Certification using the Mobius Base Logic

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

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

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

1 Introduction: Role of the logic in Mobius

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

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

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

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

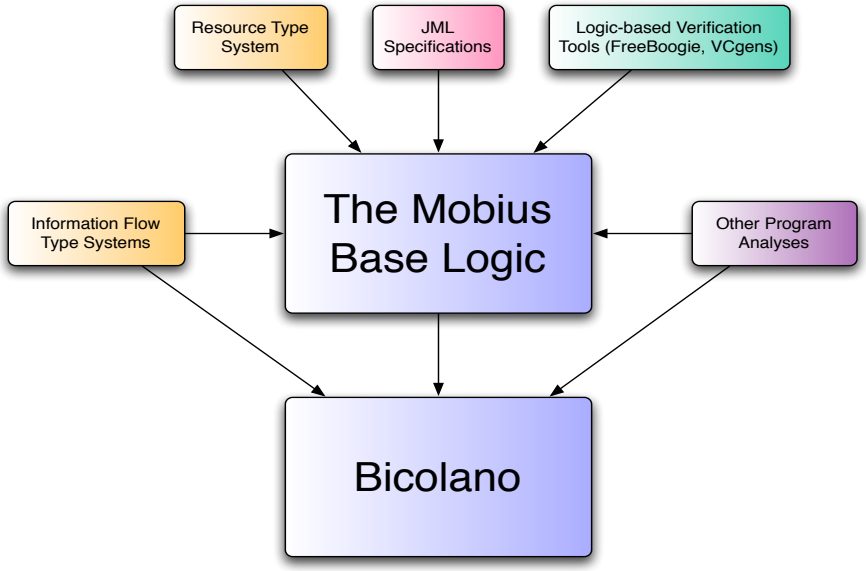

Fig. 1. Core components of the MOBIUS TCB

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

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

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

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

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

1.1 Related work

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

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

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

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

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

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

2 Bicolano

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

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

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

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

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

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

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

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

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

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

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

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

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

[3] The formalisation separates the small-step judgements for method invocations from the execution of basic instructions and jumps, and then defines a single recursive judgement combining the two. See [30] for the formal details.

Exceptional steps The judgement $\vdash_M\ l, s\ \Rightarrow_{\mathsf{excn}}\ h, a$ describes exceptional small steps where the execution of the instruction at program point M, l in state s results in the creation of a fresh exception object, located at address a in the heap h. In the case of method invocations, a single exceptional step is also observed by the callee if the invoked method raised an exception that could not be locally handled (cf. rule InvsExcn).

Small step judgements Non-exceptional and handled exceptional small steps are combined to the small step judgement $\vdash_M l, s \Rightarrow l', r$ using the two rules NormStep and ExcnStep. The reflexive transitive closure of this relation is denoted by $\vdash_M l, s \Rightarrow^* l', r$

Big-step judgements The judgement form $\vdash_M\ l, s \Downarrow t$ captures the execution of method M from the instruction at label l onwards, until the end of the method. This relation is defined by the three rules Comp, Vret and Uncaught.

Deep step judgements The judgement $\vdash_M\ l, s \Uparrow r$ is defined similarly to the big-step judgement, by the rules D-Refl, D-Trans D-Invs, and D-Uncaught. This judgement associates states across invocation boundaries, i.e. r may occur in a subframe of the method M. This is achieved by rule D-Invs which associates a call state of a (static) method with states reachable from the initial state of the callee. A similar rule for virtual methods is omitted from this presentation.

Small and big-step judgements are mutually recursive due to the occurrence of a big-step judgement in hypotheses of the rules for method invocations on the one hand and rule Comp on the other.

3 Base logic

This section outlines the non-resource-extended program logic.

3.1 Phrase-oriented assertions and judgements

The structure of assertions and judgements of the logic are governed by the requirement to enable the interpretation of type systems as well as the representation of core idioms of JML. High-level type systems typically associate types (in contexts) to program phrases. Compiling a well-formed program phrase into bytecode yields a code segment that is the postfix of a JVM method, i.e. all program points without control flow successors contain return instructions. Consequently, judgements in the logic associate assertions to a program label which represents the execution of the current method invocation from the current point (i.e. a state applicable at the program point) onwards. In case of method termination, a partial-correctness assertion (post-condition) applies that relates this current state to the return state. As the guarantee given by type soundness results often extends to infinite computations (e.g. type safety, i.e. absence of type errors), judgements furthermore include assertions that apply to non-terminating

$$\textsc{InvsNorm}\ \frac{M(l) = \mathsf{invokestatic}\ M' \qquad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Downarrow NormState(k, v)}{\vdash_M l, (O@O', S, h) \Rightarrow_{\mathsf{norm}} suc_M(l), (v :: O', S, k)}$$

$$\textsc{InvsExcn}\ \frac{M(l) = \mathsf{invokestatic}\ M' \qquad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Downarrow ExcnState(k, a)}{\vdash_M l, (O@O', S, h) \Rightarrow_{\mathsf{excn}} k, a}$$

$$\textsc{NormStep}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r}{\vdash_M l, s \Rightarrow l', r} \qquad \textsc{ExcnStep}\ \frac{\vdash_M l, (O, S, h) \Rightarrow_{\mathsf{excn}} k, a \quad getClass(k, a) = e \quad Handler(M, l, e) = l'}{\vdash_M l, (O, S, h) \Rightarrow l', ([a], S, k)}$$

$$\textsc{Comp}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Downarrow t}{\vdash_M l, s \Downarrow t} \qquad \textsc{Vret}\ \frac{M(l) = \mathsf{vreturn}}{\vdash_M l, (v :: O, S, h) \Downarrow NormState(h, v)}$$

$$\textsc{Uncaught}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \quad getClass(h, a) = e \quad Handler(M, l, e) = \emptyset}{\vdash_M l, s \Downarrow ExcnState(h, a)}$$

$$\textsc{D-Refl}\ \frac{}{\vdash_M l, s \Uparrow s} \qquad \textsc{D-Trans}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Uparrow s''}{\vdash_M l, s \Uparrow s''}$$

$$\textsc{D-Invs}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Uparrow s}{\vdash_M l, (O@O', S, h) \Uparrow s}$$

$$\textsc{D-Uncaught}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \quad getClass(h, a) = e \quad Handler(M, l, e) = \emptyset}{\vdash_M l, s \Uparrow ([a], \emptyset, h)}$$

Fig. 2. Bicolano: selected judgements and operational rules

computations. These *strong invariants* relate the state valid at the subject label to each future state in the current method invocation. This interpretation includes states in subframes, i.e. in method invocations that are triggered in the phrase represented by the subject label.

Infinite computations are also covered by the interpretation of local annotations in JML, i.e. assertions occurring at arbitrary program points which are to be satisfied whenever the program point is visited. The logic distinguishes these explicitly given annotation from strong invariants as the former ones are not necessarily present at all program points. A further specification idiom of JML that has a direct impact on the form of assertions is `\old` which refers to the initial state of a method invocation and may appear in post-conditions, local annotations, and strong invariants.

Formulae that are shared between postconditions, local annotations, and strong invariant, and additionally only concern the relationship between the subject state and the initial state of the method may be captured in pre-conditions.

Thus, the judgement of the logic are of the form $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ where M, l denotes a program point (composed of a method identifier and an instruc-

tion label), and the assertions forms are as follows, where $\mathcal{B}$ denotes the set of booleans.

Assertions $A \in \mathit{Assn} = \Sigma_0 \times \Sigma \to \mathcal{B}$ occur as preconditions A and local annotations Q, and relate the current state to the initial state of the current frame.

