, and P. Müller. UJ: Type soundness for universe types. To appear at `http://slurp.doc.ic.ac.uk/pubs.html#uj06`.

[28] 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.

[29] 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.

[30] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[31] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *Programming Languages Design and Implementation*, pages 174–188. ACM Press, June 1990.

[32] 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.

[33] W. Dietl, S. Drossopoulou, and P. Müller. GUJ: Generic universe types. Preliminary version available from `http://www.sct.ethz.ch/publications/index.html`.

[34] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[35] 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.

[36] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Workshop on Design & Diagnostics of Electronic Circuits & Systems*, pages 15–20. IEEE Press, 2006.

[37] *Programming Languages and Systems: Proceedings of the 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006*, volume 3924 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[38] *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.

[39] C. Flanagan and M. Abadi. Types for safe locking. In *European Symposium on Programming*, pages 91–108, 1999.

[40] 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.

[41] 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.

[42] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition.* The Java Series. Addison-Wesley, 2005.

[43] 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.

[44] N. Heintze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Principles of Programming Languages*, pages 365–377. ACM Press, 1998.

[45] 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.

[46] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming*, 58(1-2):115–140, October 2005.

[47] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *ESOP 2006* [37], pages 22–37.

[48] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Principles of Programming Languages*, pages 81–92. ACM Press, 2002.

[49] 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.

[50] 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.

[51] 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`.

[52] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP 2006* [37], pages 115–130.

[53] P. López-García. *Non-failure Analysis and Granularity Control in Parallel Execution of Logic Programs.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 2000.

[54] P. López-García, M. Hermenegildo, and S. K. Debray. A methodology for granularity based control of parallelism in logic programs. *Journal of Symbolic Computation*, 22:715–734, 1996.

[55] Y. Lu and J. Potter. A type system for reachability and acyclicity. In A. P. Black, editor, *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 479–503. Springer-Verlag, 2005.

[56] Yi Lu and J. Potter. On ownership and accessibility. In *European Conference on Object-Oriented Programming*, pages 99–123, 2006.

[57] Yi Lu and J. Potter. Protecting representation with effect encapsulation. In *Principles of Programming Languages*, pages 359–371, 2006.

[58] 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.

[59] 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.

[60] H. Mantel, H. Sudbrock, and T. Krauß er. Combining different proof techniques for verifying information flow security. In G. Puebla, editor, *Logic-based Program Synthesis and Transformation*, volume Raporta di Ricerca CS-2006-5, Università Ca' Foscari Di Venezia, Venice, Italy, July 12–14 2006.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming*, pages 74–98, 2006.

[66] MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements, 2006. Available online from `http://mobius.inria.fr`.

[67] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[68] 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.

[69] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[70] 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/`.

[71] 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.

[72] 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.

[73] David A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS 2006* [38], pages 279–296.

[74] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [67], 2006.

[75] 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.

[76] 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.

[77] A. Reinhard. Analyse nebenläufiger programme unter intransitiven sicherheitspolitiken. Master's thesis, RWTH Aachen, May 2006.

[78] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Computer Security Foundations Workshop*, pages 177–189. IEEE Press, 2006.

[79] 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.

[80] 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.

[81] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, 2003.

[82] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop*, pages 200–215. IEEE Press, 2000.

[83] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations Workshop*, pages 255–269. IEEE Press, 2005.

[84] D. Sannella and M. Hofmann. Mobile resource guarantees. EU Project IST-2001-33149, 2002–2005. http://groups.inf.ed.ac.uk/mrg/.

[85] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Principles of Programming Languages*, pages 355–364, 1998.

[86] 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.

[87] L. Thiele and R. Wilhelm. Design for time-predictability. In *Workshop on Design of Systems with Predictable Behaviour*, volume 03471 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2004.

[88] 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.

[89] 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.

[90] D. Wackerly, W. Mendenhall, and R. Scheaffer. *Mathematical Statistics With Applications 5th Edition.* P W S Publishers, 1995.

[91] R. Wilhelm. Formal analysis of processor timing models. In S. Graf and L. Mounier, editors, *SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 1–4. Springer-Verlag, 2004.

[92] 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.

[93] Dachuan Yu and Nayeem Islam. A typed assembly language for confidentiality. In *ESOP 2006* [37], pages 162–179.

[94] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

[95] 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.