A Bytecode Logic for JML and Types

Lennart Beringer and Martin Hofmann

Institut für Informatik, Universität München
Oettingenstrasse 67, 80538 München, Germany
{beringer, mhofmann}@tcs.ifi.lmu.de

Abstract. We present a program logic for virtual machine code that may serve as a suitable target for different proof-transforming compilers. Compilation from JML-specified source code is supported by the inclusion of annotations whose interpretation extends to non-terminating computations. Compilation from functional languages, and the communication of results from intermediate level program analysis phases are facilitated by a new judgement format that admits the compositionality of type systems to be reflected in derivations. This makes the logic well suited to serve as a language in which proofs of a PCC architecture are expressed. We substantiate this claim by presenting the compositional encoding of a type system for bounded heap consumption. Both the soundness proof of the logic and the derivation of the type system have been formally verified by an implementation in Isabelle/HOL.

1 Introduction

Modeling languages such as JML [25] allow the software architect to specify functional and non-functional behaviour of code modules. Typically, these languages comprise a variety of specification idioms such as partial-correctness specifications using pre- and post-conditions, termination measures, specification of exceptional behaviour, model fields, ghost variables and fields, invariants at object or class level, lightweight specifications, or the inclusion of pure (i.e. non-side-effecting) code in specification clauses. Although the precise interpretation of some of these features is still a matter of ongoing debate, a number of verification tools have been presented that validate code w.r.t. JML specifications [14]. Although the proposed formalisms mainly target Java source code, they can relatively easily be adapted to bytecode.

The adaptation of specification constructs to low-level code admits a smooth translation of high-level specifications into specifications of mobile code units. However, we do not expect that a similarly direct transfer of validation strategies such as verification condition generators would suffice for their verification, for two reasons. Firstly, bytecode that was obtained by compilation from languages other than Java may not be amenable to the same proof strategies, or may lead to different verification conditions if it has undergone an obfuscation routine. Secondly, a recipient may require transmitted code to be complemented by a proof certifying that the code is safe to execute [28]. Typically, the production of certificates exploits results of program analyses such as type systems. In this case, the validation of certificates by the code consumer is supported if the type system's structuring principles (invariants) are communicated as part

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 389–405, 2006.

of the certificate [4,13]. Again, it is not guaranteed that these abstraction barriers are respected by a verification strategy for source code verification.

In this paper, we therefore propose a program logic for a bytecode language that satisfies requirements motivated by JML specifications and admits different verification strategies to be implemented, including strategies that are suitable for validating high-level type systems. More specifically, we present a formalism where partial-correctness method specifications can be complemented by method invariants and local annotations at intermediate program points whose interpretation applies to terminating *as well as non-terminating* program executions. Non-terminating executions are not covered by traditional (partial or total) Hoare logics, but are required for a faithful interpretation of JML code annotations. They are also desirable for proof-carrying code (PCC) frameworks: the significance of a certificate regarding the safety or the consumption of resources is increased if its validity does not derive from a partial-correctness interpretation - for example, consider a certificate purporting to guarantee an upper bound on the runtime. On the other hand, non-terminating program executions are often implicitly covered by program analysis formalisms such as type systems, but this fact is often not stated (or proven) explicitly, for example if the soundness proof is formulated as a syntactic subject-reduction proof w.r.t. a big-step operational semantics. In order to demonstrate the suitability of our logic for the interpretation of such type systems, we present the syntax-directed encoding of a type system for bounded heap consumption which covers terminating and non-terminating executions.

For presentational reasons, the program logic described in the present paper covers only a small fragment of the JVML. However, in collaboration with partners from the Mobius project [8], a variation of the logic has been produced that covers a more substantial subset of JVML, including virtual method invocations, static fields, arrays, exceptions, and various datatypes. At the same time, work is under way to translate JML specification constructs that are not considered in the present paper into the extended logic, in particular the constructs of JML specification level 0 [25].

Motivation and overview of assertion format. The format of judgements in a program logic is strongly influenced by semantic considerations, i.e. by the conclusions one may draw from a derivable judgement regarding the operational behaviour. Our logic aims to fulfill two sets of requirements. The first requirement concerns JML annotations at intermediate program points. Their common understanding mandates that an assertion A associated to a program point ℓ should be satisfied whenever the control flow reaches ℓ. At first sight, this interpretation motivates a notion of validity like

$$\forall\, s.\ \ell_0, s_0 \to^* \ell, s \Rightarrow A(s) \tag{1}$$

where s_0 denotes the entry state of the program fragment (e.g. method) and ℓ_0 the label of the first instruction. Indeed, this interpretation extends partial-correctness program logics by also applying to non-terminating program executions. Furthermore, the generalisation to binary predicates A, with validity defined by

$$\forall\, s.\ \ell_0, s_0 \to^* \ell, s \Rightarrow A(s_0, s), \tag{2}$$

admits assertions to refer to the initial state, as is required for the translation of idioms such as JML's `old` keyword [22].

Although program logics motivated by such an interpretation have been proposed [32,7,1], the resulting proof systems appear unsatisfactory, since they mandate the concurrent satisfaction of local conditions at all program labels, for a fully annotated program. For example, the proof rule for program points in Rinard's logic [32] involves a universal quantification over all predecessor labels. This, in our opinion, precludes local reasoning, by which we mean that the validity of an assertion at a program point ℓ should refer to the phrase represented by ℓ. Local behaviour is the source from which type systems for high-level languages draw their compositionality. In order to achieve our second goal, the interpretation of type systems, it appears necessary that this behaviour be reflected in the logic. Thus, an assertion at ℓ should constrain executions *from* ℓ *onwards*, irrespective of the path used to *reach* ℓ. While this demand contradicts a formulation following (1), it would enable us to exploit the syntax-directedness of typing rules in the proofs of derived proof rules, i.e. of lemmas for a syntactically determined subclass of assertions.