Postconditions $B \in \mathit{Post} = \Sigma_0 \times \Sigma \times \mathcal{T} \to \mathcal{B}$ relate the current state to the initial and final state of a (terminating) execution of the current frame.

Invariants $I \in \mathit{Inv} = \Sigma_0 \times \Sigma \times \Sigma \to \mathcal{B}$ relate the initial state of the current method, the current state, and any future state of the current frame or a subframe of it.

The component $\mathcal{G}$ of a judgement represents a proof context and is represented as an association of specification triples $(A, B, I) \in \mathit{Assn} \times \mathit{Post} \times \mathit{Inv}$ to program points.

The behaviour of methods is described using three assertion forms.

Method preconditions $R \in \mathit{MethPre} = \Sigma_0 \to \mathcal{B}$ are interpreted hypothetically, i.e. their satisfaction implies that of the method postconditions and invariants but is not directly enforced to hold at all invocation points.

Method postconditions $T \in \mathit{MethSpec} = \Sigma_0 \times \mathcal{T} \to \mathcal{B}$ constrain the behaviour of terminating method executions and thus relate only initial and final states.

Method invariants $\Phi \in \mathit{MethInv} = \Sigma_0 \times \Sigma \to \mathcal{B}$ constrain the behaviour of terminating and non-terminating method executions by relating the initial state of a method frame to any state that occurs during its execution.

A program specification is given by a method specification table $\mathcal{M}$ that associates to each method a method specification $\mathcal{S} = (R, T, \Phi)$, a proof context $\mathcal{G}$, and a table $\mathcal{Q}$ of local annotations $Q \in \mathit{Assn}$. From now on, let $\mathcal{M}$ denote some arbitrary but fixed specification table satisfying $\mathit{dom}\ \mathcal{M} = \mathit{dom}\ P$.

3.2 Assertion transformers

In order to notationally simplify the presentation of the proof rules, we define operators that relate assertions occurring in judgements of adjacent instructions. The following operators apply to the non-exceptional single-step execution of basic instructions.

$$\begin{aligned}
\mathsf{Pre}(M, l, l', A)(s_0, r) &= \exists\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \wedge A(s_0, s) \\
\mathsf{Post}(M, l, l', B)(s_0, r, t) &= \forall\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \to B(s_0, s, t) \\
\mathsf{Inv}(M, l, l', I)(s_0, r, t) &= \forall\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \to I(s_0, s, t)
\end{aligned}$$

These operators resemble WP-operators, but are separately defined for pre-conditions, post-conditions, and invariants.

Exceptional behaviour of basic instructions is captured by the operators

$$\begin{aligned}
\mathsf{Pre}^{\mathsf{excn}}(M,l,e,A)(s_0,r) &= \exists\ s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \wedge \mathit{getClass}(h,a) = e \wedge \\
&\qquad r = ([a],\mathit{lv}(s),h) \wedge A(s_0,s) \\
\mathsf{Post}^{\mathsf{excn}}(M,l,e,B)(s_0,r,t) &= \forall\ s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow \mathit{getClass}(h,a) = e \rightarrow \\
&\qquad r = ([a],\mathit{lv}(s),h) \rightarrow B(s_0,s,t) \\
\mathsf{Inv}^{\mathsf{excn}}(M,l,e,I)(s_0,r,t) &= \forall\ s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow \mathit{getClass}(h,a) = e\ \rightarrow \\
&\qquad r = ([a],\mathit{lv}(s),h) \rightarrow I(s_0,s,t)
\end{aligned}$$

In the case of method invocations, we replace the reference to the operational judgement by a reference to the method specifications, and include the construction and destruction of a frame. For example, the operators for non-exceptional execution of static methods are

$$\begin{aligned}
&\mathsf{Pre}_{\mathsf{sinv}}(R,T,A,[x_1,\ldots,x_n])(s_0,s) = \\
&\quad \exists\ O\ S\ h\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n,h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n,h),(k,v))) \wedge \\
&\qquad\qquad s = (v :: O,S,k) \wedge A(s_0,([v_1,\ldots,v_n]@O,S,h)) \\
&\mathsf{Post}_{\mathsf{sinv}}(R,T,B,[x_1,\ldots,x_n])(s_0,r,t) = \\
&\quad \forall\ O\ S\ k\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n,h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n,h),(k,v)))\ \rightarrow \\
&\qquad\qquad r = (v :: O,S,k) \rightarrow B(s_0,([v_1,\ldots,v_n]@O,S,h),t) \\
&\mathsf{Inv}_{\mathsf{sinv}}(R,T,I,[x_1,\ldots,x_n])(s_0,s,r) = \\
&\quad \forall\ O\ S\ k\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n,h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n,h),(k,v)))\ \rightarrow \\
&\qquad\qquad s = (v :: O,S,k) \rightarrow I(s_0,([v_1,\ldots,v_n]@O,S,h),r)
\end{aligned}$$

The exceptional operators for static methods cover exceptions that are raised during the execution of the invoked method but not handled locally. Due to space limitations we omit the operators for exceptional (null-pointer exceptions w.r.t. the invoking object) and non-exceptional behaviour of virtual methods.

3.3 Selected proof rules

An addition to influencing the types of assertions, type systems also motivate the use of a certain form of judgements and proof rules. Indeed, one of the advantages of type systems is their compositionality i.e. the fact that statements regarding a program phrase are composed from the statements referring to the constituent phrases, as in the following typical proof rule for a language of expressions

$$\frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 + e_2 : \mathbf{int}}.$$

Transferring this scheme to bytecode leads to a rule format where hypothetical judgements refer to the control flow successors of the phrase in the judgement's conclusion. In addition to supporting syntax-directed reasoning, this orientation renders the explicit construction of a control flow graph unnecessary, as no control flow predecessor information is required to perform a proof.

Figure 3 presents selected proof rules. These are motivated as follows.

