A Program Logic for Bytecode

Fabian Bannwart [1] and Peter Müller [2,3]

ETH Zürich, CH-8092 Zürich, Switzerland

Abstract

Program logics for bytecode languages such as Java bytecode or the .NET CIL can be used to apply Proof-Carrying Code concepts to bytecode programs and to verify correctness properties of bytecode programs. This paper presents a Hoare-style logic for a sequential bytecode kernel language similar to Java bytecode and CIL. The logic handles object-oriented features such as inheritance, dynamic method binding, and object structures with destructive updates, as well as unstructured control flow with jumps. It is sound and complete.

Key words: Java Bytecode, .NET CIL, program verification, Hoare logic

1 Introduction

Intermediate languages such as Java bytecode and the .NET CIL are part of standardized execution environments that are independent of a particular hardware, operating system, or source programming language. Therefore, they support platform-independence and language interoperability.

Although programs are usually developed in a source language and then compiled to an intermediate language (bytecode), several applications require that formal reasoning is applied on the bytecode level rather than the source level: (1) Software for small devices is often developed directly in an intermediate language without using a source language. The typically high correctness and security requirements of such software can be met by formal verification, applied on the bytecode level. (2) Proof-Carrying Code [15] embeds formal proofs of program properties into compiled code such as bytecode. Code consumers can check these proofs before executing code from untrusted sources. (3) Proofs about bytecode programs can be used to improve and speed up JIT compilation [21].

[1] `fybannwart@student.ethz.ch`

[2] `peter.mueller@inf.ethz.ch`

[3] This work was supported by ETH Research Grant TH -26/04-2

Formal verification of bytecode programs requires a program logic for bytecode. This paper presents a Hoare-style program logic for a kernel bytecode language. The logic supports the typical object-oriented features such as classes and objects, inheritance, instance fields, instance methods and dynamic method binding, as well as unstructured control flow with conditional and unconditional jumps. For brevity, we omit static class members, exception handling, class initialization, and value classes in this paper. However, our logic covers these features [4]. An extension of our logic to full Java bytecode or .NET CIL is straightforward.

Approach. The logic presented in this paper has been developed within a project that aims at generating verified bytecode automatically from verified source programs. That is, we aim at developing a so-called proof-transforming compiler, which translates a source program and a proof of certain properties of the source program to the bytecode level [3]. Proof-transforming compilers are similar to certifying compilers in Proof-Carrying Code [8], but take a source proof as input. To simplify the proof translation, our bytecode logic resembles Poetzsch-Heffter and Müller's source code logic [19]: both logics are based on the same model of the object store, handle inheritance, dynamic method binding, and recursion in the same way, and use the same language-independent rules (for instance, the rule of consequence). Therefore, proofs for corresponding source and bytecode programs have a similar proof structure and are based on identical proof obligations in first-order logic (for instance, for the rule of consequence).

For the bytecode instructions, we adapt program logics for programs with unstructured control flow [5]. Instead of using triples like in classic Hoare logic, each instruction I is preceded by an assertion that gives all properties that must hold at that point in the code for being able to verify the given method body as a whole. This precondition has to be established by all predecessors of I, which usually includes the instruction that precedes I in the program text as well as all instructions that jump to I.

Our logic assumes that the bytecode program is well-formed, in particular, well-typed. That is, we consider programs that are accepted by the bytecode verifier.

Outline. We introduce the bytecode kernel language and its operational semantics in Sec. 2. The program logic is presented in Sec. 3. We sketch the soundness proof in Sec. 4. In Sec. 5, we show how our logic can be applied in a wp-fashion and illustrate how source proofs can be translated to the bytecode logic. Related work is discussed in Sec. 6.

2 The Bytecode Language $\mathrm{VM_K}$

In this section, we present the bytecode kernel language, $\mathrm{VM_K}$, and its operational semantics.

2.1 VM$_K$ Programs

As in Java or .NET, a VM$_K$ program consists of classes with fields and methods. The methods are implemented as method bodies consisting of a sequence of labeled bytecode instructions. The bytecode instructions operate on an evaluation stack (sometimes called operand stack), local variables (which also include parameters), and the object store (heap). The instructions of VM$_K$ are explained along with their operational semantics below.

We make some assumptions in order to keep the formalism simple: methods are always virtual, return a value, and take two parameters: the receiver, `this`, and one explicit parameter, `p`. Each method body ends with a `return` instruction, which returns the control flow to the caller. This instruction can occur only as the last instruction of a method body. A method returns the value stored in the special local variable `result`.

In this paper, we omit static class members, exceptions, class initialization, and value classes. An extension of the logic to these features and several instructions not discussed here is presented in our technical report [4].

VM$_K$ is very similar to Java bytecode and .NET CIL. However, it does not support CIL's structured exception handling and Java's method-local subroutines that are used to compile `finally` clauses. These features can be handled by code expansion [23]. Moreover, VM$_K$ does not support CIL's class modifier `.beforefieldinit`, which indicates that a class can be initialized any time before the access of static fields (that is, not necessarily immediately before the first use of a class). This behavior is difficult to model in program logics.

2.2 The Object Store

Source and bytecode programs support the same operations on the object store. Therefore, we build on an existing formal model of the object store [18], which we briefly summarize here.

The state of all objects and the information whether an object is allocated in the current program state is formalized by an abstract data type with sort *ObjectStore* and the following functions:

$$\begin{array}{ll}
\mathrm{iv}(v,f): & Value \times FieldId \rightarrow InstVar \\
OS\langle a := v\rangle : & ObjectStore \times InstVar \times Value \rightarrow ObjectStore \\
OS(f): & ObjectStore \times InstVar \rightarrow Value \\
OS\langle T\rangle : & ObjectStore \times ClassTypeId \rightarrow ObjectStore \\
\mathrm{new}(OS,T): & ObjectStore \times ClassTypeId \rightarrow Value
\end{array}$$

A $Value$ is a value of a primitive type or a reference. $FieldId$ and $ClassTypeId$ are unique identifiers of fields and classes, resp. $InstVar$ is the set of field addresses of all objects in the program. $\mathrm{iv}(v,f)$ yields the address of a field identified by f from object v. $OS\langle a := v\rangle$ returns the object store where the instance variable a is updated with the new value v. $OS\langle T\rangle$ yields the store where a new object of type T is allocated. new(OS, T) returns a fresh object of type T in OS. For an axiomatization of these functions see [18].

To have a uniform treatment for variables and the object store in the formal semantics, we use \$ as identifier for the current object store.

2.3 Operational Semantics

In this subsection, we present an operational semantics for $\mathrm{VM_K}$.