In Bannwart and Müller's logic [7], program points are decorated with (unary) assertions E that are interpreted w.r.t. a partial-correctness specification of the surrounding method. Assuming a fully specified program, each local judgement $\vdash \{E_\ell\}\ \ell$ is valid if the satisfaction of E_ℓ in the state prior to executing the instruction at ℓ guarantees the satisfaction of the assertions of all successor labels of ℓ:

$$\forall\, s.\ \ell_0, s_0 \rightarrow^* \ell, s \Rightarrow E_\ell(s) \Rightarrow \forall\, \ell'\ s'.\ \ell, s \rightarrow \ell', s' \Rightarrow E_{\ell'}(s'). \tag{3}$$

Thus, E_ℓ denotes a *pre-condition* for $E_{\ell'}$ and consequently (by transitivity) for the method specification (which is identical to the specification of the return instruction). However, this format does not admit a rule of consequence, as E_ℓ in (3) suggests that assertions could be strengthened, while $E_{\ell'}$ suggests that they can be weakened, which is also what one would expect from JML annotations. Furthermore, the fact that the final state is only mentioned indirectly, via the implicit reference to the method specification, is an obstacle to local reasoning: the method specification relates a final state of a (terminating) execution only to the *initial* state, but not the state at label ℓ.

Our proposed solution consists of introducing several assertion forms, with specific roles. Judgements explicitly relate a program point ℓ to a (binary) *pre-condition* A, a (ternary) *post-condition* B, and a (ternary) *invariant* I, and implicitly refer to a global table Q that assigns (binary) *annotations* Q to some program points (not all program points are required to be annotated). Informally, the interpretation of such a judgement asserts that whenever ℓ is reached from s_0 with current state s, and $A(s_0, s)$ holds, then

- $B(s_0, s, t)$ holds, provided that the method terminates with final state t
- $I(s_0, s, H)$ holds, provided that H is the heap component of any state arising during the continuation of the method invocation surrounding s, *including invocations of further methods*, i.e. subframes
- $Q(s_0, s')$ holds, provided that s' is reached at some label ℓ' during the continuation of the method invocation surrounding s, *but not including subframes*, where $\mathsf{Q}(\ell') = Q$

In order to support the descent into subframes in the interpretation of invariants, partial-correctness method specifications are complemented by *method invariants* which relate

the frame-initial state to the heap component of any state arising during the execution of the method (*including subframes*), irrespective of its termination behaviour. Both kinds of invariants are thus *strong invariants* in the sense of Hähnle and Mostowski [19]: they mandate that the property holds *throughout* the execution of a program fragment, instead of merely stipulating that the property holds upon termination whenever it was satisfied in the initial state. The decision to consider only a state's heap component in invariants is motivated by the fact that the operand stack and the (naming of) local variables should be considered implementation details of a method. For example, the substitution of a method by an improved implementation that uses different local variables should not affect invariants of surrounding methods.

The proposed format admits the expected rule of consequence where pre-conditions can be strengthened, while post-conditions and invariants may be weakened. Furthermore, JML annotations are directly supported as these may be collected in Q and will be satisfied whenever the annotated label is visited, irrespective of the termination behaviour. References to the frame-initial state are also supported, thus enabling the direct translation of specification idiom `old`. Finally, the format enables syntax-directed interpretations of type systems as all items involved in the execution of the code fragment *starting* at ℓ are available in the judgement for ℓ. Conceptually, the emphasis on syntactic structure that distinguishes our logic from the above-mentioned work appears similar to the difference between Hoare logic and Floyd's reasoning techniques for flowcharts.

Synopsis. The remainder of this paper is structured as follows: in Section 2, we present syntax and operational semantics of a small bytecode language which serves as our vehicle for presenting the logic. This allows us (Section 3) to formally define our notion of validity. We then present the proof system and outline its soundness proof. We demonstrate the suitability of the logic for giving interpretations of type systems that affect terminating and non-terminating program executions by outlining the encoding of a type system for bounded heap consumption in Section 4. Finally, we conclude and discuss related work. The material presented in Sections 2 to 4 is based on a development of the logic in the theorem prover Isabelle/HOL, including a formalised soundness proof and a formal derivation of the encoded typing rules. Following the approach advocated by Kleymann [24], the formalisation uses a deep embedding of the programming language syntax, while assertions are embedded shallowly in the meta-logic of the theorem prover. The corresponding Isabelle sources are available from [12].

2 Syntax and Dynamic Semantics

For the purpose of this paper, we consider instructions

$$\begin{aligned} ins ::= {} & \mathsf{Load}\ x \mid \mathsf{Store}\ x \mid \mathsf{Const}\ z \mid \mathsf{Unop}\ u \mid \mathsf{Binop}\ o \mid \mathsf{New}\ c \mid \mathsf{Getfield}\ c\ f \mid \\ & \mathsf{Putfield}\ c\ f \mid \mathsf{Goto}\ l \mid \mathsf{If0}\ l \mid \mathsf{Invokestatic}\ M \mid \mathsf{Return} \end{aligned}$$

where x ranges over a set $\mathcal{X}$ of (local) variables (also called registers), z over integer constants and Null, u and o over unary and binary operations (like isNull, add, mul,...), respectively, c over a set $\mathcal{C}$ of class names, f over a set $\mathcal{F}$ of field names, l over a set $\mathcal{L}$ of program labels, and m over a set $\mathcal{M}$ of method names. All these sets are assumed to

be mutually distinct. Method identifiers $M = (c, m)$ combine class and method names, and program points ℓ are of the form $\ell = M, l$.

A method definition $(par, l, body, suc)$ consists of a list $par = [x_1, \ldots, x_n]$ of (distinct) formal parameters, the label l of the first instruction, a method body $body$, represented as a finite map from program labels l to instructions, and a partial function $suc : \mathcal{L} \rightharpoonup_{fin} \mathcal{L}$ that maps labels to their control flow successors.

A program consists of a finite map from method identifiers to method definitions. All notions in the remainder of this paper are formulated with respect to an arbitrary but fixed program, which we denote by P. For $P(M) = (par, l, body, suc)$ we also write $init_M$ for l, $M(l)$ for $body(l)$, and suc_M for suc.

The dynamic semantics is defined over a set $\mathcal{V}$ of values that is ranged over by v and comprises constants z and addresses $a \in \mathcal{A}$. JVM states $s \in \Sigma$ are built from operand stacks, stores, and heaps

$$\begin{array}{ll} O \in \mathcal{O} = \mathcal{V}\ list & s \in \Sigma = \mathcal{O} \times \mathcal{S} \times \mathcal{H} \\ S \in \mathcal{S} = \mathcal{X} \rightharpoonup_{fin} \mathcal{V} & s_0 \in \Sigma_0 = \mathcal{S} \times \mathcal{H} \\ H \in \mathcal{H} = \mathcal{A} \rightharpoonup_{fin} \mathcal{C} \times (\mathcal{F} \rightharpoonup_{fin} \mathcal{V}) & t \in \mathcal{T} = \mathcal{H} \times \mathcal{V}. \end{array}$$

The categories Σ_0 and $\mathcal{T}$ represent initial and terminal states which occur at the beginning (end) of a frame's execution. For $s_0 = (S, H)$ we write $state(s_0) = ([\,], S, H)$ for the local state that extends s_0 with an empty operand stack. For $par = [x_1, \ldots, x_n]$ and $O = [v_1, \ldots, v_n]$ we write $par \mapsto O$ for $[x_i \mapsto v_i]_{i=1,\ldots,n}$. Finally, we write $heap(s)$ to access the heap component of a state s, and similarly for initial and terminal states.