$$\textsc{Instr}\ \frac{\begin{array}{c} basic(M,l) \qquad SC_1 \qquad SC_2 \qquad l'' = suc_M(l) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l'',A)\}\, M,l'\, \{\mathsf{Post}(M,l,l'',B)\}\, (\mathsf{Inv}(M,l,l'',I)) \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \\ \mathcal{G} \vdash \{\mathsf{Pre}^{\mathsf{excn}}(M,l,e,A)\}\, M,l'\, \{\mathsf{Post}^{\mathsf{excn}}(M,l,e,B)\}\, (\mathsf{Inv}^{\mathsf{excn}}(M,l,e,I)) \\ \forall\, s_0\ s\ h\ a.\ (\forall\ e.\ getClass(h,a) = e\ \rightarrow Handler(M,l,e) = \emptyset)\ \rightarrow \\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow A(s_0,s) \rightarrow B(s_0,s,(h,a)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{Goto}\ \frac{\begin{array}{c} M(l) = \mathsf{Goto}\ l' \qquad SC_1 \qquad SC_2 \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l',A)\}\, M,l'\, \{\mathsf{Post}(M,l,l',B)\}\, (\mathsf{Inv}(M,l,l',I)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{If0}\ \frac{\begin{array}{c} M(l) = \mathsf{ifz}\ l' \qquad SC_1 \qquad SC_2 \qquad l'' = suc_M(l) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l',A)\}\, M,l'\, \{\mathsf{Post}(M,l,l',B)\}\, (\mathsf{Inv}(M,l,l',I)) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l'',A)\}\, M,suc_M(l)\, \{\mathsf{Post}(M,l,l'',B)\}\, (\mathsf{Inv}(M,l,l'',I)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{InvS}\ \frac{\begin{array}{c} M(l) = \mathsf{invokestatic}\ M' \quad \mathcal{M}(M') = (R,T,\Phi) \quad SC_1 \quad SC_2 \\ \forall\, s_0\ O\ S\ h\ O'\ r\ v_i.\ (R(par_{M'} \mapsto O,h) \rightarrow \Phi((par_{M'} \mapsto O,h),r))\ \rightarrow \\ A(s_0,(O@O',S,h)) \rightarrow I(s_0,(O@O',S,h),r) \\ A_1 = \mathsf{Pre}_{\mathsf{sinv}}(R,T,A,par_{M'}) \qquad B_1 = \mathsf{Post}_{\mathsf{sinv}}(R,T,B,par_{M'}) \\ \mathcal{G} \vdash \{A_1\}\, M,suc_M(l)\, \{B_1\}\, (\mathsf{Inv}_{\mathsf{sinv}}(R,T,I,par_{M'})) \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \\ \mathcal{G} \vdash \{\mathsf{Pre}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,A,e,par_{M'})\}\, M,l'\, \{\mathsf{Post}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,B,e,par_{M'})\} \\ (\mathsf{Inv}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,I,e,par_{M'})) \\ \forall\, s_0\ O\ S\ h\ O'\ k\ a.\ (R(par_{M'} \mapsto O,h) \rightarrow \Phi((par_{M'} \mapsto O,h),(k,a)))\ \rightarrow \\ (\forall\ e.\ getClass(k,a) = e\ \rightarrow Handler(M,l,e) = \emptyset)\ \rightarrow \\ A(s_0,(O@O',S,h)) \rightarrow B(s_0,(O@O',S,h),(k,a)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{Ret}\ \frac{\begin{array}{c} M(l) = \mathsf{vreturn} \qquad SC_1 \qquad SC_2 \\ \forall\, s_0\ v\ O\ S\ h.\ A(s_0,(v :: O,S,h)) \rightarrow B(s_0,(v :: O,S,h),(h,v)) \end{array}}{\mathcal{G} \vdash \{A\}\, M,l\, \{B\}\, (I)}$$

$$\textsc{Conseq}\ \frac{\begin{array}{cc} \mathcal{G} \vdash \{A'\}\, \ell\, \{B'\}\, (I') & \forall\, s_0\ s.\ A(s_0,s) \rightarrow A'(s_0,s) \\ \forall\, s_0\ s\ t.\ B'(s_0,s,t) \rightarrow B(s_0,s,t) & \forall\, s_0\ s\ r.\ I'(s_0,s,r) \rightarrow I(s_0,s,r) \end{array}}{\mathcal{G} \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

$$\textsc{Ax}\ \frac{\begin{array}{c} \mathcal{G}(\ell) = (A,B,I) \qquad \forall\, s_0\ s.\ A(s_0,s) \rightarrow I(s_0,s,s) \\ \forall Q.\ \mathcal{Q}(\ell) = Q \rightarrow (\forall\, s_0\ s.\ A(s_0,s) \rightarrow Q(s_0,s)) \end{array}}{\mathcal{G} \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

Fig. 3. Program logic: selected syntax-directed rules

Rule INSTR describes the behaviour of basic instructions. The hypothetical judgement for the successor instruction involves assertions that are related to the assertions in the conclusion by the transformers for normal termination. A further hypothesis captures exceptions that are handled locally, i.e. those exceptions e to which the exception handler of the current method associates a handling instruction (predicate $\mathit{Handler}(M, l, e) = l'$). Exceptions that are not handled locally result in abrupt termination of the method. Consequently, these exceptions are modelled in a side condition that involves the method postcondition rather than a further judgemental hypothesis.

Finally, the side conditions SC_1 and SC_2 ensure that the invariant I and the local annotation Q (if existing) are satisfied in any state reaching label l.

$$SC_1 = \forall\, s_0\ s.\ A(s_0, s) \to I(s_0, s, s)$$
$$SC_2 = \forall\, Q.\ \mathcal{Q}(M, l) = Q \to (\forall\, s_0\, s.\ A(s_0, s) \to Q(s_0, s))$$

In particular, SC_2 requires us to prove any annotation that is associated with label l. Satisfaction of I in later states, and satisfaction of local annotations Q' of later program points are guaranteed by the judgement for $\mathit{suc}_M(l)$.

The rules for conditional and unconditional jumps include a hypotheses for the control flow successors, and the same side conditions for local annotations and invariants as rule INSTR. No further hypotheses or side conditions regarding exceptional behaviour are required as these instructions do not raise exceptions. These rules also account for the verification of loops which on the level of bytecode are rendered as jumps. Loop invariants can be inserted as postconditions B at their program point. Rule AX allows one to use such invariants whereas according to Definition 1 they must be established once in order for a verification to be valid.

In rule INVS, the invariant of the callee, namely Φ (more precisely: the satisfaction of Φ whenever the initial state of the callee satisfies the precondition R), and the local precondition A may be exploited to establish the invariant I. This ensures that I will be satisfied by all states that arise during the execution of M', as these states will always conform to Φ. The callee's post-condition T is used to construct the assertions that occur in the judgement for the successor instruction l'. Both conditions reflect the transfer of the method arguments and return values between the caller and the callee. This protocol is repeated in the hypothesis and the side condition for the exceptional cases which otherwise follow the pattern mentioned in the description of the rule INSTR.

A similar rule for virtual methods is omitted. The rule for method returns, RET, ties the precondition A to the post-condition B w.r.t. the terminal state that is constructed using the topmost value of the operand stack.

Finally, the *logical rules* CONSEQ and AX arise from the standard rules by adding suitable side conditions for strong invariants and local assertions.

3.4 Behavioural subtyping and verified programs

We say that method specification (R, T, Φ) *implies* (R', T', Φ') if

– for all s_0 and t, $R(s_0) \rightarrow T(s_0, t)$ implies $R'(s_0) \rightarrow T'(s_0, t)$, and
– for all s_0 and s, $R(s_0) \rightarrow \Phi(s_0, s)$ implies $R'(s_0) \rightarrow \Phi'(s_0, s)$

Furthermore, we say that $\mathcal{M}$ satisfies *behavioural subtyping* for P if whenever P contains an instruction invokevirtual M' with $\mathcal{M}(M') = (\mathcal{S}', \mathcal{G}', \mathcal{Q}')$, and M overrides M', then there are $\mathcal{S}$, $\mathcal{G}$ and $\mathcal{Q}$ with $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$ such that $\mathcal{S}$ implies $\mathcal{S}'$. Finally, we call a derivation $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ *progressive* if it contains at least one application of a non-logical rule.

Definition 1. *P is* verified *with respect to $\mathcal{M}$, notation $\mathcal{M} \vdash P$, if*

– *$\mathcal{M}$ satisfies behavioural subtyping for P, and*
– *for all M, $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$, and $\mathcal{S} = (R, T, \Phi)$*
  - *a progressive derivation $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ exists for any l, A, B, and I with $\mathcal{G}(M, l) = (A, B, I)$, and*
  - *a progressive derivation $\mathcal{G} \vdash \{A\}\, M, init_M\, \{B\}\, (I)$ exists for*

$$A(s_0, s) \equiv s = state(s_0) \wedge R(s_0)$$
$$B(s_0, s, t) \equiv s = state(s_0) \rightarrow T(s_0, t)$$
$$I(s_0, s, r) \equiv s = state(s_0) \rightarrow \Phi(s_0, r).$$

As the reader may have noticed, behavioural subtyping only affects method specifications but not the proof contexts $\mathcal{G}$ or annotation tables $\mathcal{Q}$. Technically, the reason for this is that no constraints on these components are required in order to prove the logic sound. Pragmatically, we argue that proof contexts and local annotations tables of overriding methods indeed should not be related to contexts and annotation tables of their overridden counterparts, as both kinds of tables expose the internal structure of method implementations. In particular, entries in proof contexts and annotation tables are formulated w.r.t. specific program points, which would be difficult to interprete outside the method boundary or indeed across different (overriding) implementations of a method.

