Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation

Egon Börger[1] and Wolfram Schulte[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] Universität Ulm, Fakultät für Informatik, D-89069 Ulm, Germany
wolfram@informatik.uni-ulm.de

Abstract. We provide concise abstract code for running the Java Virtual Machine (JVM) to execute compiled Java programs, and define a general compilation scheme of Java programs to JVM code. These definitions, together with the definition of an abstract interpreter of Java programs given in our previous work [3], allow us to prove that any compiler that satisfies the conditions stated in this paper compiles Java code correctly. In addition we have validated our JVM and compiler specification through experimentation.
The modularity of our definitions for Java, the JVM and the compilation scheme exhibit orthogonal language, machine and compiler components, which fit together and provide the basis for a stepwise and provably correct *design-for-reuse*. As a by-product we provide a challenging realistic case study for mechanical verification of a compiler correctness proof.

1 Introduction

Every justification showing that a proposed compiler behaves well is relative to a definition of the semantics of source and target language. In our previous work [3] we have developed a platform independent, rigorous yet easily manageable definition for an interpreter of Java programs, which captures the intuitive understanding Java programmers have of the semantics of their code. In this paper we provide a mathematical (read: rigorous and platform independent) yet practical model of an interpreter for the Java Virtual Machine, which formalizes the concepts presented in the JVM specification [6], as far as they are needed for the compilation of Java programs. We also extract from the JVM specification the definition of a scheme for the compilation of Java to JVM code and prove its correctness.

Main Theorem. *Every compiler that satisfies the conditions listed in this paper compiles Java programs correctly into JVM code.*

We split the JVM and the compilation function into an incremental sequence of four machines and functions—whose structure corresponds to the conservative extension relation among the modular components we exhibited for Java [3]—and define the JVM at two levels of abstraction: a ground model with an abstract

Luboš Prim et al. (Eds.): MFCS'98, LNCS 1450, pp. 17–35, 1998.

class file and abstract instructions, and a refined model where the abstract instructions are implemented by concrete JVM instructions. The structure of our Java machine is carried over mutatis mutandis to the basic structure of the abstract interpreter we are defining here for the JVM as target machine for Java compilation.

In sections 2 to 5 we define the sequence of successively extended JVM machines $\mathrm{JVM}_{\mathcal{I}}$, $\mathrm{JVM}_{\mathcal{C}}$, $\mathrm{JVM}_{\mathcal{O}}$ and $\mathrm{JVM}_{\mathcal{E}}$ for the compilation of programs from the imperative core $\mathrm{Java}_{\mathcal{I}}$ of Java and its extensions $\mathrm{Java}_{\mathcal{C}}$ (by classes, eg. procedures), $\mathrm{Java}_{\mathcal{O}}$ (by object-oriented features, eg. class instances) and $\mathrm{Java}_{\mathcal{E}}$ (by exceptions). We discuss here only the single threaded JVM, although our approach could easily include also multiple threads (see our multi-agent Java model with threads in [3]). We skip those language constructs which can be reduced by standard program transformation techniques to the core constructs dealt with explicitly in our Java models. We still do not consider Java packages, compilation units, visibility of names, strings, arrays, input/output, loading, linking and garbage collection. These features are the object of further refinements of the JVM model presented here. For proof details, the instruction refinement, an extensive bibliography and the discussion of related work we refer the interested reader to an extended version of this paper [1].

2 $\mathrm{JVM}_{\mathcal{I}}$ and the Compilation of $\mathrm{Java}_{\mathcal{I}}$ Programs

For the specification of Java, the JVM and the proof machinery, we use Abstract State Machines (ASMs). ASM specifications have a simple mathematical foundation [5], which justifies their intuitive understanding as "pseudo code" over abstract data. We define the basic JVM, called $\mathrm{JVM}_{\mathcal{I}}$, which is used as the target for compiling Java's statements and expressions over primitive types. We prove that $\mathrm{JVM}_{\mathcal{I}}$ executes the compilation of $\mathrm{Java}_{\mathcal{I}}$ programs correctly.