As in [7], the operational semantics is given by two judgements, a small-step relation $\Rightarrow \subseteq (\mathcal{L} \times \Sigma) \times (\mathcal{L} \times \Sigma)$, and its closure up to the end of the current frame, $\Downarrow \subseteq (\mathcal{L} \times \Sigma) \times \mathcal{T}$. Both relations are indexed by the current method. The (mutually recursive) relationship between these relations, and the rules for New, Goto, and Invokestatic are shown in Figure 1. The rules for the other instruction forms are similar.

$$\textsc{New}\ \frac{M(l) = \mathsf{New}\ c \qquad a \notin dom\ H}{\vdash_M l, (O, S, H) \Rightarrow suc_M(l), (a :: O, S, H[a \mapsto (c, [\,])])} \qquad \textsc{Goto}\ \frac{M(l) = \mathsf{Goto}\ l'}{\vdash_M l, s \Rightarrow l', s}$$

$$\textsc{Invs}\ \frac{M(l) = \mathsf{Invokestatic}\ M' \qquad M' \in dom\ P \qquad \vdash_{M'} init_{M'}, state(par_{M'} \mapsto O', H) \Downarrow H', v}{\vdash_M l, (O' @ O, S, H) \Rightarrow suc_M(l), (v :: O, S, H')}$$

$$\textsc{Comp}\ \frac{\vdash_M l, s \Rightarrow l', s' \qquad \vdash_M l', s' \Downarrow t}{\vdash_M l, s \Downarrow t} \qquad \textsc{Return}\ \frac{M(l) = \mathsf{Return}}{\vdash_M l, (v :: O, S, H) \Downarrow H, v}$$

Fig. 1. Operational semantics: relations $\Rightarrow$ and $\Downarrow$ (excerpt)

3 Program Logic

3.1 Format of Assertions and Judgements

Judgements associated with program points involve formulae of the following three forms, where $\mathcal{B}$ denotes the set of booleans.

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

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

Invariants. $I \in Inv = (\Sigma_0 \times \Sigma \times \mathcal{H}) \to \mathcal{B}$ relate the initial state of the current method, the current state, and the heap component of a state of the current frame or a subframe of the current frame.

The behaviour of methods is described using two assertion forms.

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

Method invariants. $\varphi \in MethInv = (\Sigma_0 \times \mathcal{H}) \to \mathcal{B}$ constrain the behaviour of terminating and non-terminating method executions by relating the initial state of a method frame to all heaps that occur during the execution of the method.

A program specification consists of two parts. The method specification table $\mathsf{M} : (\mathcal{C} \times \mathcal{M}) \to (MethSpec \times MethInv)$ defines the externally visible behaviour. In addition, local annotations Q which constrain the behaviour at intermediate program points are collected in a partial map $\mathsf{Q} : ((\mathcal{C} \times \mathcal{M}) \times \mathcal{L}) \rightharpoonup_{fin} Assn$. For the remainder of this section, let M and Q denote some arbitrary but fixed specification and annotation tables.

3.2 Interpretation of Assertions and Judgements

In addition to the operational judgements defined in Figure 1, the interpretation of the program logic refers to two auxiliary relations. The first one, denoted by $\vdash_M l, s \Rightarrow^* l', s'$, is the reflexive and transitive closure of $\Rightarrow$ and is defined in the standard way. The second relation, denoted by $\vdash_M l, s \Uparrow s'$ and defined in Figure 2, extends $\Rightarrow^*$ by also relating l, s to s' if s' is a state that occurs later than s either in the same frame as s or in a subframe of that frame. This is achieved by the rule R-INVS that relates the call-state of a method invocation to the initial state of the subframe.

$$\text{R-REFL}\ \frac{}{\vdash_M l, s \Uparrow s} \qquad \text{R-TRANS}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Uparrow s''}{\vdash_M l, s \Uparrow s''}$$

$$\text{R-INVS}\ \frac{M(l) = \mathsf{Invokestatic}\ M' \qquad M' \in dom\ P \qquad \vdash_{M'} init_{M'}, state(par_{M'} \mapsto O', H) \Uparrow s}{\vdash_M l, (O'@O, S, H) \Uparrow s}$$

Fig. 2. Auxiliary operational relation $\Uparrow$

Definition 1. *A triple* (A, B, I) *is* valid at $\ell = M, l$, *notation* $\models \{A\}\, \ell\, \{B\}\, (I)$, *if for all* s_0 *and* s *with* $\vdash_M init_M, s_0 \Rightarrow^* l, s$ *and* $A(s_0, s)$,

- *if* $\vdash_M l, s \Downarrow t$ *then* $B(s_0, s, t)$,
- *if* $\vdash_M l, s \Uparrow s'$ *then* $I(s_0, s, heap(s'))$, *and*
- *if* $\vdash_M l, s \Rightarrow^* l', s'$ *and* $\mathsf{Q}(M, l') = Q$ *then* $Q(s_0, s')$.

Note that the third clause applies to annotations Q associated with *future* labels l' in the same method M, and that these are interpreted without direct recourse to the current state s, although the proof of $Q(s_0, s')$ may exploit the precondition $A(s_0, s)$.

In order to store recursive proof assumptions during the verification of loops, proof contexts G may be used. These are finite maps which associate triples (A, B, I) to program points ℓ.

Definition 2. *Context G is called* valid*, notation $\models G$, if $\models \{A\}\, \ell\, \{B\}\, (I)$ holds for all $G(\ell) = (A, B, I)$. Similarly, specification table M is valid, notation $\models \mathsf{M}$, if all M, Φ and φ with $\mathsf{M}(M) = (\Phi, \varphi)$ satisfy $\models \{A\}\, M, init_M\, \{B_\Phi\}\, (I_\varphi)$, where*

$$\begin{aligned} A &= \lambda\, (s_0, s).\ s = state(s_0) \\ B_\Phi &= \lambda\, (s_0, s, t).\ s = state(s_0) \rightarrow \Phi(s_0, t),\ \textit{and} \\ I_\varphi &= \lambda\, (s_0, s, H).\ s = state(s_0) \rightarrow \varphi(s_0, H). \end{aligned}$$

Finally, program P is valid, notation $\models P$, if there is a G such that $\models G$ and $\models \mathsf{M}$.