Configurations. A configuration $\langle S, \sigma, l\rangle$ of a method execution consists of a state, S, an evaluation stack, σ, and the program counter, l, which is the label of the next instruction to be executed. The state maps identifiers for local variables (sort $VarId$), formal parameters, and the current object store to values. The evaluation stack is a sequence of values.

$$\begin{aligned} &State \equiv (VarId \cup \{\, \texttt{this}, \texttt{p}\,\} \to Value \cup \{undef\}) \times (\{\,\$\,\} \to ObjectStore) \\ &Stack \equiv Value^{*} \end{aligned}$$

For $S \in State$, we write $S(\texttt{x})$ for the application to a variable or parameter identifier and $S(\$)$ for the application to the object store. The sequence $(\sigma, e_1, e_2, \ldots)$ is the sequence obtained from σ by appending e_1, then e_2, etc.

l is a valid label, that is, it is in set of labels $\{0, \ldots, |\mathtt{p}| - 1\}$ of a method body $\mathtt{p}$. $|\mathtt{p}|$ denotes the number of instructions in $\mathtt{p}$. $\mathtt{p}(l)$ is the instruction at label l in $\mathtt{p}$. When the method body $\mathtt{p}$ is clear from the context, we simply write I_l for the instruction at label l.

Instruction Semantics. The transition relation $\mathtt{p}; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle$ expresses that the execution of the instruction I_l in the method body $\mathtt{p}$ brings the machine from configuration $\langle S, \sigma, l\rangle$ to $\langle S', \sigma', l'\rangle$. For a given method body $\mathtt{p}$, the multi-step relation $\to^{*}$ is the reflexive transitive closure of $\to$.

The transition relation is the smallest relation satisfying the rules in Fig. 1. The instructions `pushc` and `pushv` push constants and variables onto the evaluation stack, resp. That is, they leave the state unchanged, add a new value to the stack, and increment the program counter. `pop` pops a value from the evaluation stack and assigns it to a variable. We summarize all binary operators such as boolean and arithmetic operators by an instruction $\texttt{binop}_{\mathrm{op}}$, which pops two values from the stack, performs the binary operation, and pushes the result. Conditional and unconditional jumps are expressed by `brtrue` and `goto` , resp. `newobj` T creates a new object of class T, thereby modifying the current object store. A reference to the new object is pushed onto the stack. The `checkcast` T instruction performs the runtime check for a cast. If the object v referenced from the top of the stack is an instance of T, the program counter is incremented. Otherwise, the execution halts. In the rule for `checkcast`, $\tau(v)$ is the (dynamic) type of value v and $\preceq$ denotes the subtype relation. `getfield` and `putfield` read and update instance fields. Both instructions pop the receiver object, y. If y is $null$, the execution

$$[\ldots\ l : \texttt{pushc}\ v\ \ldots]; \langle S, \sigma, l\rangle \rightarrow \langle S, (\sigma, v), l+1\rangle$$
$$[\ldots\ l : \texttt{pushv}\ x\ \ldots]; \langle S, \sigma, l\rangle \rightarrow \langle S, (\sigma, S(x)), l+1\rangle$$
$$[\ldots\ l : \texttt{pop}\ x\ \ldots]; \langle S, (\sigma, v), l\rangle \rightarrow \langle S[x \mapsto v], \sigma, l+1\rangle$$
$$[\ldots\ l : \texttt{binop}_{\mathrm{op}}\ \ldots]; \langle S, (\sigma, v_1, v_2), l\rangle \rightarrow \langle S, (\sigma, v_1\,\mathrm{op}\,v_2), l+1\rangle$$
$$[\ldots\ l : \texttt{brtrue}\ l'\ \ldots]; \langle S, (\sigma, \mathit{true}), l\rangle \rightarrow \langle S, \sigma, l'\rangle$$
$$[\ldots\ l : \texttt{brtrue}\ l'\ \ldots]; \langle S, (\sigma, \mathit{false}), l\rangle \rightarrow \langle S, \sigma, l+1\rangle$$
$$[\ldots\ l : \texttt{goto}\ l'\ \ldots]; \langle S, \sigma, l\rangle \rightarrow \langle S, \sigma, l'\rangle$$
$$[\ldots\ l : \texttt{newobj}\ T\ \ldots]; \langle S, \sigma, l\rangle \rightarrow \langle S[\$ \mapsto S(\$)\langle T\rangle], (\sigma, \mathrm{new}(S(\$), T)), l+1\rangle$$

$$\frac{\tau(v) \preceq T}{[\ldots\ l : \texttt{checkcast}\,T\ \ldots]; \langle S, (\sigma, v), l\rangle \rightarrow \langle S, (\sigma, v), l+1\rangle}$$

$$\frac{y \neq \texttt{null}}{[\ldots\ l : \texttt{getfield}\ T@a\ \ldots]; \langle S, (\sigma, y), l\rangle \rightarrow \langle S, (\sigma, S(\$)(\mathrm{iv}(y, T@a))), l+1\rangle}$$

$$\frac{\begin{array}{c} y \neq \texttt{null} \\ S_p = S[\$ \mapsto S(\$)\langle \mathrm{iv}(y, T@a) := v\rangle] \end{array}}{[\ldots\ l : \texttt{putfield}\ T@a\ \ldots]; \langle S, (\sigma, y, v), l\rangle \rightarrow \langle S_p, \sigma, l+1\rangle}$$

$$\frac{\begin{array}{c} y \neq \texttt{null} \\ \mathfrak{p}' = \mathit{body}(\mathit{impl}(\tau(y), m)) \qquad \mathfrak{p}'(l') = \texttt{return} \\ \mathfrak{p}'; \langle \{\texttt{this} \mapsto y, \texttt{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle \rightarrow^* \langle S', \sigma', l'\rangle \\ S_p = S[\$ \mapsto S'(\$)] \qquad \sigma_p = (\sigma, S'(\texttt{result})) \end{array}}{[\ldots\ l : \texttt{invokevirtual}\ T{:}m\ \ldots]; \langle S, (\sigma, y, v), l\rangle \rightarrow \langle S_p, \sigma_p, l+1\rangle}$$

Fig. 1. Rules of the operational semantics.

halts. Otherwise, `getfield` pushes the value of the instance variable onto the stack. `putfield` pops a second value and updates the instance variable with that value, that is, modifies the object store. Field identifiers are written as $\mathit{Type}@\mathit{fieldname}$.