The following grammars recall the syntax of $\mathrm{Java}_{\mathcal{I}}$ [3] and introduce the corresponding instruction set $\mathrm{JVM}_{\mathcal{I}}$:

$$
\begin{array}{lll}
\mathit{Exp} & ::= & \mathit{Lit} \\
 & \mid & \mathit{Uop}\ \mathit{Exp} \\
 & \mid & \mathit{Exp}\ \mathit{Bop}\ \mathit{Exp} \\
 & \mid & \mathit{Var} \\
 & \mid & \mathit{Var} = \mathit{Exp} \\
 & \mid & \mathit{Exp}?\ \mathit{Exp} : \mathit{Exp} : \\
\\
\mathit{Stm} & ::= & ; \\
 & \mid & \mathit{Exp}; \\
 & \mid & \mathit{Lab} : \mathit{Stm} \\
 & \mid & \mathtt{break}\ \mathit{Lab}; \\
 & \mid & \mathtt{continue}\ \mathit{Lab}; \\
 & \mid & \mathtt{if}\ (\mathit{Exp})\ \mathit{Stm}\ \mathtt{else}\ \mathit{Stm} \\
 & \mid & \mathtt{while}\ (\mathit{Exp})\ \mathit{Stm} \\
 & \mid & \{\ \mathit{Stm}^*\ \}
\end{array}
\qquad
\begin{array}{lll}
\mathit{Instr} & ::= & \mathtt{const}\ (\mathit{Lit}) \\
 & \mid & \mathtt{uapply}\ (\mathit{Uop}) \\
 & \mid & \mathtt{bapply}\ (\mathit{Bop}) \\
 & \mid & \mathtt{load}\ (\mathit{Varnum} \times \mathit{Typ}) \\
 & \mid & \mathtt{store}\ (\mathit{Varnum} \times \mathit{Typ}) \\
 & \mid & \mathtt{dup}\ (\mathit{Typ}) \\
 & \mid & \mathtt{pop}\ (\mathit{Typ}) \\
 & \mid & \mathtt{ifZero}\ (\mathit{Lab}) \\
 & \mid & \mathtt{goto}\ (\mathit{Lab}) \\
 & \mid & \mathtt{label}\ (\mathit{Lab}) \\
\\
\mathit{Varnum} & == & \mathit{Nat} \\
\mathit{Code} & == & \mathit{Instr}^*
\end{array}
$$

The $\mathrm{JVM}_{\mathcal{I}}$ instruction set bears a close resemblence to a traditional stack machine like the P-machine. $\mathrm{JVM}_{\mathcal{I}}$ provides instructions to load constants, to apply various unary and binary operators, to load and store a variable, to duplicate and to remove values, and to jump unconditionally or conditionally to a label. Variable locations in the JVM are represented by natural numbers. A $\mathrm{JVM}_{\mathcal{I}}$ program is a sequence of instructions.

The universes *Lit*, *Uop*, *Bop*, *Var*, *Typ*, *Lab* contain Java literals, unary and binary operators, variables, primitive types and labels, respectively. With the exception of *Var*, these universes are also used in the $\mathrm{JVM}_{\mathcal{I}}$.

2.1 The Machine $\mathrm{JVM}_{\mathcal{I}}$ for Imperative Code

The JVM is a typed word-oriented stack-machine running the given bytecode *code* : *Code*. As a consequence the central dynamic part of a $\mathrm{JVM}_{\mathcal{I}}$ state consists of a program counter *pc*, a local variable environment *loc* and an operand stack *opd*. The following declarations show their formalization: the first column defines the used types, the second column defines the state, and the third column defines the condition on the initial state. (We consider sequences as isomorphic to functions having an interval of natural numbers starting at 0 as their domain.)

$$\begin{array}{lll} Pc == Nat & pc : Pc & pc = next_{unlab}(0, code) \\ Loc == Varnum \rightarrow Word & loc : Loc & loc = \emptyset \\ Opd == Word^* & opd : Opd & opd = \epsilon \end{array}$$

The close analogy between the abstract and concrete program counters in $\mathrm{Java}_{\mathcal{I}}$ and $\mathrm{JVM}_{\mathcal{I}}$, the memories for local variables and for intermediate values, and their initializations reflects the refinement process, which applied to the machine $\mathrm{Java}_{\mathcal{I}}$ yields $\mathrm{JVM}_{\mathcal{I}}$. This correspondence will guide the justification of the correctness of this first step towards an implementation of Java on the JVM.