3.3 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 operators for simple instructions,

$$\begin{aligned} PRE(M, l, A)(s_0, r) &= \exists\, s\, l'.\ \vdash_M l, s \Rightarrow l', r \wedge A(s_0, s) \\ POST(M, l, B)(s_0, r, t) &= \forall\, s\, l'.\ \vdash_M l, s \Rightarrow l', r \rightarrow B(s_0, s, t) \\ INV(M, l, I)(s_0, r, H) &= \forall\, s\, l'.\ \vdash_M l, s \Rightarrow l', r \rightarrow I(s_0, s, H) \end{aligned}$$

resemble WP-operators, but are separately defined for pre-conditions, post-conditions, and invariants. In the case of method invocations, we replace the reference to the operational judgement by a reference to the method specification, and include the construction and destruction of a frame

$$\begin{aligned} PRE_{sinv}(\Phi, A, par) &= \lambda\, (s_0, s).\ \exists\, O\ S\ H'\ H\ O'\ v.\ s = (v :: O, S, H') \wedge \\ &\qquad \Phi((par \mapsto O', H), (H', v)) \wedge A(s_0, (O'@O, S, H)) \\ POST_{sinv}(\Phi, B, par) &= \lambda\, (s_0, s, t).\ \forall\, O\ S\ H'\ H\ O'\ v.\ s = (v :: O, S, H') \rightarrow \\ &\qquad \Phi((par \mapsto O', H), (H', v)) \rightarrow B(s_0, (O'@O, S, H), t) \\ INV_{sinv}(\Phi, I, par) &= \lambda\, (s_0, s, H).\ \forall\, O\ S\ H'\ H''\ O'\ v.\ s = (v :: O, S, H') \rightarrow \\ &\qquad \Phi((par \mapsto O', H''), (H', v)) \rightarrow I(s_0, (O'@O, S, H''), H) \end{aligned}$$

Finally, the rule for the conditional jump instruction involves operators that take the dependence on the outcome of the branch condition into account:

$$\begin{aligned} A^+ &= \lambda\, (s_0, s).\ \forall\, O\ S\ H.\ s = (0 :: O, S, H) \rightarrow A\ s_0\ s \\ A^- &= \lambda\, (s_0, s).\ \forall\, O\ S\ H\ z.\ s = (z :: O, S, H) \rightarrow z \neq 0 \rightarrow A\ s_0\ s \\ B^+ &= \lambda\, (s_0, s, t).\ \forall\, O\ S\ H.\ s = (0 :: O, S, H) \rightarrow B(s_0, s, t) \end{aligned}$$

$$B^- = \lambda\,(s_0, s, t).\ \forall\, O\ S\ H\ z.\ s = (z :: O, S, H) \to z \neq 0 \to B(s_0, s, t)$$
$$I^+ = \lambda\,(s_0, s, H).\ \forall\, O\ S\ H'.\ s = (0 :: O, S, H') \to I(s_0, s, H)$$
$$I^- = \lambda\,(s_0, s, H).\ \forall\, O\ S\ H'\ z.\ s = (z :: O, S, H') \to z \neq 0 \to I(s_0, s, H),$$

3.4 Proof Rules

The proof system is presented in Figures 3 and 4, and has two judgement forms, $G \vdash \{A\}\,\ell\,\{B\}\,(I)$ and $G \vdash \langle A\rangle\,\ell\,\langle B\rangle\,(I)$. Both forms associate a program point to a precondition, a postcondition, and an invariant, relative to a proof context G. The motivation for using two judgement forms stems from the interaction between the rules that alter the flow of control inside a method frame (for the language considered in this paper only conditional and unconditional jumps, but in general also instructions that may throw an exception) and the rule AX that extracts such assumptions from G. Our approach separates the *usage* of an assumption from its *justification*. The axiom rule can only be used to derive judgements of the form that is required in the hypothesis of the syntax-directed rules, $G \vdash \langle A\rangle\,\ell\,\langle B\rangle\,(I)$. In contrast, the definition of verified programs requires us to discharge an assumption $G(\ell) = (A, B, I)$ by exhibiting a proof of $G \vdash \{A\}\,\ell\,\{B\}\,(I)$. Such a proof *cannot* simply consist of an application of the rule AX, but will necessarily end (modulo applications of the rule CONSEQ-F) in a syntax-directed rule. Consequently, the justification of an assumption is forced to inspect the corresponding code block, eliminating the possibility to insert arbitrary (incorrect) assumptions. In order to chain together a sequence of syntax-directed rules, we introduce a further rule, INJ, that turns a derivation of $G \vdash \{A\}\,\ell\,\{B\}\,(I)$ into one of $G \vdash \langle A\rangle\,\ell\,\langle B\rangle\,(I)$ – but no rule is given for converting in the opposite direction. The separation into two judgement forms thus represents an alternative to global well-definedness conditions on derivation trees, as it enforces that assumptions in G can not be justified vacuously by reference to G but only by inspecting the corresponding code block. Semantically, the judgement forms differ in bounds the number of operational steps for which a judgement is required to be valid.

The proof rules are oriented such that the conclusion is an unconstrained judgement and proof hypotheses refer to successor instructions. Hence, a verification condition generator may be defined as a proof strategy that traverses the program in the direction of the flow of control.

Syntax-directed rules. The syntax-directed rules are shown in Figure 3, and are motivated as follows.

Rule INSTR describes the behaviour of *basic* instructions.

$$\mathit{basic}(M, l) \equiv M(l) \in \left\{ \begin{array}{l} \mathsf{Load}\ x, \mathsf{Store}\ x, \mathsf{Const}\ z, \mathsf{Unop}\ u, \mathsf{Binop}\ o, \\ \mathsf{New}\ c, \mathsf{Getfield}\ c\ f, \mathsf{Putfield}\ c\ f \end{array} \right\}$$

The hypothetical judgement for the successor instruction involves assertions that are related to the assertions in the conclusion by the basic transformers presented in the previous section. In addition, 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.