The most complex rule handles invocations of virtual methods. We assume that method calls are augmented by the static type of their receiver expression. For instance, a method m invoked on an expression of static type T is denoted by $T{:}m$. The implementation of a method $T{:}m$ in class S is denoted by $\mathit{impl}(S, T{:}m)$ or simply by $\mathit{impl}(S, m)$. Note that S can inherit m from a superclass. The body of a method m is denoted by $\mathit{body}(m)$. `invokevirtual` $T{:}m$ pops the receiver object, y, and the actual parameter value, v. Each method execution has its own evaluation stack, which is destroyed when its method invocation completes. Therefore, the body of the dynamically-bound method m, $\mathfrak{p}'$, is executed in a configuration with an empty stack and the actual arguments assigned to the formal parameters. The exeuction of $\mathfrak{p}'$ terminates when it reaches its last instruction, `return`. Control returns to the caller after the value of `result` is pushed onto the stack.

3 Program Logic

The Hoare-style program logic presented in this section allows one to formally verify that implementations satisfy interface specifications given as pre- and postconditions.

3.1 Method and Instruction Specifications

Our treatment of methods follows Poetzsch-Heffter and Müller's program logic for Java source programs [19]: We distinguish between method implementations and virtual methods. A *method implementation* $T@m$ represents the concrete implementation of method m in class T. A *virtual method* $T{:}m$ represents the common properties of all method implementations that might by invoked dynamically when m is called on a receiver of static type T, that is, $impl(T, m)$ (if $T{:}m$ is not abstract) and all overriding subclass methods.

Method Specifications. Properties of methods and method bodies are expressed by Hoare triples of the form $\{P\}$ comp $\{Q\}$, where P, Q are sorted first-order formulas and comp is a method implementation $T@m$, a virtual method $T{:}m$, or a method body $\mathtt{p}$. We call such a triple *method specification.* The triple $\{P\}$ comp $\{Q\}$ expresses the following refined partial correctness property: if the execution of comp starts in a state satisfying P, then (1) comp terminates in a state in which Q holds, or (2) comp aborts due to errors or actions that are beyond the semantics of the programming language (for instance, memory allocation problems), or (3) comp runs forever.

The pre- and postconditions of method specifications must not refer to variables or stack elements. Preconditions may refer to the formal parameters `this` and `p`, as well as the current object store $\$$. Postconditions may refer to $\$$ and `result`.

For the treatment of recursive methods, we use sequents of the form $\mathcal{A} \vdash \{P\}$ comp $\{Q\}$ where $\mathcal{A}$ is a set of method specifications. Intuitively, such a sequent expresses the fact that the triple $\{P\}$ comp $\{Q\}$ can be proved based on some assumptions $\mathcal{A}$ about methods (see [19] for details).

Instruction Specifications. The unstructured control flow of bytecode programs makes it difficult to handle instruction sequences, because jumps can transfer control into and from the middle of a sequence. Therefore, our logic treats each instruction individually: each individual instructions I_l in a method body $\mathtt{p}$ has a precondition E_l. An instruction with its precondition is called an *instruction specification*, written as $\{E_l\}\, l : I_l$.

Obviously, the meaning of an instruction specification $\{E_l\}\, l : I_l$ cannot be defined in isolation. $\{E_l\}\, l : I_l$ expresses that if the precondition E_l holds when the program counter is at position l, then the precondition $E_{l'}$ of I_l's successor instruction I'_l holds after normal termination of I_l.

Like method specifications, instruction specifications can have assumptions. An instruction specification with assumption set $\mathcal{A}$ is denoted by $\mathcal{A} \vdash \{E_l\}\, l : I_l$.

Connecting Instruction and Method Specifications. Individual instructions can be combined at the level of method bodies since $\mathrm{VM_K}$ guarantees that the instruction sequence constituting a method body is always entered at the first instruction and left after the last instruction. All jumps are local within a method body. The precondition of a method implementation is the precondition of the first instruction of its body, the method postcondition is the precondition of the `return` instruction. Consequently, a method implementation $T@m$ satisfies its method specification if all instructions in the body of $T@m$ satisfy their instruction specifications. This connection is formalized by the body rule:

$$\frac{\forall i \in \{0, \ldots, |body(T@m)| - 1\} : (\mathcal{A} \vdash \{E_i\}\, i : I_i)}{\mathcal{A} \vdash \{E_0\}\ \ body(T@m)\ \ \left\{E_{|body(T@m)|-1}\right\}}$$

$\{E_0\}\ \ body(T@m)\ \ \left\{E_{|body(T@m)|-1}\right\}$ has to be an admissible method specification, in particular, E_0 and $E_{|body(T@m)|-1}$ must not refer to local variables.

3.2 Rules for Instruction Specifications

All rules for $\mathrm{VM_K}$ instructions, except for method calls, have the following form:

$$\frac{E_l \Rightarrow \mathrm{wp}^1_{\mathtt{p}}(I_l)}{\mathcal{A} \vdash \{E_l\}\, l : I_l}$$

$\mathrm{wp}^1_{\mathtt{p}}(I_l)$ is the *local weakest precondition* of instruction I_l. Such a rule expresses that the precondition of I_l has to imply the weakest precondition of I_l w.r.t. all possible successor instructions of I_l.

The definition of $\mathrm{wp}^1_{\mathtt{p}}$ is shown in Fig. 2. Within an assertion, the current stack is referred to as s, and its elements are denoted by non-negative integers: element 0 is the top element, etc. The interpretation $[\![E_l]\!] : State \times Stack \rightarrow Value$ for s is $[\![s(0)]\!]\langle S, (\sigma, v)\rangle = v$ and $[\![s(i+1)]\!]\langle S, (\sigma, v)\rangle = [\![s(i)]\!]\langle S, \sigma\rangle$. The functions *shift* and *unshift* express the substitutions that occur when values are pushed onto and popped from the stack, resp.:

$$\begin{aligned} shift(E) &= E[s(i+1)/s(i) \text{ for all } i \in \mathbb{N}] \\ unshift &= shift^{-1} \end{aligned}$$

$shift^n$ denotes n consecutive applications of *shift*.