Local variables and the operand stack store values of the abstract universe *Word*. *Word*s are supposed to hold at least 32-bit quantities. Java's values, which occupy at most 32-bits, are represented on the level of the JVM as single *Words*. Java's 64-bit values are mapped to multiple consecutive locations in the local environment and on the operand stack in an implementation dependent way. We define JVM values (*Val*) as sequences of Words, i.e. $Val == Word^*$. A valid word sequence has length one (32-bit) or two (64-bit). The JVM implements values and operations on Java datatypes as follows. Booleans are represented as integers: 0 is used for *false*, and 1 for *true*. Operations working on boolean, byte, short or char are not supported by the JVM. Instead, upon retrieving the value of a boolean, byte, char or short, it is automatically cast into an int. When writing a value to a boolean, byte, char or short variable, an int is passed and the JVM truncates it to the relevant size.

For the $\mathrm{JVM}_{\mathcal{I}}$ we use two static code traversing functions *next* and *jump*, which yield the next statement to be executed and the next statement after the given labeled statement, respectively. Both functions are defined using an auxiliary function $next_{unlab}$ that skips `label` instructions. (The expression $\iota\, x \mid p(x)$ denotes the uniquely determined object x that satisfies $p(x)$.)

$$\begin{array}{ll} next(pc, code) & = next_{unlab}(pc + 1, code) \\ jump(l, code) & = next_{unlab}(\iota\, pc \mid code(pc) = \mathtt{label}\,(l)) \\ next_{unlab}(pc, code) & = min\{pc' \mid pc' \geq pc \wedge \forall l \mid code(pc') \neq \mathtt{label}\,(l)\} \end{array}$$

We also use the following $\mathrm{JVM}_{\mathcal{I}}$ macros, where the homonymy with $\mathrm{Java}_{\mathcal{I}}$ macros reflects the refinement relations on which our correctness proof is based.

$$\begin{array}{ll} proceed & == pc := next(pc, code) \\ goto(l) & == pc := jump(l, code) \\ pc \text{ is } instr & == code(pc) = instr \end{array}$$

The following rules define the semantics of the $\mathrm{JVM}_{\mathcal{I}}$ instructions.

$$\begin{array}{l}
\textbf{if } pc \text{ is } \mathtt{const}\,(lit) \\
\textbf{then} \\
\quad opd := \widetilde{lit} \cdot opd \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{uapply}\,(\odot) \wedge \\
\quad (v, opd') = split(\mathcal{A}(\odot), opd) \\
\textbf{then} \\
\quad opd := \widetilde{\odot}\, v \cdot opd' \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{bapply}\,(\otimes) \wedge \\
\quad (v_2, v_1, opd') = split(\mathcal{A}(\otimes), opd) \wedge \\
\quad (\otimes \in DivMods) \Rightarrow (v_2 \neq 0) \\
\textbf{then} \\
\quad opd := v_1 \,\widetilde{\otimes}\, v_2 \cdot opd' \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{dup}\,(t) \wedge \\
\quad (v, opd') = split(t, opd) \\
\textbf{then} \\
\quad opd := v \cdot v \cdot opd' \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{pop}\,(t) \wedge \\
\quad (v, opd') = split(t, opd) \\
\textbf{then} \\
\quad opd := opd' \\
\quad proceed
\end{array}$$

$$\begin{array}{l}
\textbf{if } pc \text{ is } \mathtt{load}\,(x, t) \\
\textbf{then} \\
\quad \textbf{if } sizeof(t) = 1 \textbf{ then} \\
\quad\quad opd := loc(x) \cdot opd \\
\quad \textbf{else if } sizeof(t) = 2 \textbf{ then} \\
\quad\quad opd := loc(x) \cdot loc(x+1) \cdot opd \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{store}\,(x, t) \wedge \\
\quad (v, opd') = split(t, opd) \\
\textbf{then} \\
\quad opd := opd' \\
\quad \textbf{if } sizeof(t) = 1 \textbf{ then} \\
\quad\quad loc(x) := v(0) \\
\quad \textbf{else if } sizeof(t) = 2 \textbf{ then} \\
\quad\quad loc(x+1) := v(1) \\
\quad\quad loc(x) := v(0) \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{goto}\,(l) \textbf{ then} \\
\quad goto(l) \\
\\
\textbf{if } pc \text{ is } \mathtt{ifZero}\,(l) \wedge \\
\quad w \cdot opd' = opd \\
\textbf{then} \\
\quad opd := opd' \\
\quad \textbf{if } w = 0 \textbf{ then } goto(l) \\
\quad\quad\quad\quad\quad \textbf{else } proceed
\end{array}$$