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

$$\text{INSTR}\ \frac{basic(M,l) \quad SC_1 \quad SC_2}{G \vdash \langle PRE(M,l,A)\rangle\, M, suc_M(l)\, \langle POST(M,l,B)\rangle\, (INV(M,l,I))}{G \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\text{GOTO}\ \frac{M(l) = \mathsf{Goto}\ l' \quad SC_1 \quad SC_2}{G \vdash \langle PRE(M,l,A)\rangle\, M, l'\, \langle POST(M,l,B)\rangle\, (INV(M,l,I))}{G \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\text{IF0}\ \frac{\begin{array}{c} M(l) = \mathsf{If0}\ l' \quad SC_1 \quad SC_2 \\ G \vdash \langle PRE(M,l,A^+)\rangle\, M, l'\, \langle POST(M,l,B^+)\rangle\, (INV(M,l,I^+)) \\ G \vdash \langle PRE(M,l,A^-)\rangle\, M, suc_M(l)\, \langle POST(M,l,B^-)\rangle\, (INV(M,l,I^-)) \end{array}}{G \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\text{INVS}\ \frac{\begin{array}{c} M(l) = \mathsf{Invokestatic}\ M' \quad M' \in dom\ P \quad \mathsf{M}(M') = (\Phi, \varphi) \quad SC_1 \quad SC_2 \\ \forall\, s_0\ O\ S\ H\ O'\ H'.\ A(s_0, (O'@O, S, H')) \rightarrow \varphi\, (par_{M'} \mapsto O', H')\ H \\ \qquad \rightarrow I(s_0, (O'@O, S, H'), H) \\ G \vdash\ \langle PRE_{sinv}(\Phi, A, par_{M'})\rangle\ M, suc_M(l)\ \langle POST_{sinv}(\Phi, B, par_{M'})\rangle \\ \qquad (INV_{sinv}(\Phi, I, par_{M'})) \end{array}}{G \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\text{RET}\ \frac{M(l) = \mathsf{Return} \quad SC_1 \quad SC_2}{\forall\, s_0\ v\ O\ S\ H.\ A(s_0, (v :: O, S, H)) \rightarrow B(s_0, (v :: O, S, H), (H, v))}{G \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

Fig. 3. Program logic: syntax-directed rules

In particular, SC_2 requires us to prove any annotation that is associated with the *current* label l, in contrast to the clause in the interpretation of judgements in Definition 1. Satisfaction of I in later states, and satisfaction of local annotations Q' of later program points are guaranteed by the judgement for $suc_M(l)$. Similarly, the rules for conditional and unconditional jumps include a hypothesis on the jump target, and side conditions for annotations and invariants. In the rule for conditional jumps, a further hypothesis models the fall-though case, and the dependency on the outcome of the branch condition is taken into account by the operators A^+ etc..

In rule INVS, the method invariant φ and the precondition A may be exploited to establish the invariant I. This ensures that I will be satisfied by all heaps that arise during the execution of M', as these heaps will always conform to φ. In contrast, the specification Φ is used to construct the assertions that occur in the judgement for the successor instruction. Both conditions reflect the transfer of the method arguments to the formal parameters of the invoked method corresponding to the constructions of a new frame in the operational semantics. Similarly, the return value and the final heap are (in a terminating execution) handed back to the invoking method, where they are used to construct the assertions for the successor instruction.

Finally, rule 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.

Logical rules. The logical rules are shown in Figure 4. We have rules of consequence

$$\textsc{Conseq-T}\ \frac{\begin{array}{ll} G \vdash \langle A' \rangle\, \ell\, \langle B' \rangle\, (I') & \forall\, s_0\ s.\ A(s_0, s) \to A'(s_0, s) \\ \forall\, s_0\ s\ t.\ B'(s_0, s, t) \to B(s_0, s, t) & \forall\, s\ H.\ I'(s_0, s, H) \to I(s_0, s, H) \end{array}}{G \vdash \langle A \rangle\, \ell\, \langle B \rangle\, (I)}$$

$$\textsc{Conseq-F}\ \frac{\begin{array}{ll} G \vdash \{A'\}\, \ell\, \{B'\}\, (I') & \forall\, s_0\ s.\ A(s_0, s) \to A'(s_0, s) \\ \forall\, s_0\ s\ t.\ B'(s_0, s, t) \to B(s_0, s, t) & \forall\, s\ H.\ I'(s_0, s, H) \to I(s_0, s, H) \end{array}}{G \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

$$\textsc{Inj}\ \frac{G \vdash \{A\}\, \ell\, \{B\}\, (I)}{G \vdash \langle A \rangle\, \ell\, \langle B \rangle\, (I)} \qquad \textsc{Ax}\ \frac{\begin{array}{ll} G(\ell) = (A, B, I) & \forall\, s_0\ s.\ A(s_0, s) \to I(s_0, s, \mathit{heap}(s)) \\ \multicolumn{2}{c}{\forall\, Q.\ \mathsf{Q}(\ell) = Q \to (\forall\, s_0\ s.\ A(s_0, s) \to Q(s_0, s))} \end{array}}{G \vdash \langle A \rangle\, \ell\, \langle B \rangle\, (I)}$$

Fig. 4. Program logic: logical rules

for both judgement forms, the above-mentioned rule for mediating between the two judgement forms, and the axiom rule. As is the case in traditional program logics, the rules of consequence allow pre-conditions to be strengthened, while post-conditions and invariants may be weakened.

Definition 3. *P is* verified*, notation $\vdash P$, if there is a G such that $G \vdash \{A\}\, \ell\, \{B\}\, (I)$ holds whenever $G(\ell) = (A, B, I)$, and for all M, Φ, and φ, $\mathsf{M}(M) = (\Phi, \varphi)$ implies*

$$G \vdash \{\lambda\, (s_0, s).\ s = \mathit{state}(s_0)\}\ M, \mathit{init}_M\ \begin{array}{l} \{\lambda\, (s_0, s, t).\ s = \mathit{state}(s_0) \to \Phi(s_0, t)\} \\ (\lambda\, (s_0, s, H).\ s = \mathit{state}(s_0) \to \varphi(s_0, H)) \end{array}$$

Note the correspondence of the latter condition with Definition 2.

3.5 Soundness

The proof of soundness establishes that verified programs are valid, and consists of two steps. We first prove that $G \vdash \{A\}\, \ell\, \{B\}\, (I)$ implies $\models \{A\}\, \ell\, \{B\}\, (I)$ under the hypothesis that all assumptions in G are valid, and likewise all method specifications in M. Following [29,5], this proof proceeds by introducing relativised notions of validity that restrict the interpretation of judgements to operational judgements of bounded height. The second step discharges the validity assumptions on G and M by proving that verified programs guarantee the validity of G and M for arbitrary bounds.