The rules for `pushc` , `pushv` , and `pop` are analogous to Hoare's assignment axiom: The precondition is obtained from the postcondition by substituting the right-hand side of the assignment for the left-hand side variable. For the push instructions, the top stack element can be regarded as the left-hand side variable; for `pop` the stack top is the right-hand side expression. All other

I_l	$\mathrm{wp}^1_{\mathtt{p}}(I_l)$
`pushc` v	$\mathit{unshift}(E_{l+1}[v/s(0)])$
`pushv` x	$\mathit{unshift}(E_{l+1}[x/s(0)])$
`pop` x	$(\mathit{shift}(E_{l+1}))[s(0)/x]$
$\mathtt{binop}_{\mathrm{op}}$	$(\mathit{shift}(E_{l+1}))[(s(1)\,\mathrm{op}\,s(0))/s(1)]$
`goto` l'	$E_{l'}$
`brtrue` l'	$(\neg s(0) \Rightarrow \mathit{shift}(E_{l+1})) \wedge (s(0) \Rightarrow \mathit{shift}(E_{l'}))$
`checkcast` T	$E_{l+1} \wedge \tau(s(0)) \preceq T$
`newobj` T	$\mathit{unshift}(E_{l+1}[\mathrm{new}(\$, T)/s(0), \$\langle T\rangle/\,\$])$
`getfield` $T@a$	$E_{l+1}[\$(\mathrm{iv}(s(0), T@a))/s(0)] \wedge s(0) \neq \mathtt{null}$
`putfield` $T@a$	$(\mathit{shift}^2(E_{l+1}))[\$\langle \mathrm{iv}(s(1), T@a) := s(0)\rangle/\,\$] \wedge s(1) \neq \mathtt{null}$
`return`	true

Fig. 2. The values of the $\mathrm{wp}^1_{\mathtt{p}}$ function. Except for `brtrue` , all instructions have only one potential successor.

stack references are adapted by applying the *unshift* and *shift* function, resp.

The `binop` instruction pops two values, performs a binary operation, and pushes the result. Therefore, *shift* is applied only once. An unconditional jump changes the control flow. Therefore, its local weakest precondition is the precondition of the jump target. A branch has two possible successors, depending on the value of the stack top. Its local weakest precondition is obtained from the preconditions of both potential successor instructions.

For a `checkcast` T instruction, one has to show that the precondition of its successor holds and that the type of the stack top is a subtype of T. Since the top stack element is not popped, *shift* is not applied here. Object creation, field read, and field update are also similar to classical assignment: `putfield` updates the current object store, `getfield` updates the top stack element, and `newobj` updates both. `getfield` and `putfield` require that the receiver object (the stack top) is non-null. `getfield` substitutes the value held by the designated instance variable for the stack top. Since it pops and pushes one element each, *shift* is not applied. `putfield` updates the current object store at the designated instance variable with the second stack element. Since it pops two values, *shift* is applied twice.

Method Calls. For the call of a virtual method *T:m*, one has to prove (1) that *T:m* satisfies its method specification, (2) that the precondition of the `invokevirtual` instruction implies the precondition of the method specification, with actual arguments substituted for the formal parameters, and

(3) that the postcondition of the method specification implies the precondition of the instruction following `invokevirtual` , with `result` substituted by the stack top. These requirements are the antecedents of the rule for `invokevirtual` :

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{P\}\ \ T{:}m\ \ \{Q\} \\ E_l \Rightarrow s(1) \neq \mathtt{null} \wedge P[s(1)/\,\mathtt{this}, s(0)/\,\mathtt{p}][\mathit{shift}(\mathtt{w})/Z] \\ Q[s(0)/\,\mathtt{result}][\mathtt{w}/Z] \Rightarrow E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{invokevirtual}\ \ T{:}m}$$

where Z is a vector $Z_0, \ldots, Z_n$ of logical variables and $\mathtt{w}$ is a vector $\mathtt{w}_0, \ldots, \mathtt{w}_n$ of local variables or stack elements (different from $s(0)$). The *shift* function for vectors is defined pointwise.

A method call does not modify the local variables and the evaluation stack of the caller, except for popping the arguments and pushing the result of the call. To express these frame properties, the invocation rule allows one to substitute logical variables in the method's pre- and postcondition by local variables and stack elements of the caller. However, $s(0)$ must not be used for a substitution because it contains the result of the call, that is, its value is not preserved by the call.

3.3 Rules for Method Specifications

The rules for method specifications are identical to Poetzsch-Heffter and Müller's source program logic. We summarize these rules briefly here. For a detailed explanation, see [19].

Virtual methods are used to model dynamically-bound methods. That is, a method specification for $T{:}m$ reflects the common properties of all implementations that might be executed on invocation of $T{:}m$. If T is a class, there are two obligations to prove a specification of a virtual method $T{:}m$: (1) Show that the corresponding implementation satisfies the specification if invoked for objects of type T. (2) Show that the specification holds for objects of proper subtypes of T.

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{P \wedge \tau(\mathtt{this}) = T\}\ \ \mathit{impl}(T, m)\ \ \{Q\} \\ \mathcal{A} \vdash \{P \wedge \tau(\mathtt{this}) \prec T\}\ \ T{:}m\ \ \{Q\}\end{array}}{\mathcal{A} \vdash \{P \wedge \tau(\mathtt{this}) \preceq T\}\ \ T{:}m\ \ \{Q\}}$$

The second antecedent of this rule and annotations of interface type methods can be proved by the following rule: If S is a subtype of T, an invocation of $T{:}m$ on an S object is equivalent to an invocation of $S{:}m$. Thus, all properties of $S{:}m$ carry over to $T{:}m$ as long as $T{:}m$ is applied to S objects:

$$\frac{}{\vdash \{\mathit{false}\}\ \mathrm{comp}\ \{\mathit{false}\}} \qquad \frac{}{\{P\}\ \mathrm{comp}\ \{Q\} \vdash \{P\}\ \mathrm{comp}\ \{Q\}}$$

$$\frac{\mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}}{\{P'\}\ \mathrm{comp}'\ \{Q'\}, \mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}} \qquad \frac{\begin{array}{c}\mathcal{A} \vdash \{P'\}\ \mathrm{comp}'\ \{Q'\} \\ \{P'\}\ \mathrm{comp}'\ \{Q'\}, \mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}\end{array}}{\mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}}$$

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{P_1\}\ \mathrm{comp}\ \{Q_1\} \\ \mathcal{A} \vdash \{P_2\}\ \mathrm{comp}\ \{Q_2\}\end{array}}{\mathcal{A} \vdash \{P_1 \wedge P_2\}\ \mathrm{comp}\ \{Q_1 \wedge Q_2\}} \qquad \frac{\begin{array}{c}\mathcal{A} \vdash \{P_1\}\ \mathrm{comp}\ \{Q_1\} \\ \mathcal{A} \vdash \{P_2\}\ \mathrm{comp}\ \{Q_2\}\end{array}}{\mathcal{A} \vdash \{P_1 \vee P_2\}\ \mathrm{comp}\ \{Q_1 \vee Q_2\}}$$

$$\frac{P \Rightarrow P' \qquad \mathcal{A} \vdash \{P'\}\ \mathrm{comp}\ \{Q'\} \qquad Q' \Rightarrow Q}{\mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}}$$

$$\frac{\mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}}{\mathcal{A} \vdash \{P \wedge R\}\ \mathrm{comp}\ \{Q \wedge R\}} \qquad \frac{\mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q\}}{\mathcal{A} \vdash \{P[t/Z]\}\ \mathrm{comp}\ \{Q[t/Z]\}}$$

$$\frac{\mathcal{A} \vdash \{P[Y/Z]\}\ \mathrm{comp}\ \{Q\}}{\mathcal{A} \vdash \{P[Y/Z]\}\ \mathrm{comp}\ \{\forall Z : Q\}} \qquad \frac{\mathcal{A} \vdash \{P\}\ \mathrm{comp}\ \{Q[Y/Z]\}}{\mathcal{A} \vdash \{\exists Z : P\}\ \mathrm{comp}\ \{Q[Y/Z]\}}$$