A `const` instruction pushes the JVM value $\widetilde{lit}$ (one or two words) on the operand stack. An unary (binary) operator changes the value(s) on top of the operand stack. The unary (binary) operators are assumed to have the same meaning as in Java (i.e. $\widetilde{\odot}$ ($\widetilde{\otimes}$)), although they may operate on extended domains. In order to abstract from the different value sizes, we use the function $split : (Typ^*, Opd) \rightarrow (Val^*, Opd)$, which given a sequence of n types and the operand stack, takes the top n values from the operand stack, such that the ith value has the size of the ith type. The function $\mathcal{A}(op)$ returns the argument types of op. The instructions dup and pop duplicate and remove the top stack value, respectively. A `load` instruction loads the value stored under the location x on top of the stack. If the type of x is a double or long, the next two locations are pushed on top of the stack. A `store` instruction stores the top (two) word(s) of the operand stack in the local environment at offset x (and $x+1$). A `goto` instruction causes execution to jump to the next instruction determined by the label. The `ifzero` instruction is a conditional `goto`. If the value on top of the operand stack is 0, execution continues at the next instruction determined by the label, otherwise execution proceeds.

The abstract nature of the $\mathrm{JVM}_\mathcal{I}$ instructions is reflected in their parameterization by types and operators. It allows us to restrict our attention to a small set of JVM instructions (or better instruction classes) without losing the generality of our model with respect to the JVM specification [6]. The extended version of this paper [1] shows how to refine these parameterized instruction to JVM's real ones.

2.2 Compilation of $\mathrm{Java}_\mathcal{I}$ Programs to $\mathrm{JVM}_\mathcal{I}$ Code

This section defines the compiling function from $\mathrm{Java}_\mathcal{I}$ to $\mathrm{JVM}_\mathcal{I}$ code. More efficient compilation schemes can be introduced but we leave optimizations for further refinement steps.

The compilation $\mathcal{E} : \mathit{Exp} \rightarrow \mathit{Code}$ of (occurrences of) $\mathrm{Java}_\mathcal{I}$ expressions to $\mathrm{JVM}_\mathcal{I}$ instructions is standard. The resulting sequence of instructions has the effect of storing the value of the expression on top of the operand stack.

To improve readability, we use the following conventions for the presentation of the compilation: We suppress the routine machinery for a consistent assignment of (occurrences of) Java variables x to JVM variable numbers $\overline{x}$. Similarly, we suppress the trivial machinery for label generation. Label providing functions lab_i, $i \in \mathit{Nat}$, are defined on occurrences of expressions and statements, are supposed to be injective and to have disjoint ranges. Functions $\mathcal{T}$ defined on occurrences of variables and expressions return their type. We abbreviate: 'Let e be an occurrence of exp in $\mathcal{E}(e) = \ldots$' by '$\mathcal{E}(e \textbf{ as } exp) = \ldots$'.

$$
\begin{array}{ll}
\mathcal{E}(lit) & = \texttt{const}\,(lit) \\
\mathcal{E}(\odot\, e) & = \mathcal{E}e \cdot \texttt{uapply}\,(\odot) \\
\mathcal{E}(e_1 \otimes e_1) & = \mathcal{E}e_1 \cdot \mathcal{E}e_2 \cdot \texttt{bapply}\,(\otimes) \\
\mathcal{E}(x) & = \texttt{load}\,(\overline{x}, \mathcal{T}(x)) \\
\mathcal{E}(x = e) & = \mathcal{E}e \cdot \texttt{dup}\,(\mathcal{T}(e)) \cdot \texttt{store}\,(\overline{x}, \mathcal{T}(x)) \\
\mathcal{E}(e \textbf{ as } e_1 ?\, e_2 :\, e_3 :) & = \mathcal{E}e_1 \cdot \texttt{ifZero}\,(lab_1(e)) \cdot \\
 & \quad \mathcal{E}e_2 \cdot \texttt{goto}\,(lab_2(e)) \cdot \texttt{label}\,(lab_1(e)) \cdot \mathcal{E}e_3 \cdot \texttt{label}\,(lab_2(e))
\end{array}
$$

Also the compilation $\mathcal{S} : \mathit{Stm} \rightarrow \mathit{Code}$ of $\mathrm{Java}_\mathcal{I}$ statements to $\mathrm{JVM}_\mathcal{I}$ instructions is standard. The compilation of `break` lab; and `continue` lab; uses the auxiliary function $target : \mathit{Stm} \times \mathit{Lab} \rightarrow \mathit{Stm}$. This function provides for occurrences of statements and labels the occurrence of the enclosing labeled statement in the given program.