Theorem 1. *If $\vdash P$ then $\models P$.*

In particular, this theorem implies that for $\vdash P$ all method specifications in M are honoured by their respective method implementations. As the proof has been formalised in Isabelle/HOL [12] we omit the details.

4 Interpretation of Type Systems

In addition to supporting the verification of programs w.r.t. JML specifications, a program logic for bytecode should also support the compositional formulation of program analysis results. In this section, we demonstrate how this can be achieved for analyses phrased as type systems. As property of interest we consider static *constant* bounds on heap consumption, with allocation-free loops. For this task, Cachera et al. presented an abstract-interpretation-based analysis at the bytecode level which involves the formalisation of various program analysis tasks (identification of mutually recursive program structures, identification of method calls in loops,...) in the theorem prover [15]. The correctness proof of their analysis thus includes a verification of the inference mechanism. During the verification of concrete programs, the fixed-point iteration and the calculation of solutions to the resulting constraints are carried out in the theorem prover.

In contrast, our type-based approach proceeds as follows. We first define an assertion format that expresses when a code block whose initial instruction is located at ℓ is guaranteed not to allocate more than n memory cells. This results in a derived proof system for bytecode in which all judgements are of the restricted form. Then, we consider a simple (first-order) functional language and prove that code resulting from compiling this language into bytecode satisfies the boundary asserted by a high-level type system: derivability in the type system guarantees derivability in the specialised program logic for the assertion interpreting the type. Thus, we avoid the formalisation of any inference mechanism (type inference). Only the outcome of the inference, a digest of the typing derivation, needs to be communicated from proof producer to proof consumer.

As a further difference to Cachera et al., our analysis is phrased at an intermediate language level. This is motivated by the fact that modern compilers perform many analysis and optimisation tasks using intermediate code representations where additional program structure can be exploited. Given that our analysis as phrased as a type system, we chose to employ a low-level functional language similar to A-normal form [18]. The similarity between such languages and the imperative program representation *Static Single Assignment* (SSA, [16]) has been observed by Appel and Kelsey [3,23].

Specialised program logic for bytecode. For each number n, we define a triple $[\![n]\!] = (A, B, I)$ consisting of a precondition, a post-condition, and an invariant.

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

Here, $|H|$ denotes the size of heap H. We specialise the two judgement forms to

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

Thus, the derivability of a judgement $G \vdash \ell\ \{n\}$ guarantees that the code located at ℓ allocates at most n items, in terminating (postcondition B) and non-terminating (invariant I) executions. For $(A, B, I) = [\![n]\!]$ we also define the method specification

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

Specialising the logic to these judgement forms yields the following rules, with empty Q.

$$\text{C-New}\ \frac{M(l) = \mathsf{New}\ c \qquad G \vdash M, suc_M(l)\ \langle n\rangle}{G \vdash M, l\ \{n+1\}} \qquad \text{C-Instr}\ \frac{basic(M,l) \quad \neg M(l) = \mathsf{New}\ c \qquad G \vdash M, suc_M(l)\ \langle n\rangle}{G \vdash M, l\ \{n\}}$$

$$\text{C-Ret}\ \frac{M(l) = \mathsf{Return}}{G \vdash M, l\ \{0\}} \qquad \text{C-Goto}\ \frac{M(l) = \mathsf{Goto}\ l' \quad G \vdash M, l'\ \langle n\rangle}{G \vdash M, l\ \{n\}}$$

$$\text{C-If}\ \frac{M(l) = \mathsf{If0}\ l' \quad G \vdash M, l'\ \langle n\rangle \quad G \vdash M, suc_M(l)\ \langle n\rangle}{G \vdash M, l\ \{n\}}$$

$$\text{C-Invs}\ \frac{M(l) = \mathsf{Invokestatic}\ M' \qquad M' \in dom\ P \qquad G \vdash M, suc_M(l)\ \langle n\rangle \qquad \mathsf{M}(M') = Spec\ k}{G \vdash M, l\ \{n+k\}} \qquad \text{C-Inj}\ \frac{G \vdash \ell\ \{n\}}{G \vdash \ell\ \langle n\rangle}$$

$$\text{C-SubF}\ \frac{G \vdash \ell\ \{n\} \quad n \le m}{G \vdash \ell\ \{m\}} \qquad \text{C-SubT}\ \frac{G \vdash \ell\ \langle n\rangle \quad n \le m}{G \vdash \ell\ \langle m\rangle} \qquad \text{C-Ax}\ \frac{G(\ell) = n}{G \vdash \ell\ \langle n\rangle}$$

$$\text{C-VP}\ \frac{\begin{array}{l}\forall\ M.\ M \in dom\ P \to (\exists\ n.\ \mathsf{M}(M) = Spec\ n \wedge G \vdash M, init_M\ \{n\}) \\ \forall\ \ell\ A\ B\ I.\ G(\ell) = (A,B,I) \to (\exists\ n.\ (A,B,I) = [\![n]\!] \wedge G \vdash \ell\ \{n\})\end{array}}{\vdash P}$$

Intermediate-level type system. The syntax of the intermediate language is stratified into primitive expressions and general expressions [18]. We include primitives for constructing empty and non-empty lists, and a corresponding pattern match expression. In order to simplify the translation into bytecode, we use method identifiers M as function names.

$$\begin{array}{rl}\mathcal{P} \ni p ::= & i \mid \mathsf{uop}\ u\ x \mid \mathsf{bop}\ o\ x\ y \mid \mathsf{Nil} \mid \mathsf{Cons}(x,y) \mid M(x_1,\ldots,x_n) \\ \mathcal{E} \ni e ::= & \mathsf{prim}\ p \mid \mathsf{let}\ x = p\ \mathsf{in}\ e \mid \mathsf{if}\ x\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \\ & (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \Rightarrow e \mid \mathsf{Cons}(x,y)\ \Rightarrow e)\end{array}$$

A program $F : (\mathcal{C} \times \mathcal{M}) \rightharpoonup_{fin} (\mathcal{X}\ list \times \mathcal{E})$ consists of a collection of function declarations in the standard way. Figure 5 presents the rules for a type system with judgements of the form $\Sigma \rhd p : n$ and $\Sigma \rhd e : n$. 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. Program F is well-typed w.r.t. signature Σ, notation $\Sigma \rhd F$, if $dom\ \Sigma = dom\ F$ and for all M, $F(M) = (par, e)$ implies $\Sigma \rhd e : \Sigma(M)$.