The distinction between progressive and non-progressive derivations prevents attempts to justify a proof context or method specification table simply by applying the axiom rule to all entries. In program logics for high-level languages, the corresponding effect is silently achieved by the unfolding of the method body in the rule for method invocations [29]. As our judgemental form does not permit such an unfolding, the auxiliary notion of progressive derivations is introduced. In our formalisation, the separation between progressive and other derivations is achieved by the introduction of a second judgement form, as described in [8].

3.5 Interpretation and soundness

Definition 2. *The triple $(\mathcal{Q}, B, I)$ is* valid at (M, l) for (s_0, s) *if*

– *for all r, if $\vdash_M l, s \Downarrow t$ then $B(s_0, s, t)$*
– *for all l' and r, if $\vdash_M l, s \Rightarrow^* l', r$ and $\mathcal{Q}(l') = Q$, then $Q(s_0, r)$, and*
– *for all r, if $\vdash_M l, s \Uparrow r$ then $I(s_0, s, r)$.*

Note that the second clause applies to annotations Q associated with arbitrary labels l' in method M that will be visited during the execution of M from (l, s) onwards. Although these annotations are interpreted without recourse to the state s, the proof of $Q(s_0, r)$ may exploit the precondition $A(s_0, s)$.

The soundness result is then as follows.

Theorem 1. *For $\mathcal{M} \vdash P$ let $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$, $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ be a progressive derivation, and $A(s_0, s)$. Then $(\mathcal{Q}, B, I)$ is valid at (M, l) for (s_0, s).*

In particular, this theorem implies that for $\mathcal{M} \vdash P$ all method specifications in $\mathcal{M}$ are honoured by their method implementations. The proof of this result may be performed in two ways. Following the approach of Kleymann and Nipkow [25, 29, 3], one would first prove that the derivability of a judgement entails its validity, under the hypothesis that contextual judgements have already been validated. For this task, the standard technique involves the introduction of relativised notions of validity that restrict the interpretation of judgements to operational judgements of bounded height. Then, the hypothesis on contextual judgements is eliminated using structural properties of the relativised validity. An alternative to this approach has been developed by Benjamin Gregoire in the course of the formalisation of the present logic. It consists of (i) defining a family of syntax-directed judgements (one judgement form for each instruction form, inlining the rule of consequence), (ii) proving that property $\mathcal{M} \vdash P$ implies that the last step in a derivation of $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ can be replaced by an application of the syntax-directed judgement corresponding to the instruction at M, l (in particular, an application of the axiom rule is replaced by the derivation for the corresponding code blocks from $\mathcal{G}$), and (iii) proving the main claim of Theorem 1 by treating the three parts of Definition 2 separately, each one by induction over the respective operational judgement.

4 Type-based verification

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

The requirement imposed on programs is similar to that of the analysis presented by Cachera et al. in [13] in that recursive program structures are denied the facility to allocate memory. However, our analysis is presented as a type system while the analysis presented in [13] is phrased as an abstract interpretation. In addition, Cachera et al.'s approach involves the formalisation of the calculation of the program representation (control flow graph) and of the inference algorithm (fixed point iteration) in the theorem prover. In contrast, our presentation separates the algorithmic issues (type inference and checking) from semantic issues (the property expressed or guaranteed) as is typical for a type-based formulation. Depending on the verification infrastructure available at the code consumer side, the PCC certificate may either consist of (a digest of) the

typing derivation or an expansion of the interpretation of the typing judgements into the MOBIUS logic. The latter approach was employed in our earlier work [10] and consists of understanding typing judgements as derived proof rules in the program logic and using syntax-directed proof tactics to apply the rules in an automatic fashion. In contrast to [10], however, the interpretation given in the present section extends to non-terminating computations, albeit for a far simpler type system.

The present section extends the work presented in [8] as the type system is now phrased for bytecode rather than an intermediate functional language and includes the treatment of exceptions and virtual methods.

Bytecode-level type system The type system consists of judgements of the form $\vdash_{\Sigma,\Lambda} \ell : n$, expressing that the segment of bytecode whose initial instruction is located at ℓ is guaranteed not to allocate more than n memory cells. Here, ℓ denotes a program point M, l while signatures Σ and Λ assign types (natural numbers n) to identifiers of methods and bytecode instructions (in particular, when those are part of a loop), respectively.

$$\text{C-New}\ \frac{n \geq 1 \qquad M(l) = \mathsf{New}\ C \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n-1}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-Instr}\ \frac{\begin{array}{c} n \geq 1 \qquad basic(M,l) \qquad \neg M(l) = \mathsf{New}\ C \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \vdash_{\Sigma,\Lambda} M, l' : n-1 \end{array}}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-If}\ \frac{n \geq 0 \qquad M(l) = \mathsf{ifz}\ l' \qquad \vdash_{\Sigma,\Lambda} M, l' : n \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-Invoke}\ \frac{\begin{array}{c} M(l) \in \{\mathsf{invokestatic}\ M', \mathsf{invokevirtual}\ M'\} \qquad \Sigma(M') = k \\ n \geq 1 \qquad k \geq 0 \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \vdash_{\Sigma,\Lambda} M, l' : n-1 \end{array}}{\vdash_{\Sigma,\Lambda} M, l : n+k}$$

$$\text{C-Ret}\ \frac{M(l) = \mathsf{vreturn}}{\vdash_{\Sigma,\Lambda} M, l : 0} \qquad \text{C-Sub}\ \frac{\vdash_{\Sigma,\Lambda} \ell : n \qquad n \leq k}{\vdash_{\Sigma,\Lambda} \ell : k} \qquad \text{C-Assum}\ \frac{\Lambda(\ell) = n}{\vdash_{\Sigma,\Lambda} \ell : n}$$

Fig. 4. Type system for constant heap space

The rules are presented in Figure 4. The first rule, C-New, asserts that the memory consumption of a code fragment whose first instruction is $\mathsf{new}\ C$ is the increment of the remaining code. Rule C-Instr applies to all basic instructions (in the case of $\mathsf{goto}\ l'$ we take $suc_M(l)$ to be l'), except for $\mathsf{new}\ C$ – the predicate $basic(m, l)$ is defined as in Section 3.3. The memory effect of these instructions is zero, as is the case for return instructions, conditionals, and (static) method

invocations in the case of normal termination. For exceptional termination, the allocation of a fresh exception object is accounted for by decrementing the type for the code continuation by one unit. The rule C-Assum allows for using the annotation attached to the instruction if it matches the type of the instruction.

A typing derivation $\vdash_{\Sigma,\Lambda} \ell : k$ is called *progressive* if it does not solely contain applications of rules C-Sub and C-Assum. Furthermore, we call P *well-typed* for Σ, notation $\vdash_\Sigma P$, if for all M and n with $\Sigma(M) = n$ there is a local specification table Λ such that a progressive derivation $\vdash_{\Sigma,\Lambda} M, \mathit{init}_M : n$ exists, and for all ℓ with $\Lambda(\ell) = k$ we have a progressive derivation $\vdash_{\Sigma,\Lambda} \ell : k$.