$$\begin{aligned}
&\mathcal{S}(;) &&= \epsilon\\
&\mathcal{S}(e;) &&= \mathcal{E}e \cdot \mathsf{pop}\,(\mathcal{T}(e))\\
&\mathcal{S}(\{\,s_1 \ldots s_m\}) &&= \mathcal{S}s_1 \cdot \ldots \cdot \mathcal{S}s_m
\end{aligned}$$

$$\begin{aligned}
&\mathcal{S}(s \textbf{ as } \mathtt{if}\,(e)\, s_1 \,\mathtt{else}\, s_2) &&= \mathcal{E}e \cdot \mathtt{ifZero}\,(lab_1(s)) \cdot\\
& &&\quad \mathcal{S}s_1 \cdot \mathtt{goto}(lab_2(s)) \cdot \mathtt{label}(lab_1(s)) \cdot \mathcal{S}s_2 \cdot \mathtt{label}(lab_2(s))\\
&\mathcal{S}(s \textbf{ as } \mathtt{while}\,(e)\, s_1) &&= \mathtt{label}\,(lab_1(s)) \cdot \mathcal{E}e \cdot \mathtt{ifZero}\,(lab_2(s)) \cdot\\
& &&\quad \mathcal{S}s_1 \cdot \mathtt{goto}\,(lab_1(s)) \cdot \mathtt{label}\,(lab_2(s))\\
&\mathcal{S}(s \textbf{ as } lab : s_1) &&= \mathtt{label}\,(lab_1(s)) \cdot \mathcal{S}s_1 \cdot \mathtt{label}\,(lab_2(s))\\
&\mathcal{S}(s \textbf{ as } \mathtt{continue}\, lab;) &&= \mathtt{goto}\,(lab_1(target(s, lab)))\\
&\mathcal{S}(s \textbf{ as } \mathtt{break}\, lab;) &&= \mathtt{goto}\,(lab_2(target(s, lab)))
\end{aligned}$$

Correctness Theorem for $Java_{\mathcal{I}}/JVM_{\mathcal{I}}$. Via the refinement relation and under the assumptions stated above, the result of executing any $\text{Java}_{\mathcal{I}}$ program in the machine $\text{Java}_{\mathcal{I}}$ is equivalent to the result of executing the compiled program on the machine $\text{JVM}_{\mathcal{I}}$.

3 $\text{JVM}_{\mathcal{C}}$ and the Compilation of Class Code

In this section we extend the basic $\text{JVM}_{\mathcal{I}}$ machine to the machine $\text{JVM}_{\mathcal{C}}$, which handles class (also called static) fields, class methods and class initializers. $\text{JVM}_{\mathcal{C}}$ thus stands for a machine that supports modules, module-local variables and procedures. We add the clauses for compiling class field access, class field assignment, class method calls and return statements to the definition of the $\text{Java}_{\mathcal{I}}$ compilation function.

The following grammar shows the extension of the syntax of $\text{Java}_{\mathcal{I}}$ to the syntax of $\text{Java}_{\mathcal{C}}$. Furthermore, we define the corresponding $\text{JVM}_{\mathcal{C}}$ instructions:

$$\begin{aligned}
Exp &::= \ldots\\
&\mid\ FieldSpec\\
&\mid\ FieldSpec = Exp\\
&\mid\ MethSpec(Exp^*)\\
Stm &::= \ldots\\
&\mid\ \mathtt{return};\\
&\mid\ \mathtt{return}\ Exp;\\
Init &::= \mathtt{static}\ Stm
\end{aligned}$$

$$\begin{aligned}
Instr &::= \ldots\\
&\mid\ \mathtt{getstatic}\,(FieldSpec \times Typ)\\
&\mid\ \mathtt{putstatic}\,(FieldSpec \times Typ)\\
&\mid\ \mathtt{invokestatic}\,(MethSpec)\\
&\mid\ \mathtt{return}\,(Typ)\\
Fcty &== (Typ^* \times Typ)\\
FieldSpec &== (Class \times Field)\\
MethSpec &== (Class \times Meth \times Fcty)
\end{aligned}$$

$\text{JVM}_{\mathcal{C}}$ provides instructions to load and store class fields, and to call and to return from class methods. Both grammars are based on the same abstract definition of field and method specifications. Field specifications consist of a class and a field name, because Java and the JVM allow fields in different classes to have the same name. Method specifications additionally have a functionality (a sequence of argument types and a result type, which can be `void`), because Java and the JVM support classes with methods having the same name but taking different parameter types.