Figure 6 defines a compilation $[\![e]\!]^C_l$ into the bytecode language. 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.

Semantic type soundness for primitive expressions now 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.

$$\text{T-NIL}\ \frac{p \notin \{\mathsf{Cons}(x,y), M(x_1,\ldots,x_n)\}}{\Sigma \rhd p : 0} \qquad \text{T-CONS}\ \frac{}{\Sigma \rhd \mathsf{Cons}(x,y) : 1}$$

$$\text{T-CALL}\ \frac{\Sigma(M) = n}{\Sigma \rhd M(x_1,\ldots,x_n) : n} \qquad \text{T-LET}\ \frac{\Sigma \rhd p : n \quad \Sigma \rhd e : m}{\Sigma \rhd \mathsf{let}\ x = p\ \mathsf{in}\ e : n+m}$$

$$\text{T-COND}\ \frac{\Sigma \rhd e_1 : n \quad \Sigma \rhd e_2 : n}{\Sigma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : n} \qquad \text{T-SUB}\ \frac{\Sigma \rhd e : m \quad m \leq n}{\Sigma \rhd e : n}$$

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

Fig. 5. Typing rules

$$\begin{aligned}
[\![i]\!]_l^C &= (C[l \mapsto \mathsf{const}\ i], l+1)\\
[\![\mathsf{uop}\ u\ x]\!]_l^C &= (C[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{unop}\ u], l+2)\\
[\![\mathsf{bop}\ o\ x\ y]\!]_l^C &= (C[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{load}\ y, l+2 \mapsto \mathsf{binop}\ o], l+3)\\
[\![\mathsf{Nil}]\!]_l^C &= (C[l \mapsto \mathsf{const\ Null}], l+1)\\
[\![\mathsf{Cons}(x,y)]\!]_l^C &= (C\begin{bmatrix} l \mapsto \mathsf{load}\ y, l+1 \mapsto \mathsf{load}\ x, l+2 \mapsto \mathsf{new\ LIST},\\ l+3 \mapsto \mathsf{store\ t}, l+4 \mapsto \mathsf{load\ t},\\ l+5 \mapsto \mathsf{putfield\ LIST\ HD}, l+6 \mapsto \mathsf{load\ t},\\ l+7 \mapsto \mathsf{putfield\ LIST\ TL}, l+8 \mapsto \mathsf{load\ t} \end{bmatrix}, l+9)\\
[\![M()]\!]_l^C &= (C[l \mapsto \mathsf{Invokestatic}\ M], l+1)\\
[\![M(x_1,\ldots,x_n)]\!]_l^C &= [\![M(x_1,\ldots,x_{n-1})]\!]_{l+1}^{C[l \mapsto \mathsf{load}\ x_n]}\\
[\![\mathsf{prim}\ p]\!]_l^C &= \mathit{let}\ (C_1,l_1) = [\![p]\!]_l^C\ \mathit{in}\ (C_1[l_1 \mapsto \mathsf{Return}], l_1+1)\\
[\![\mathsf{let}\ x = p\ \mathsf{in}\ e]\!]_l^C &= \mathit{let}\ (C_1,l_1) = [\![p]\!]_l^C, (C_2,l_2) = (C_1[l_1 \mapsto \mathsf{store}\ x], l_1+1)\\
&\quad \mathit{in}\ [\![e]\!]_{l_2}^{C_2}\\
[\![\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]_l^C &= \mathit{let}\ (C_E,l_2) = [\![e_2]\!]_{l+2}^C, (C_T,l_1) = [\![e_1]\!]_{l_2}^{C_E}\\
&\quad \mathit{in}\ (C_T[l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{If0}\ l_2], l_1)
\end{aligned}$$

$$\left[\!\!\left[\begin{array}{l}\mathsf{case}\ x\ \mathsf{of}\\ \quad \mathsf{Nil} \Rightarrow e_1\\ \mid \mathsf{Cons}(x,y)\ \Rightarrow e_2\end{array}\right]\!\!\right]_l^C = \mathit{let}\ (C_C,l_N) = [\![e_2]\!]_{l+9}^C, (C_N,l_1) = [\![e_1]\!]_{l_N}^{C_C}\ \mathit{in}$$
$$(C_N\begin{bmatrix} l \mapsto \mathsf{load}\ x, l+1 \mapsto \mathsf{unop}\ (\lambda\, v.\, v = Nullref),\\ l+2 \mapsto \mathsf{If0}\ l_N, l+3 \mapsto \mathsf{Load}\ x,\\ l+4 \mapsto \mathsf{Getfield\ LIST\ HD}, l+5 \mapsto \mathsf{Store}\ h,\\ l+6 \mapsto \mathsf{Load}\ x, l+7 \mapsto \mathsf{Getfield\ LIST\ TL},\\ l+8 \mapsto \mathsf{Store}\ t \end{bmatrix}, l_1)$$

Fig. 6. Translation into bytecode

Proposition 1. *If* $\Sigma \rhd p : n$, $[\![p]\!]_l^C = (C_1, l_1)$, *and* $G \vdash M, l_1\ \langle m \rangle$, *then* $G \vdash M, l\ \{n+m\}$.

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

Proposition 2. *If* $\Sigma \rhd e : n$ *and* $[\![e]\!]^C_l = (C_1, l_1)$ *then* $G \vdash M, l \ \{n\}$.

Both results are easily proven by induction on the typing judgement. For presentational reasons we have omitted technical side conditions that ensure that the table M contains precisely the interpretations of Σ, and that the global program P contains precisely the translations of F, where for each entry, we reverse the list of formal parameters due to the order in which the translation pushes arguments onto the operand stack. Denoting these conditions by $[\![\Sigma]\!]$ and $[\![F]\!]$, respectively, we obtain overall type soundness, i.e. the verifiability of well-typed programs:

Theorem 2. *If* $\Sigma \rhd F$ *then* $\vdash [\![F]\!]$.

Again, the proof has been formalised in Isabelle/HOL [12].

5 Discussion

We presented a program logic for bytecode suitable for translating features found in modern specification formalisms and for interpreting type systems in a compositional way. Using a judgement format which separates postconditions, invariants, and annotations, the logic supports reasoning about terminating and non-terminating executions.