Fig. 3. Language-independent rules. R and t are terms that do not reference program variables. Y and Z are distinct logical variables.

$$\frac{\begin{array}{c}S \preceq T \\ \mathcal{A} \vdash \{P \wedge \tau(\mathtt{this}) \preceq S\}\ \ S{:}m\ \{Q\}\end{array}}{\mathcal{A} \vdash \{P \wedge \tau(\mathtt{this}) \preceq S\}\ \ T{:}m\ \{Q\}}$$

Finally, a specification of a method implementation $T@m$ holds if it holds for its body. To handle recursion, the specification of $T@m$ may be assumed for the proof of the body.

$$\frac{\mathcal{A}, \{P\}\ T@m\ \{Q\} \vdash \{P \wedge \mathtt{this} \neq \mathtt{null}\}\ \mathit{body}(T@m)\ \{Q\}}{\mathcal{A} \vdash \{P\}\ T@m\ \{Q\}}$$

Besides the axiomatic semantics, the programming logic for $\mathrm{VM_K}$ contains language-independent axioms and rules to handle assumptions and to establish a connection between the predicate logic of pre- and postconditions and triples of the programming logic (Fig. 3). These rules can be applied to method specifications.

3.4 Example

To illustrate how our logic works, we verify a method `int abs(int p)` that returns the absolute value of its argument. For simplicity, we assume that `abs` is declared in a class `Math` that does not have any subclasses. We prove that the method satisfies the following specification:

$$\{\texttt{p} = P\}\ \mathit{Math{:}abs}\ \{(P \geq 0 \Rightarrow \texttt{result} = P) \wedge (P < 0 \Rightarrow \texttt{result} = -P)\}$$

The logical variable P is used to refer to `p`'s initial value from the postcondition. It is necessary to meet the syntactic restrictions of method specifications that formal parameters must not occur in postconditions (Sec. 3.1). To derive this triple, we first derive the instruction specifications for `abs`' body (we omit assumptions for brevity):

$$\begin{array}{rl}
\{\texttt{p} = P \wedge \tau(\texttt{this}) = \texttt{Math} \wedge \texttt{this} \neq \texttt{null}\} & 0: \texttt{pushv p} \\
\{(s(0) < 0 \Rightarrow P < 0) \wedge (s(0) \geq 0 \Rightarrow P \geq 0) \wedge \texttt{p} = P\} & 1: \texttt{pushc 0} \\
\{(s(1) < s(0) \Rightarrow P < 0) \wedge (s(1) \geq s(0) \Rightarrow P \geq 0) \wedge \texttt{p} = P\} & 2: \texttt{binop}_{\geq} \\
\{(s(0) < 0 \Rightarrow P < 0) \wedge (s(0) \geq 0 \Rightarrow P \geq 0) \wedge \texttt{p} = P\} & 3: \texttt{brtrue 8} \\
\{P < 0 \wedge \texttt{p} = P\} & 4: \texttt{pushc 0} \\
\{P < 0 \wedge s(0) - \texttt{p} = -P\} & 5: \texttt{pushv p} \\
\{P < 0 \wedge s(1) - s(0) = -P\} & 6: \texttt{binop}_{-} \\
\{P < 0 \wedge s(0) = -P\} & 7: \texttt{goto 9} \\
\{P \geq 0 \wedge \texttt{p} = P\} & 8: \texttt{pushv p} \\
\{(P \geq 0 \Rightarrow s(0) = P) \wedge (P < 0 \Rightarrow s(0) = -P)\} & 9: \texttt{pop result} \\
\{(P \geq 0 \Rightarrow \texttt{result} = P) \wedge (P < 0 \Rightarrow \texttt{result} = -P)\} & 10: \texttt{return}
\end{array}$$

One can easily see, that the precondition of each instruction implies the local weakest precondition. For instance, the precondition $P \geq 0 \wedge \texttt{p} = P$ of instruction $8:$ `pushv p` implies the local weakest precondition, $(P \geq 0 \Rightarrow p = P) \wedge (P < 0 \Rightarrow p = -P)$.

By the body rule, we combine these instruction specifications to the method specification of `abs`' body, and then derive the specification of $Math@abs$ (we abbreviate $(P \geq 0 \Rightarrow \texttt{result} = P) \wedge (P < 0 \Rightarrow \texttt{result} = -P)$ by Q):

$$\frac{\{\texttt{p} = P \wedge \tau(\texttt{this}) = \texttt{Math} \wedge \texttt{this} \neq \texttt{null}\}\ \mathit{body}(Math@abs)\ \{Q\}}{\{\texttt{p} = P \wedge \tau(\texttt{this}) = \texttt{Math}\}\ Math@abs\ \{Q\}}$$

Since `Math` does not have subclasses, we have $\tau(\texttt{this}) \prec \texttt{Math} \Rightarrow false$. Therefore, we can derive by the rule of consequence:

$$\frac{\{false\}\ \mathit{Math{:}abs}\ \{false\}}{\{\texttt{p} = P \wedge \tau(\texttt{this}) \prec \texttt{Math}\}\ \mathit{Math{:}abs}\ \{Q\}}$$

Since `abs` is implemented in class `Math`, we have $\mathit{impl}(\texttt{Math}, \texttt{abs}) = Math@abs$. Therefore, we can conclude the proof by combining the above two triples:

$$\frac{\begin{array}{c}\{\texttt{p} = P \wedge \tau(\texttt{this}) = \texttt{Math}\}\ Math@abs\ \{Q\} \\ \{\texttt{p} = P \wedge \tau(\texttt{this}) \prec \texttt{Math}\}\ \mathit{Math{:}abs}\ \{Q\}\end{array}}{\{\texttt{p} = P\}\ \mathit{Math{:}abs}\ \{Q\}}$$

4 Soundness

Our logic is sound with respect to the operational semantics. In this section, we sketch the soundness proof. The complete proof is presented in our technical report [4], which also contains the completeness proof.