Type checking and inference The tasks of checking and automatically finding (inference) of typing derivations are not our main concern here. Nevertheless, we discuss briefly how this can be achieved.

For this simple type system checking a given typing derivation amounts to verifying the inequations that arise as side conditions. Furthermore, given Σ, Λ a corresponding typing derivation can be reconstructed by applying the typing rules in a syntax-directed fashion. In order to construct Σ, Λ as well (type inference) one writes down a "skeleton derivation" with indeterminates instead of actual numeric values and then solves the arising system of linear inequalities. Alternatively, one can proceed by counting allocation statements along paths and loops in the control-flow graph.

Our main interest here is, however, the use of existing type derivations however obtained in order to mechanically construct proofs in the program logic. This will be described now.

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

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

Here, $|h|$ denotes the size of heap h and $\mathit{heap}(s)$ extracts the heap component of a state. We specialise the main judgement form of the bytecode logic to

$$\mathcal{G} \vdash \ell\ \{n\} \equiv \mathit{let}\ (A, B, I) = [\![n]\!]\ \mathit{in}\ \mathcal{G} \vdash \{A\}\,\ell\,\{B\}\,(I).$$

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

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

and for a given Λ we define $\mathcal{G}_\Lambda$ pointwise by $\mathcal{G}_\Lambda(\ell) = [\![\Lambda(\ell)]\!]$.

Finally, we say that $\mathcal{M}$ satisfies Σ, notation $\mathcal{M} \models \Sigma$, if for all methods M, $\mathcal{M}(M) = (\mathit{Spec}\ n, \mathcal{G}_\Lambda, \emptyset)$ holds precisely if $\Sigma(M) = n$, where Λ is the context associated with M in $\vdash_\Sigma P$. Thus, method specification table $\mathcal{M}$ contains for each method the precondition, postcondition and invariant from Σ, the (complete) context determined from Λ, and the empty local annotation table $\mathcal{Q}$.

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

Proposition 1. *For $\mathcal{M} \models \Sigma$ let M be provided in $\mathcal{M}$ with some annotation table Λ such that $\vdash_{\Sigma,\Lambda} M, l : n$ is progressive. Then $\mathcal{G}_\Lambda \vdash M, l\ \{n\}$.*

From this, one may obtain the following, showing that well-typed programs satisfy the verified-program property:

Theorem 2. *Let $\mathcal{M} \models \Sigma$ and $\vdash_\Sigma P$, and let $\mathcal{M}$ satisfy behavioural subtyping for P. Then $\mathcal{M} \vdash P$.*

Discussion In order to improve the precision of the analysis, a possibility is to combine the type system with a null-pointer analysis. For this, we would specialise the proof rules for instructions which might throw a null-pointer exception. At program points for which the analysis guarantees absence of such exceptions, we may then use a specialised typing rule. For example, a suitable rule for the field access operation is the following.

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

Program points for which the analysis is unable to discharge the side condition $\mathit{refNotNull}(m,l)$ would be dealt with using the standard rule. Similarly, instructions that are guaranteed not to throw runtime exceptions (like load x, store x, dup) may be typed using the optimised rule

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

We expect that justifying these specialised rules using the program logic would not pose major problems, while the formal integration with other program analyses (such as the null-pointer analysis) is a topic for future research.

5 Resource-extended program logic

In this section we give a brief overview of an extension of the MOBIUS base logic as described in Section 3 for dealing with resources in a generic way. The extension addresses the following shortcoming of the basic logic:

Resource consumption Specific resources that we would like to reason about include instruction counters, heap allocation, and frame stack height. A well-known technique for modelling these resources is *code instrumentation*, i.e. the introduction of (real or ghost) variables and instructions manipulating these. However, code instrumentation appears inappropriate for a PCC environment, as it does not provide an end-to-end guarantee that can be understood without reference to the program at hand. In particular, the overall satisfaction of a resource property using code instrumentation requires an analysis of the annotated program, i.e. a proof that the instrumentation variables are introduced and manipulated correctly. Furthermore, the interaction between additional variables of different domains, and between auxiliary variables and proper program variables is difficult to reason about.

Execution traces Here, the goal is to reason about properties concerning a full terminating or non-terminating execution of a program, for example by imposing that an execution satisfies a formula expressed in temporal logics or a policy given in terms of a security automaton. Such specifications may concern the entire execution history, i.e. be defined over a sequence of (intermediate) Bicolano states, and are thus not expressible in the MOBIUS base logic.

Ghost variables are heavily used in JML, both for resource-accounting purposes as well as functional specifications, but are not directly expressible in the base logic.

In this section we extend the base logic by a generic resource-accounting mechanism that may be instantiated to the above tasks. In addition to the work reported here, we have also performed an analysis of the usage made of ghost variables in JML, and have developed interpretations of ghost variables in native and resource-extended program logics [24]. In particular, loc.cit. contains a formalised proof demonstrating how resource counting using ghost variables in native logics may be effectively eliminated, by translating each proof derivation into a derivation in the resource-extended logic.

5.1 Semantic modelling of generic resources

In order to avoid the pitfalls of code instrumentation discussed above, a semantic modelling of resource consumption was chosen. The logic is defined over an extended operational semantics, the judgements of which are formulated over the same components as the standard Bicolano operational semantics, plus a further resource-accounting component [20]. The additional component is of the a priori unspecified type ACT, and occurs as a further component in initial, final, and intermediate states. In addition, we introduce transfer functions that update the content of this component according to the other state components, including the program counter. The operational semantics of the extended framework is then obtained by embedding each non-extended judgement form in a judgement form over extended states and invoking the appropriate transfer functions on the resource component. While these definitions of the operational semantics

are carried out once and for all, the implementation of the transfer functions themselves is programmable. Thus, realisations of the framework for particular resources may be obtained by instantiating the ACT to some specific type and implementing the transfer functions as appropriate. The program logic remains conceptually untouched, i.e. it is structurally defined as the logic from Section 3, but the definitions of assertion transformers and rules, and the soundness proof, are adapted to extended states and modified operational judgements.

In comparison to admitting the definition of ad-hoc extensions to the program logic, we argue that the chosen approach is better suited to the PCC applications, as the consumer has a single point of reference where to specify his policy, namely the implementation of the transfer functions.

5.2 Application: block-booking

As an application of the resource-extended program logic, we consider a scenario where an application repeatedly sends some data across a network provided that each such operation is sanctioned by an interaction with the user. In order to avoid authorisation requests for individual send operations, a high-level language might contain a primitive **auth**(n) that asks the user to authorise n messages in one interaction. A reasonable resource policy for the code consumer then is to require that no send operation be carried out without authorisation, and that at each point of the execution, the acquired authorisations suffice for servicing the remaining **send** operations. (For simplicity, we assume that refusal by the user to sanction an authorisation request simply blocks or leads to immediate non-termination without any observable effect.)

We note that as in the case of the logic loop constructs from the high-level language are mapped to conditional and unconditional jumps that must be typed using the corresponding rules.