The necessity of complementing partial-correctness assertions by guarantees that apply to intermediate states and non-terminating computations has also been observed by Hähnle and Mostowski [19]. Based on an extension of first-order dynamic logic with trace modalities [9], they discuss the verification of transaction properties in the context of JAVACARD. Similar requirements arise from object invariants [26] and idioms like ESC-Java's *validity* of objects [17]. The logics developed in connection with the LOOP tool (e.g. [21]) apply at the source code level, or a representation of source code and (JML) specifications in a theorem prover. Various termination modes are considered in [21], but some rules, such as the rule for while, can only be applied in special circumstances. The logic is formulated as a set of derived proof rules, so proof search may always fall back on the underlying operational semantics. In contrast, our formulation as a syntactic proof system admits a study of (relative) completeness, following the approach of Kleymann, Nipkow, and ourselves [24,29,5].

The usage of expressive program logics as a mediating formalism between the operational semantics of a low-level language and type systems was already explored in our previous work [13]. Here, we presented an interpretation in a partial-correctness program logic of a type system for bounded heap consumption where the amount of memory used may depend on the structural size of input data [20]. The encoding involved formulae that express the structured use of a freelist and enforce various disjointness conditions. Heap-represented data structures are required to obey a linear typing regime. The interpretations of the typing rules are formally derived in the theorem prover in such a way that the partitioning of the heap into regions holding particular data structures is performed once, during the derivation of the proof rules. Compared to the verification of application programs using separation logic [31], the verification using the derived proof rules proceeds at a higher-level, for the price of being limited to programs originating from high-level code that obeys a particular typing discipline. Compared to the FPCC approach of formalising type systems [4], the explicit use of a program logic introduces a useful abstraction barrier. Proof patterns arising

repeatedly in the verification of program analyses (e.g. the verification of recursive program structures) can be dealt with once-and-for-all. Thus, the program logic may serve as a formalism in which different program analyses may be compared and integrated.

In contrast to our approach of interpreting typing calculi, Benton's logic [11] includes (basic) type information in judgements, extending bytecode verification conditions. Consequently, methods can be given more modular specifications that, for example, constrain the heap to the segment relevant for the verification of the method body, similar to separation logic [31]. In our approach, such local-reasoning principles would be formulated in the interpretation of type judgements, i.e. in derived proof rules [13]. As a further difference, Benton's logic is interpreted extensionally, by reference to program contexts. This enables Benton to prove that certain program transformations are semantics-preserving (see also [10]), while we primarily aim to certify intensional properties such as the consumption of resources [6].

A further approach to integrating types and program logics is proposed by Nanevski and Morrisett [27]. Following a two-level approach that separates effectful from pure computations, Hoare-triples describing side-effecting computations are injected into the type system using a monadic type constructor. The result is a rich, dependently typed reasoning framework whose operational soundness has been established using progress- and preservations lemmas. An extension that treats polymorphism and supports local reasoning using constructs from separation logic appears in [2].

As was mentioned in the introduction, our logic has already been extended to a substantial fragment of the JVML. The basis of this extension is the Bicolano formalisation of the JVML [30]. In connection with this effort, Benjamin Gregoire recently proposed a variation of our soundness proof that eliminates the auxiliary notion of step-indexed validity. Based on his observation, a new formalisation has been produced using the Coq theorem prover. In addition, work is currently under way to include further specification idioms, in particular ghost items and modifies-clauses, by translating them into the format proposed in this paper. It is planned to extend the logic towards multi-threaded programs. For this, we expect the form of invariants presented in the present paper to be particularly useful. Over time, we thus expect that the presented formalism will yield a solid foundation for the certification of functional and non-functional code properties.

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 3.1 for the numerous discussions on JML and program logics, and on formalising these in theorem provers, and to the referees for the valuable feedback they provided.

References

1. E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 331(2-3):251–290, 2005.
2. L. B. Aleksandar Nanevski, Greg Morrisett. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*. ACM Press, Sept. 2006. To appear.

3. A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.
4. A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS), Proceedings*. IEEE Computer Society, 2001.
5. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs'04. Proceedings*, volume 3223 of *LNCS*, pages 34–49. Springer, 2004.
6. D. Aspinall, L. Beringer, and A. Momigliano. Optimisation validation. In J. Knoop, G. C. Necula, and W. Zimmermann, editors, *Proceedings of the 5th International Workshop on Compiler Optimization Meets Compiler Verification (COCV'06)*, ENTCS. Elsevier, 2006. To appear.
7. F. Y. Bannwart and P. Müller. A logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141(1) of *ENTCS*, pages 255–273. Elsevier, 2005.
8. G. Barthe. Mobius – Mobility, Ubiquity and Security. http://mobius.inria.fr.
9. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning (IJCAR'01)*, volume 2083 of *LNCS*, pages 626–641. Springer, 2001.
10. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM Symposium on Principles of Programming Languages, POPL'04, Venice, Italy*, pages 14–25. ACM, 2004.
11. N. Benton. A typed, compositional logic for a stack-based abstract machine. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS'05. Proceedings*, volume 3780 of *LNCS*, pages 364–380. Springer, 2005.
12. L. Beringer and M. Hofmann. A bytecode logic for JML and types – Isabelle/HOL sources. `http://www.tcs.ifi.lmu.de/~beringer/BytecodeLogic.tar.gz`, 2006.
13. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR'04, Montevideo, Uruguay. Proceedings*, volume 3452 of *LNCS*, pages 347–362. Springer, 2004.
14. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
15. 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 *LNCS*, pages 91–106. Springer, 2005.
16. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4), Oct. 1991.
17. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.
18. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM Conference on Programming language design and implementation*, pages 237–247. ACM Press, 1993.
19. R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

20. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03: Proceedings of the 30th ACM Symposium on Principles of programming languages*, pages 185–197. ACM Press, 2003.
21. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE'01. Proceedings*, volume 2029 of *LNCS*, pages 284–299. Springer, 2001.
22. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.
23. R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.
24. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.
25. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML Reference Manual (draft). http://www.cs.iastate.edu/ leavens/JML, May 2006.
26. K. R. M. Leino and R. Stata. Checking object invariants. Technical Report #1997-007, Digital Equipment Corporation Systems Research Center, Palo Alto, USA, 1997.
27. A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.
28. G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM Symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.
29. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. C. Bradfield, editor, *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL. Proceedings*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.
30. D. Pichardie. Bicolano – Byte Code Language in Coq. http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html, 2006.
31. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS'02). Proceedings*, pages 55–74. IEEE Computer Society, 2002.
32. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, July 1999.