We express soundness on the level of method specifications: if a method specification $\{P\}\ M\ \{Q\}$ can be proved, then it actually holds. Following Gordon [10], we embed both the operational and the axiomatic semantics into higher order logic (see [19] for details). For the operational semantics, *sem* denotes the multistep relation: $\mathit{sem}(C, \mathfrak{p}, C') \equiv \mathfrak{p}; C \to^* C'$. The fact that the triple $\{P\}\ M\ \{Q\}$ holds is formalized as predicate $H(P, M, Q)$, which is defined as follows:

$$\begin{aligned}
H(P, \mathfrak{p}, Q) \equiv\ & \forall (C \equiv \langle \{\texttt{this} \mapsto \texttt{this}_0, \texttt{p} \mapsto \texttt{p}_0, \$ \mapsto \$_0\}, (), 0\rangle), \\
& (C' \equiv \langle S', \sigma', l'\rangle) : \\
& \qquad \mathit{sem}(C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{return} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C' \\
H(P, T@m, Q) \equiv\ & H(\texttt{this} \neq \texttt{null} \wedge P, \mathit{body}(T@m), Q) \\
H(P, T_0 : m, Q) \equiv\ & \forall T \preceq T_0 : H(\tau(\texttt{this}) = T \wedge P, \mathit{impl}(T, m), Q)
\end{aligned}$$

The soundness prove runs by induction on the structure of the derivation tree for a Hoare triple. For a rule with antecedents $\{P_i\}\ M_i\ \{Q_i\}$ and consequent $\{P\}\ M\ \{Q\}$, we prove $(\bigwedge_i H(P_i, M_i, Q_i)) \Rightarrow H(P, M, Q)$. To focus on the specialties of the bytecode logic, we simplified this translation in two ways: (1) we ignore the assumptions of sequents since they are not important for the rules of $\mathrm{VM_K}$ instructions; (2) the translation misses out the inductive argument associated with the treatment of recursive methods. Both aspects are covered by the translation presented in [19].

Since the rules for method specifications in the $\mathrm{VM_K}$ logic, in particular, the language-independent rules, are identical to the rules of our source logic, the proofs for these rules are identical for both logics. We do not repeat these cases here.

The most interesting case is the body rule, which connects individual instructions to a method body (see Sec. 3.1). For this rule, we have to prove $H(E_0, \mathit{body}(T@m), E_{|\mathit{body}(T@m)|-1})$. It is however easier to derive the more general property

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : \mathit{sem}(C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C'$$

which is proved by induction on the length of the derivation of $\mathit{sem}(C, \mathfrak{p}, C')$. For the induction step, we have to consider each individual step of the derivation and prove:

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : (\mathfrak{p}; C \to C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C')$$

We prove this property by case distinction over all possible instructions I_l.

The proofs of these cases rely on the following two substitution lemmas:

Lemma 4.1

$$[\![E]\!]\langle S,\sigma,l\rangle \Longleftrightarrow [\![\mathit{shift}^{|\kappa|}(E)]\!]\langle S,(\sigma,\kappa),l\rangle$$

Lemma 4.2

$$\begin{aligned}&[\![E[s_0/s(i_0),\ldots,s_n/s(i_n),y_0/x_0,\ldots,y_m/x_m]]\!]\langle S,\sigma,l\rangle \Longleftrightarrow\\ &\qquad [\![E]\!]\langle S[x_0\mapsto [\![y_0]\!]\langle S,\sigma,l\rangle,\ldots,x_m\mapsto [\![y_m]\!]\langle S,\sigma,l\rangle],\\ &\qquad\qquad \sigma[i_0\mapsto [\![s_0]\!]\langle S,\sigma,l\rangle,\ldots,i_n\mapsto [\![s_n]\!]\langle S,\sigma,l\rangle],l\rangle\end{aligned}$$

For brevity, we only show one case of the prove: `pushc` . All other cases, except for `invokevirtual` are analogous, see [4].

$$\begin{array}{lll} & [\![E_l]\!]\langle S,\sigma\rangle & \text{– antecedent of the rule}\\ \Rightarrow & [\![\mathrm{wp}^1_{\mathtt{p}}(l:\mathtt{pushc}\ v)]\!]\langle S,\sigma\rangle & \text{– definition of } \mathrm{wp}^1_{\mathtt{p}}\\ \Longleftrightarrow & [\![\mathit{unshift}(E_{l+1}[v/s(0)])]\!]\langle S,\sigma\rangle & \text{– Lemma 4.1}\\ \Longleftrightarrow & [\![E_{l+1}[v/s(0)]]\!]\langle S,(\sigma,t)\rangle & \text{– Lemma 4.2}\\ \Longleftrightarrow & [\![E_{l+1}]\!]\langle S,(\sigma,v)\rangle & \end{array}$$

5 Applying the Logic

To verify a method body, one has to find suitable specifications for each of its instructions. While this task can be cumbersome for programs with complex control flow, the specifications can be derived systematically in many practical cases. In this section, we show by an example that instruction specifications can be derived by weakest precondition transformation. If the source code and a proof for the source program are available, the instructions and their specifications can also be obtained by proof transformation.

5.1 Weakest Preconditions

Except for method calls, the rules for the instructions of $\mathrm{VM_K}$ are formulated in terms of the local weakest precondition, $\mathrm{wp}^1_{\mathtt{p}}(I_l)$. For given preconditions of all possible successors of an instruction I_l, $\mathrm{wp}^1_{\mathtt{p}}(I_l)$ yields the weakest precondition of I_l. For brevity, we ignore method calls in this subsection. An extension to method calls is straightforward, see [4,22].

Fig. 4 shows the body of a method `Math@pow2(int p)` that calculates $2^{\mathtt{p}}$. We assume that the method postcondition, E_{15}, is given by an interface specification. This example illustrates that in programs with loops, the preconditions of several instructions mutually depend on each other: E_{14} depends on E_3, which in turn depends on E_{14}. Therefore, we cannot directly use the local weakest precondition function $\mathrm{wp}^1_{\mathtt{p}}$ to calculate E_{14}.