We now outline a bytecode-level type and effect system for this task, for a sublanguage of scalar (integer) values and unary static methods. Effects τ are rely-guarantee pairs (m, n) of natural numbers: a code fragment with this effect satisfies the above policy whenever executed in a state with at least m unused authorisations, with at least n unused authorisations being left over upon termination. The number of authorisations that are additionally acquired, and possibly used, during the execution are unconstrained. Types $C, D, \ldots$ are sets of integers constraining the values stored in variables or operand stack positions. Judgements take the form $\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} \ell : C, \tau$, with the following components:

- the abstract store Δ maps local variables to types
- the abstract operand stack η is represented as a list of types
- Ξ is an equivalence relation relation ranging over identifiers ρ from $dom\ \Delta \cup dom\ \eta$ where $dom\ \eta$ is taken to be the set $\{0, \ldots, |\eta| - 1\}$. The role of Ξ is to capture equalities between values on the operand stack and the store.
- instruction labels $\ell = (M, l)$ indicate the current program point, as before
- the type C describes the return type
- the effect τ captures the pre-post-behaviour of the subject phrase with respect to authorisation and send events

– the proof context Λ associates sets of tuples $(\Delta, \eta, \Xi, C, \tau)$ to labels l (implicitly understood with respect to method M).
– the method signature table Σ maps method names to type signatures of the form $\forall i{\in}I.\ C_i \xrightarrow{(m_i,n_i)} D_i$. Limiting our attention to static methods with a single parameter, such a poly-variant signature indicates that for each i in some (unspecified) index set I, the method is of type $C_i \xrightarrow{(m_i,n_i)} D_i$, i.e. takes arguments satisfying constraint C_i to return values satisfying D_i with (latent) effect (m_i, n_i).

In addition to ignoring virtual methods (and consequently avoiding the need for a condition enforcing behavioural subtyping of method specifications), we also ignore exceptions. Finally, while our example program contains simple objects we do not give proof rules for object construction or field access. We argue that this impoverished fragment of the JVML suffices for demonstrating the concept of certificate generation for effects, and leave an extension to larger language fragments as future work.

For an arbitrary relation R, we let $Eq(R)$ denote its reflexive, transitive and symmetric closure. We also define the operations $\Xi - \rho$, $\Xi + \rho$ and $\Xi[\rho := \rho']$ on equivalence relation Ξ and identifiers ρ and ρ', as follows.

$$\begin{aligned} \Xi - \rho &\equiv \Xi \setminus \{(\rho_1, \rho_2) \mid \rho = \rho_1 \vee \rho = \rho_2\} \\ \Xi + \rho &\equiv \Xi \cup \{(\rho, \rho)\} \\ \Xi[\rho := \rho'] &\equiv Eq((\Xi - \rho) \cup \{(\rho, \rho')\}) \end{aligned}$$

The interpretation of position ρ in a pair (O, S) is given by $[\![x]\!]_{(O,S)} = S(x)$ and $[\![n]\!]_{(O,S)} = O(n)$. The interpretation of a triple Δ, η, Ξ in a pair (O, S) is given by the formula

$$[\![\Delta, \eta, \Xi]\!]_{(O,S)} = \begin{cases} dom\ \Delta \subseteq dom\ S \wedge |\eta| = |O| \wedge \\ \forall x \in dom\ \Delta.\ S(x) \in \Delta(x) \wedge \\ \forall i < |\eta|.\ O(i) \in \eta(i) \wedge \\ \forall (\rho, \rho') \in \Xi.\ [\![\rho]\!]_{(O,S)} = [\![\rho']\!]_{(O,S)} \end{cases}$$

With the help of these operations, the type system is now defined by the rules given in Figure 5. Due to the formulation at the bytecode level, the authorisation primitive does not have a parameter but obtains its argument from the operand stack.

The rule for conditionals, E-If, exploits the outcome of the branch condition by updating the types of all variables associated with the top operand stack position in Ξ. This limited form of copy propagation will be made use of in the verification of an example program below.

In the rule of consequence, E-Sub, subtyping on types is denoted by $C <: D$ and given by subset inclusion, and is extended to abstract stores (notation $\Delta <: \Delta'$) and abstract operand stacks (notation $\eta <: \eta'$) in a pointwise fashion. Sub-effecting is given by the reflexive closure of the rule

$$\frac{k \geq m + d \quad l \leq n + d}{(m, n) <: (k, l)}.$$

$$\text{E-Send}\ \frac{M(l) = \mathbf{send} \qquad \Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m-1, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Auth}\ \frac{\begin{array}{c} M(l) = \mathbf{auth} \qquad \forall i \in C.\ i \geq k \\ \Delta, \eta, \Xi - |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m+k, n) \end{array}}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Goto}\ \frac{M(l) = \mathsf{goto}\ \ l' \qquad \Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l' : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-If}\ \frac{\begin{array}{c} M(l) = \mathsf{ifz}\ \ l' \qquad \Xi' = \Xi - |\eta| \\ \Delta_1 = \Delta[x \mapsto \Delta(x) \cap (\mathbf{Z} \setminus \{0\})]_{(|\eta|, x) \in \Xi} \\ \eta_1 = \eta[i \mapsto \eta(i) \cap (\mathbf{Z} \setminus \{0\})]_{(|\eta|, i) \in \Xi\ \wedge\ 0 \leq i < |\eta|} \\ \Delta_2 = \Delta[x \mapsto \Delta(x) \cap \{0\}]_{(|\eta|, x) \in \Xi} \\ \eta_2 = \eta[i \mapsto \eta(i) \cap \{0\}]_{(|\eta|, i) \in \Xi\ \wedge\ 0 \leq i < |\eta|} \\ \Delta_1, \eta_1, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : (m, n) \qquad \Delta_2, \eta_2, \Xi' \vdash_{\Sigma,\Lambda} M, l' : D, (m, n) \end{array}}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Store}\ \frac{\begin{array}{c} M(l) = \mathsf{store}\ \ x \qquad \Xi' = (\Xi[x := |\eta|]) - |\eta| \\ \Delta[x \mapsto C], \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n) \end{array}}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Load}\ \frac{\begin{array}{c} M(l) = \mathsf{load}\ \ x \qquad \Xi' = \Xi[|\eta| := x] \\ \Delta, \Delta(x) :: \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n) \end{array}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Push}\ \frac{M(l) = \mathsf{push}\ \ c \quad \Delta, \{c\} :: \eta, \Xi + |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Binop}\ \frac{\begin{array}{c} M(l) = \mathsf{binop}\ \oplus \qquad C = \{z | z = x\ \ \oplus\ \ y, x \in C_1, y \in C_2\} \\ \Delta, C :: \eta, ((\Xi - |\eta|) - (|\eta| + 1)) + |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n) \end{array}}{\Delta, C_1 :: C_2 :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-InvS}\ \frac{\begin{array}{c} M(l) = \mathsf{invokestatic}\ M' \qquad \Sigma(M') = \forall i \in I.\ C_i \xrightarrow{\tau_i} D_i \qquad k \in I \\ \Xi' = (\Xi - |\eta|) + |\eta| \qquad \Delta, D_k :: \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (n_k, n) \end{array}}{\Delta, C_k :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m_k, n)}$$

$$\text{E-Vret}\ \frac{M(l) = \mathsf{vreturn}}{\Delta, D, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (0, 0)} \qquad \text{E-Ax}\ \frac{(\Delta, \eta, \Xi, D, \tau) \in \Lambda(l)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, \tau}$$

$$\text{E-Sub}\ \frac{\begin{array}{c} \Delta', \eta', \Xi' \vdash_{\Sigma,\Lambda} \ell : C, \tau' \\ \Delta <: \Delta' \qquad \eta <: \eta' \\ C <: D \quad \tau' <: \tau \quad \Xi' \subseteq \Xi \end{array}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} \ell : D, \tau} \qquad \text{E-Univ}\ \frac{\forall\ O\ S.\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} = \mathit{False}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

Fig. 5. Type and effect system for block-booking

The final rule, E-Univ, allows us to associate an arbitrary effect and result type to a code segment under the condition that the constraints Δ, η, Ξ on the initial state are unsatisfiable. The main use of this rule is in cases where branch conditions render one branch dead code.

In order to prove the soundness of the type system in the extended program logic, we instantiate the parameter ACT to the type of finite words over the set $\{\mathbf{send}\} \cup \{\mathbf{auth}(z) \mid z \geq 0\}$ and implement the transfer functions such that each execution of the primitives **send** and **auth** results in appending the appropriate action to the trace - in case of authorisation events, the number z is obtained by inspecting the topmost value of the operand stack.

We interpret a judgement $\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)$ as the logic statement

$$[\![\Lambda]\!]_M \vdash \{\lambda\, s_0.\ \mathit{True}\}\, M, l\, \{[\![(\Delta, \eta, \Xi, m, n, D)]\!]\}\, ([\![(\Delta, \eta, \Xi, m)]\!]),$$

with the following components. The postcondition $[\![(\Delta, \eta, \Xi, m, n, D)]\!]$ is

$$\lambda\, (s_0, (O, S, h, X), (h, v, Y)).\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} \rightarrow \\ (\exists Z.\ v \in D \wedge Y = XZ \wedge |Z|_{\mathsf{auth}} + m \geq |Z|_{\mathsf{send}} + n).$$

For any terminating execution starting in an initial store and operand stack conforming to the abstractions Δ and η, and respecting the equivalence relation Ξ, this property guarantees that the return value satisfies D. Furthermore, the sub-traces for authorisation and send events (obtained by projecting from the trace Z of all events encountered during the execution of the phrase) satisfy the inequality interpreting the effect.

A similar explanation holds for the definition of the invariant $[\![(\Delta, \eta, \Xi, m)]\!]$,

$$\lambda\, (s_0, (O, S, h, X), (O', S', h', X')).\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} \rightarrow \\ (\exists Z.\ X' = XZ \wedge |Z|_{\mathsf{auth}} + m \geq |Z|_{\mathsf{send}}).$$

The local proof context $[\![\Lambda]\!]_M$ is given by

$$[(M, l) \mapsto (\mathit{True}, [\![(\Delta, \eta, \Xi, m, n, D)]\!], [\![(\Delta, \eta, \Xi, m)]\!])]_{\Lambda(l)=(\Delta,\eta,\Xi,D,(m,n))},$$

i.e. by translating the entries of Λ pointwise. Finally, each specification entry $\Sigma(M) = \forall i \in I.\ C_i \xrightarrow{(m_i,n_i)} D_i$ results in an entry $\mathcal{M}(M) = (R, T, \Phi)$ in the bytecode logic specification table, where

$$\begin{aligned} R(s_0) &= \mathit{True} \\ T((S, h, X), (h, v, Y)) &= \forall i \in I.\, S(\mathrm{arg}) \in C_i \rightarrow \\ &\qquad (\exists\ Z.\ v \in D_i \wedge Y = XZ \wedge \\ &\qquad\quad |Z|_{\mathsf{auth}} + m_i \geq |Z|_{\mathsf{send}} + n_i) \\ \Phi((S, h, X), (O, S', h', X')) &= \forall i \in I.\, S(\mathrm{arg}) \in C_i \rightarrow \\ &\qquad (\exists\ Z.\ X' = XZ \wedge |Z|_{\mathsf{auth}} + m_i \geq |Z|_{\mathsf{send}}) \end{aligned}$$

where arg is the formal parameter. Based on this interpretation, certificate generation may now be obtained by deriving the typing rules from the program logic

and introducing appropriate notions of progressive derivations and well-typed programs (in the absence of virtual methods: without a behavioural subtyping condition), in a similar way as in Section 4. The formalisation of this is left as future research.

5.3 Example

We assume two builtin integer-valued functions `size_string` yielding the number of SMS messages required to send a given string, and `size_book` which gives the size of an address book. Figure 6 presents Java-style pseudocode for sending a given string to all addresses of a given address book after requiring the necessary permissions. The program first computes the total number of SMS messages

```
public interface Parameters {
  int p=...; //some constant >= 0
}
class BlockBooking {
  static void send () {...};
  static void auth (int p) {...};
  void block_send(Java.lang.String s, addrbook b) {
       int n = size_string(s);
       int m = size_book(b);
       int nb_sms = n * m;
       int j = 0;
       int sent = 0;
       while (nb_sms - sent > 0) {
         if j > 0 {
           //current authorisations suffice
           send();
           sent = sent + 1;
           j = j - 1
         } else {
           //acquire p new authorisations
           auth (Parameters.p);
           j = Parameters.p;
         }
       }
       return 0;
  }
}
```

Fig. 6. Program for sending a message using authorisation chunks of size p

and then sends the messages where authorisations are acquired in blocks of size

p, for arbitrary fixed $p \geq 0$. The primitives for sending and authorising messages are modelled as additional (static) methods.

Figure 7 shows the bytecode for method `block_send`, which comprises six basic blocks. In order to verify that this method does not send more messages

```
0   aload_1 //variable s
1   invokestatic sizestring
4   istore_3 //variable n
5   aload_2 // variable b
6   invokestatic sizebook
9   istore 4 //variable m
11  iload_3
12  iload 4
14  imul
15  istore 5 //variable nbms
17  iconst_0
18  istore 6 //variable j
20  iconst_0
21  istore 7 //variable sent

23  iload 5
25  iload 7
27  isub
28  ifle 64

31  iload 6
33  ifle 54

36  invokestatic send
39  iload 7
41  iconst_1
42  iadd
43  istore 7
45  iload 6
47  iconst_1
48  isub
49  istore 6
51  goto 23

54  iconst_3 // parameter p
55  invokestatic auth
58  iconst_3
59  istore 6
61  goto 23

64  iconst_0
65  ireturn
```

Fig. 7. Bytecode for method `BlockBooking.block_send`.

than authorised, we derive the typing

$$[s \mapsto C, b \mapsto D], [\,], \emptyset \vdash_{\Sigma,\Lambda} \texttt{block\_send}, 0 : \{0\}, (0,0)$$

where C and D are arbitrary and

$$\begin{aligned}
\Sigma &\equiv [\texttt{sizestring} \mapsto \{(C,0,0,\mathbf{Z})\}, \texttt{sizebook} \mapsto \{(D,0,0,\mathbf{Z})\}] \\
\Lambda &\equiv [23 \mapsto \{spec_d \mid 0 \leq d\}] \\
spec_d &\equiv (\Delta_d, [\,], \Xi_d, \{0\}, (d,0)) \\
\Delta_d &\equiv [n \mapsto \mathbf{Z}, m \mapsto \mathbf{Z}, nbsms \mapsto \mathbf{Z}, j \mapsto \{d\}, sent \mapsto \mathbf{Z}^{\geq 0}] \\
\Xi_d &\equiv \{(n,n),(m,m),(nbsms,nbsms),(j,j),(sent,sent)\}.
\end{aligned}$$

The proof context Λ contains a single entry, namely a polyvariant loop invariant for instruction 23. The invariant contains one entry for each $0 \leq d$, where the index specifies precisely the content of variable j and links this value to the pre-effect. The equivalence relation relevant at this program point contains

merely the reflexive entries for all (integer) variables. The verification of the above judgement applies the rules syntax-directedly for instructions $0, \ldots, 21$, and then applies the axiom rule for label 23, guarded by an application of rule E-Sub.