Following classical wp-calculi [6], we can use fixed-point iteration to resolve such recursive dependencies. This iteration propagates the method postcondi-

$\{\mathtt{p} = P\}$	`0 : pushc 1`
$\{s(0) = 2^{P-\mathtt{p}}\}$	`1 : pop result`
$\{\mathtt{result} = 2^{P-\mathtt{p}}\}$	`2 : goto 11`
$\{\mathtt{result} = 2^{P-\mathtt{p}} \land \mathtt{p} \neq 0\}$	`3 : pushv p`
$\{\mathtt{result} \cdot 2 = 2^{P-(s(0)-1)}\}$	`4 : pushc 1`
$\{\mathtt{result} \cdot 2 = 2^{P-(s(1)-s(0))}\}$	`5 : binop_` $_-$
$\{\mathtt{result} \cdot 2 = 2^{P-s(0)}\}$	`6 : pop p`
$\{\mathtt{result} \cdot 2 = 2^{P-\mathtt{p}}\}$	`7 : pushv result`
$\{s(0) \cdot 2 = 2^{P-\mathtt{p}}\}$	`8 : pushc 2`
$\{s(1) \cdot s(0) = 2^{P-\mathtt{p}}\}$	`9 : binop` $_*$
$\{s(0) = 2^{P-\mathtt{p}}\}$	`10 : pop result`
$\{\mathtt{result} = 2^{P-\mathtt{p}}\}$	`11 : pushv p`
$\{\mathtt{result} = 2^{P-\mathtt{p}} \land (s(0) \neq 0) = (\mathtt{p} \neq 0)\}$	`12 : pushc 0`
$\{\mathtt{result} = 2^{P-\mathtt{p}} \land (s(1) \neq s(0)) = (\mathtt{p} \neq 0)\}$	`13 : binop` $_{\neq}$
$\{\mathtt{result} = 2^{P-\mathtt{p}} \land s(0) = (\mathtt{p} \neq 0)\}$	`14 : brtrue 3`
$\{\mathtt{result} = 2^{P}\}$	`15 : return`

$$\begin{tikzcd}
I_0 \arrow[d] \\
I_1 \arrow[d] \\
I_2 \arrow[ddddddddd, bend left] \\
I_3 \arrow[d] \\
I_4 \arrow[d] \\
I_5 \arrow[d] \\
I_6 \arrow[d] \\
I_7 \arrow[d] \\
I_8 \arrow[d] \\
I_9 \arrow[d] \\
I_{10} \arrow[d] \\
I_{11} \arrow[d] \\
I_{12} \arrow[d] \\
I_{13} \arrow[d] \\
I_{14} \arrow[uuuuuuuuuuu, bend left] \arrow[d] & \text{start} \\
I_{15}
\end{tikzcd}$$

Fig. 4. Bytecode of method `Math@pow2(int p)`. Each instruction specification can be constructed from the successors' specifications.

tion, Q, backwards through the control flow graph until the instruction specifications do not change anymore. The weakest precondition ψ_l of an instruction I_l is defined in infinitary logic. The local weakest precondition $\mathrm{wp}^1_{\mathfrak{p}}(I_l)^{(k)}$ is defined analogously to $\mathrm{wp}^1_{\mathfrak{p}}$, but refers to the computed instruction specifications $\psi^{(k)}_{l'}$ of all successors $I_{l'}$ of I_l instead of the $E_{l'}$. In our technical report [4], we show that ψ_l is actually the weakest precondition.

$$\begin{aligned}
\psi^{(k)}_{|\mathfrak{p}|-1} &= Q \\
\psi^{(0)}_{l} &= \mathit{false} && \text{for } l \neq |\mathfrak{p}| - 1 \\
\psi^{(k+1)}_{l} &= \mathrm{wp}^1_{\mathfrak{p}}(I_l)^{(k)} && \text{for } l \neq |\mathfrak{p}| - 1 \\
\psi_l &= \textstyle\bigvee_{n \in \mathbb{N}_0} \psi^{(n)}_{l}
\end{aligned}$$

The fixed-point iteration can be avoided if programmers provide the specifications for those branch instructions that are part of a loop. This specification is typically the conjunction of the loop invariant and the property that the result of evaluating the loop condition is stored in $s(0)$. In our example, the loop invariant is $\mathtt{result} = 2^{P-\mathtt{p}}$, and the loop condition is $\mathtt{p} \neq 0$. Therefore, we get:

$$E_{14} \equiv \mathtt{result} = 2^{P-\mathtt{p}} \land s(0) = (\mathtt{p} \neq 0)$$

Based on this specification, we can calculate the instruction specifications of E_{14}'s predecessors by applying $\mathrm{wp}^1_{\mathtt{p}}$. The specifications in Fig. 4 are obtained from the calculated specifications by straightforward simplifications.

Since E_{14} has not been constructively derived, we have to prove that this specification is strong enough to establish the specifications of the successors, E_3 and E_{15}. That is, we have to show $E_{14} \Rightarrow \mathrm{wp}^1_{\mathtt{p}}(I_{14})$, which is easy:

$$(\mathtt{result} = 2^{P-\mathtt{p}} \wedge s(0) = (\mathtt{p} \neq 0)) \Rightarrow \\ (\neg s(0) \Rightarrow \mathit{shift}(\mathtt{result} = 2^{P})) \wedge (s(0) \Rightarrow \mathit{shift}(\mathtt{result} = 2^{P-\mathtt{p}}))$$

5.2 Transformation of Source Proofs

As explained in the introduction, one of the design criteria for the $\mathrm{VM_K}$ logic was to enable proof-transforming compilers, which translate a proof for a source program along with the code to $\mathrm{VM_K}$. In this subsection, we illustrate this approach by an example.

A proof-transforming compiler is based on transformation functions, $\mathcal{S}$ and $\mathcal{S}_E$, for statements and expressions, resp. Both functions yield a sequence of $\mathrm{VM_K}$ instructions and their specifications. $\mathcal{S}$ generates this sequence from a proof for a source statement. $\mathcal{S}_E$ generates is from a source expression and a precondition for its evaluation. These functions can be defined inductively, that is, the translation of a proof tree can be defined as a composition of the translations of its sub-trees [3].

For example, for proof trees whose root is an application of the while rule, $\mathcal{S}$ is defined as follows:

$$\mathcal{S}\left(\frac{\dfrac{T}{\{e \wedge P\}\ S\ \{P\}}}{\{P\}\ \mathtt{while}(e)S\ \{\neg e \wedge P\}}\right) = \begin{array}{c} \{P\}l_1 : \mathtt{goto}\ l_3 \\ \{e \wedge P\}l_2 : \mathcal{S}\left(\dfrac{T}{\{e \wedge P\}\ S\ \{P\}}\right) \\ \{P\}l_3 : \mathcal{S}_{\mathrm{E}}(P, e) \\ \{\mathit{shift}(P) \wedge s(0) = e\}l_4 : \mathtt{brtrue}\ l_2 \\ \{P \wedge \neg e\} \end{array}$$

The translation function uses symbolic labels. $\{e \wedge P\}\ l_2$ and $\{P\}\ l_3$ are the preconditions and labels of the first instructions generated by the applications of $\mathcal{S}$ to the loop body and $\mathcal{S}_E$ to the loop condition, resp. The "dangling" precondition $P \wedge \neg e$ is the precondition of the next instruction $l_4 + 1$ in the final method body. One can easily see that each instruction I_l satisfies $E_l \Rightarrow \mathrm{wp}^1_{\mathtt{p}}(I_l)$, that is, $\mathcal{S}$ generates a valid $\mathrm{VM_K}$ proof.