The overall verification complements the verification of the above judgement with a justification of the context Λ, by providing a progressive derivation for the loop invariant. Again, this verification proceeds syntax-directedly through the loop, terminating in (subtyping-protected) applications of the rule E-Ax. At the point where method `send` is invoked (instruction label 36) a case-split is performed on the condition $d = 0$. If this condition holds, a vacuous statement is obtained as the invocation occurs in the branch $j > 0$, and our invariant ensures that j contains the value d. The vacuity is detected as the entry for j in Δ is $\emptyset$ at that point: the load instruction at label 36 inserts $(0, j)$ into Ξ, hence the type associated with j in the fall-through-hypothesis of the branch at label 33 (in particular: at label 36) is $\{d\} \cap (\mathbf{Z} \setminus \{0\}) = \emptyset$ where the term $\{d\}$ was propagated unmodified to instruction 36 from instruction 23. Consequently, the case $d = 0$ may be immediately discharged by an invocation of rule E-Univ. The case $d > 0$ admits the application of the proof rule E-Send, and the remainder of the branch is again proven in a syntax-directed fashion.

Type checking and inference Again, we briefly discuss these issues for this system. The type system is generic in that types may be arbitrary sets of integers. In order to support effective typechecking and inference one must of course restrict these sets themselves and also the sets of types that arise in annotations and method specifications. A popular and for our intended application sufficient way consists of restricting types to convex polyhedra specified by a system of linear inequalities and to confine sets of types to those arising by intersecting a fixed convex polyhedron with a hyperplane specified by one or more additional parameters. Notice that the types in our running example are all of this form.

When we make this restriction (formally by applying the subtyping rule immediately after each rule to bring the types back into the polyhedral format) then type checking amounts to checking inclusion of convex polyhedra which can be efficiently performed by linear programming. Furthermore, Farkas' Lemma also furnishes short, efficiently computable, and efficiently checkable certificates [21, 28]. Indeed, since any convex polyhedron is the intersection of hyperplanes, deciding containment of convex polyhedra reduces to deciding whether a convex polyhedron $H = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$ is contained in a hyperplane of the form $P = \{\boldsymbol{x} \mid \boldsymbol{c}^T\boldsymbol{x} \leq d\}$. This, however, is the case iff $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in H\} \leq d$; a linear programming problem. Now, the latter inequality can be certified by providing a vector $\boldsymbol{r} \geq 0$ (componentwise) such that $\boldsymbol{r}^T A = \boldsymbol{c}^T$ and $\boldsymbol{r}^T\boldsymbol{b} \leq d$. For then, whenever $\boldsymbol{x} \in H$, i.e., $A\boldsymbol{x} \leq \boldsymbol{b}$ then $\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{r}^T A\boldsymbol{x} \leq \boldsymbol{r}^T\boldsymbol{b} \leq d$. Farkas' lemmas asserts that such a vector $\boldsymbol{r}$ exists whenever $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in H\} \leq d$. Given its existence we can efficiently compute it by minimising $\boldsymbol{y}^T\boldsymbol{b}$ subject to $\boldsymbol{y}^T A = \boldsymbol{c}^T$ and $\boldsymbol{y} \geq 0$.

Regarding automatic type inference as opposed to type checking one has to find unknown convex polyhedra specified by fixpoint equations. Besson et al. [12]

report that this can be done by iteration using widening heuristics from [19]. The range and efficiency remains, however, unexplored in loc. cit. In our particular application we expect constraints to be sufficiently simple so that these heuristics or those proposed in [26] will be successful. Inference of the equivalence relations Ξ can be achieved by employing standard copy-propagation techniques known from compiler constructions.

6 Discussion

We have described the use of the Mobius base logic as a unified backend for both program analyses and type systems. The Mobius base logic has been formally proved sound with respect to the Bicolano formalisation of the JVM. Compared to direct soundness proofs of type systems and analyses with respect to Bicolano the use of the Mobius base logic as an intermediary offers two distinctive advantages. First, the soundness proof of the Mobius base logic already does much of the work that is common to soundness proofs, in particular inducting on steps in the operational semantics and stack height. The Mobius logic is more transparent and allows for proof by invariant and recursion. Secondly, the standardised format of assertions in the Mobius base logic makes it easier to compare results of different type systems and analyses and also to assess whether the asserted property coincides with the intuitively desired property.

The resource extension to both Bicolano and the Mobius base logic allows for direct specification and certification of resource-related intensional properties without having to go through indirect observations such as values of ordinary program variables that are externally known to reflect some resource behaviour. This is particularly important in the PCC scenario where providers and users of specifications and certificates do not coincide and might have different objectives.

Similarly, the strong invariants enhance the expressive power of the Mobius base logic compared to standard Hoare logics in that resource behaviour of nonterminating programs is appropriately accounted for. In this way, the usual strong guarantees of type systems and program analyses may be adequately reflected in the logic.

We have demonstrated this use of the Mobius base logic on one of the Mobius case studies: a block-booking scheme whose deployment could avoid the inflation of permission requests that lead to social vulnerabilities.

Acknowledgements This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein. We are grateful to all members of the MOBIUS Working Group on work package 3, in particular Benjamin Gregoire, David Pichardie, Aleksy Schubert and Randy Pollack, for the numerous discussions on program logics, JML, and types, and on formalising these in theorem provers. The constructive feedback from the reviewers helped us to improve content and presentation of the paper.

References

1. E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *Logic for Programming Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 380–397. Springer-Verlag, 2005.
2. A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.
3. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Theorem Proving in Higher-Order Logic*, volume 3223 of *Lecture Notes in Computer Science*, pages 34–49, Berlin, Sept. 2004. Springer-Verlag.
4. F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.
5. G. Barthe and C. Fournet, editors. *Trustworthy Global Computing, Third Symposium (TGC'07), Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
7. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, Apr. 2003.
8. L. Beringer and M. Hofmann. A bytecode logic for JML and types. In *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science 4279, pages 389–405. Springer-Verlag, 2006.
9. L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Workshop*. IEEE Press, 2007.
10. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452, pages 347–362. Springer-Verlag, 2005.
11. F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 2006.
12. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Inria Research Report 6333, 2007.
13. D. Cachera, T. P. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer-Verlag, 2005.
14. B. Chang, A. Chlipala, and G. Necula. A framework for certified program analysis and its applications to mobile-code safety. In E. Emerson and K.S.Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI'06), Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, 2006.
15. B. Chang, A. Chlipala, G. Necula, and R. Schneck. The open verifier framework for foundational verifiers. In J. Morrisett and M. Fähndrich, editors, *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 1–12. ACM Press, 2005.

16. MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements. Available online from http://mobius.inria.fr, 2006.
17. MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.
18. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, Aug. 1977.
19. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.
20. P. Czarnik and A. Schubert. Extending operational semantics of the java bytecode. In Barthe and Fournet [5], pages 57–72.
21. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
22. X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 67–78, New York, NY, USA, January 2007. ACM Press.
23. R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In U. Montanari, D. Sannella, and R. Bruni, editors, *Trustworthy Global Computing, Second Symposium (TGC'06), Revised Selected Papers*, volume 4661 of *Lecture Notes in Computer Science*, pages 116–131. Springer-Verlag, 2007.
24. M. Hofmann and M. Pavlova. Elimination of ghost variables in program logics. In Barthe and Fournet [5], pages 1–20.
25. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.
26. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. ACM POPL 2004*, pages 330–341, 2004.
27. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
28. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC Computer Science Laboratory, June 1981.
29. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2002.
30. D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inia.fr/bicolano. Summary appears in [17], 2006.
31. C. L. Quigley. A Programming Logic for Java Bytecode Programs. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, (TPHOLs'03), Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.
32. M. Wildmoser. *Verified Proof Carrying Code*. PhD thesis, Institut für Informatik, Technische Universität München, 2005.
33. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, Aug. 2004.
34. T. Y. Woo and S. S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.