We illustrate the proof translation by the source code version of the method $\mathit{Math@pow2}$ in Fig. 4 (`result` is abbreviated by `r`):

```
r = 1;
while(p != 0)  { p = p - 1; r = r * 2; }
```

Like the bytecode version, the source implementation satisfies the triple $\{P = \mathtt{p}\}\ \mathit{Math@pow2}\ \{r = 2^P\}$. Consider the source proof for the while loop:

$$T_0 \equiv \cfrac{T_1 \equiv \cfrac{\cdots}{\{\mathtt{r} = 2^{P-\mathtt{p}} \wedge \mathtt{p} \neq 0\}\ \mathtt{p} = \mathtt{p} - 1; \mathtt{r\ =\ r\ *\ 2;}\ \{\mathtt{r} = 2^{P-\mathtt{p}}\}}}{\{\mathtt{r} = 2^{P-\mathtt{p}}\}\ \mathtt{while(p} \neq \mathtt{0)\{p = p - 1; r\ =\ r\ *\ 2;\}}\ \{\mathtt{r} = 2^{P-\mathtt{p}} \wedge \mathtt{p} = 0\}}$$

The translation of the proof for the loop body, $\mathcal{S}(T_1)$, yields instructions 3 to 10 of the instruction sequence in Fig. 4:

$$\mathcal{S}(T_1) \equiv \left[\{\mathtt{r} = 2^{P-\mathtt{p}} \wedge \mathtt{p} \neq 0\}\ 3 : \mathtt{pushv\ p}, \ldots, \{s(0) = 2^{P-\mathtt{p}}\}\ 10 : \mathtt{pop\ r}\right]$$

The translation of the whole loop, $\mathcal{S}(T_0)$, is obtained by applying the pattern described above. This translation yields the following instruction sequence, which corresponds to instructions 2 to 14 in Fig. 4:

$$\begin{aligned}\mathcal{S}(T_0) \equiv\ & \left[\{\mathtt{r} = 2^{P-\mathtt{p}}\}\ 2 : \mathtt{goto\ 11}\right] \cdot \mathcal{S}(T_1) \cdot \mathcal{S}_E(\mathtt{p} \neq 0, \mathtt{r} = 2^{P-\mathtt{p}}) \\ & \cdot \left[\{\mathit{shift}(\mathtt{r} = 2^{P-\mathtt{p}}) \wedge s(0) = (\mathtt{p} \neq 0)\}\ 14 : \mathtt{brtrue\ 3}\right]\end{aligned}$$

6 Related Work

Whereas the operational semantics of intermediate languages such as the .NET CIL and Java bytecode has been studied intensely [9,11,13,23], very few program logics for these languages have been published.

Our logic was inspired by Benton's logic for an imperative subset of the .NET CIL [5]. This logic does not support object-oriented features such as objects, references, or methods. Unlike Benton, we do not merge specifications and type information. Instead, we require that certain well-typedness constraints are checked by a bytecode verifier before our logic is applied.

Quigley [20,21] presents rules for Hoare-like reasoning about a small subset of Java bytecode within Isabelle. Her treatment is based on trying to rediscover high-level control structures (such as while loops), which precludes the verification of arbitrary instruction sequences.

The MRG project developed a program logic for the verification of functional and resource properties of a specialized form of Java bytecode (called Grail) [2]. Grail uses a functional form to represent bytecode, whereas our logic handles the imperative and object-oriented features of $\mathrm{VM_K}$ directly.

A number of program logics for object-oriented source programming languages have been proposed [1,7,12,14,16,17]. The object store model and the treatment of method specifications of the logic presented here are adopted from Poetzsch-Heffter and Müller's work [18,19].

7 Conclusions

We have presented a program logic for a bytecode language similar to Java bytecode and the .NET CIL. The key idea of the logic is to combine Hoare triples for methods with instruction specifications, which consist only of a precondition. Like in source logics, method specifications and the corresponding rules are used to handle inheritance and dynamic method binding. Specifications of individual instructions allow one to handle unstructured control flow in an unpretentious and effective manner.

As future work, we plan to use the $\mathrm{VM_K}$ logic to apply Proof-Carrying Code to functional correctness of Java programs. In particular, we will develop a proof-transforming compiler that translates verified source programs into verified bytecode. A first case study based on the $\mathrm{VM_K}$ logic lead to promising results.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.

[2] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Theorem Proving in Highre Order Logics (TPHOLs)*, LNCS. Springer-Verlag, 2004.

[3] F. Y. Bannwart. A logic for bytecode and the translation of proofs from sequential Java. ETH Zürich, 2004.

[4] F. Y. Bannwart and P. Müller. A logic for bytecode. Technical Report 469, ETH Zürich, 2004. Available from http://sct.inf.ethz.ch/publications.

[5] N. Benton. A typed logic for stacks and jumps. Available from research.microsoft.com/~nick/stacks.pdf, 2004.

[6] R. Berghammer. Soundness of a purely syntactical formalization of weakest preconditions. In D. Spreen, editor, *Electronic Notes in Theoretical Computer Science*, volume 35. Elsevier, 2000.

[7] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.

[8] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Programming Language Design and Implementation (PLDI)*, pages 95–107. ACM Press, 2000.

[9] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Principles of Programming Languages (POPL)*, pages 248–260. ACM Press, 2001.

[10] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[11] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[12] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer-Verlag, 2000.

[13] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.

[14] K. R. M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In B. Pierce, editor, *Foundations of Object-Oriented Languages (FOOL)*, 1997. Available from: www.cs.indiana.edu/hyplan/pierce/fool/.

[15] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, pages 106–119. ACM Press, 1997.

[16] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[17] D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2002.

[18] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.

[19] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[20] C. Quigley. A programming logic for Java bytecode programs. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.

[21] C. L. Quigley. *A Programming Logic for Java Bytecode Programs.* PhD thesis, University of Glasgow, 2004.

[22] N. Rauch. Precondition generation for a Java subset. In G. Schellhorn D. Haneberg and W. Reif, editors, *FM-TOOLS 2002*, Report 2002-11, pages 1–6. Universität Augsburg, Institut für Informatik, 2002.

[23] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation.* Springer-Verlag, 2001.