Field and method specifications use the abstract universes *Class*, *Field* and *Method*. *Class* is assumed to stand for fully qualified Java class names, *Field* and *Method* for identifiers.

3.1 The Machine $\mathrm{JVM}_{\mathcal{C}}$ for Class Code

JVM and Java programs are structured into classes, which establish the program's execution environment. For a general, high-level definition of a provably correct compilation scheme from Java to JVM Code, we can abstract from many data structure specifics of the particular JVM class format. This format is called class file in the JVM specification [6].

Our abstract class file refines in a natural way the class environment of $\mathrm{Java}_{\mathcal{C}}$, providing for every class its kind (whether it is a class or an interface), its superclass (if there is any), a list of the interfaces the class implements, and a table for fields and methods. Class files do not include definitions for fields or methods provided by any superclass.

$$\begin{array}{lll} Env & == & Class \to ClassDec \\ ClassKind & ::= & AClass \mid AnInterface \\ ClassDec & == & (kind : ClassKind \times super : [Class] \times ifaces : Class^* \times \\ & & \quad fTab : Field \to FieldDec \times mTab : (Meth \times Fcty) \to MethDec) \end{array}$$

In $\mathrm{JVM}_{\mathcal{C}}$ fields and methods can only be static. Fields have a type and optionally a constant value. If a method is implemented in the class, the method body defines its code.

$$\begin{array}{lll} FieldDec & == & (fKind : MemberKind \times fTyp : Typ \times fConstVal : [Val]) \\ MethDec & == & (mKind : MemberKind \times mBody : [Code]) \\ MemberKind & ::= & Static \end{array}$$

In $\mathrm{JVM}_{\mathcal{C}}$ we have a fixed environment *env* : *Env*, defined by the given program. The following functions operate on this environment. The function *mCode* retrieves for a given method specification the method's code to be executed. The function *fInitVal* yields for a given field specification the field's constant value, provided it is available; otherwise, the function returns the default value of the field's type (where *default* : *Typ* $\to$ *Val*).

$$\begin{array}{lll} mCode(c, m, f) & = & mBody(mTab(env(c))(m, f)) \\ fInitVal(c, f) & = & \textbf{case } fTab(env(c))(f) \textbf{ of } \begin{array}[t]{ll} (\_, \_, val) & : val \\ (\_, fTyp, []) & : default(fTyp) \end{array} \end{array}$$

The function *supers* calculates the transitive closure of *super*. The function *cfields* returns the set of all fields declared by the class.

$$\begin{array}{l} supers : Class \to Class^* \\ cfields : Class \to \mathcal{P}\, FieldSpec \end{array}$$

For these functions the homonymy to Java$_{\mathcal{C}}$ functions shows the data refinement relation in going from Java$_{\mathcal{C}}$ to JVM$_{\mathcal{C}}$.

Due to the presence of method calls in JVM$_{\mathcal{C}}$ we have to embed the one single JVM$_{\mathcal{I}}$ frame (pc, loc, opd) into the JVM$_{\mathcal{C}}$ frame stack *frames*, enriched by a fourth component which always holds the dynamic chain of method specifications. This embedding defines the refinement relation between JVM$_{\mathcal{I}}$ and JVM$_{\mathcal{C}}$. We refine the static function *code*, so that it always denotes the code stored in the environment under the current method specification *mspec*. The current class, method and functionality are denoted by *cclass*, *cmeth* and *cfcty*, respectively, where $mspec = (cclass, cmeth, cfcty)$.

$$
\begin{array}{lll@{\qquad}lll}
pcs & : & Pc^* & pc & == & top(pcs) \\
locs & : & Loc^* & loc & == & top(locs) \\
opds & : & Opd^* & opd & == & top(opds) \\
mspecs & : & MethSpec^* & mspec & == & top(mspecs) \\
frames & == & (pcs, locs, opds, mspecs) & code & == & mCode(mspec)
\end{array}
$$

Before a class can be used its class initializers must be executed. At the JVM level class initializers appear as class methods with the special name `<clinit>`. Initialization must be done lazily, i.e. when a class is *first used* in Java, and when a reference is *resolved* in the JVM. Resolution is the process of checking symbolic references from the current class to other classes and interfaces. Since Java's notion of class initialization does not correspond to the related class resolution notion of the JVM, we name the initialization related functions and sets differently. A class can be in one of three states. We introduce a dynamic function *res*, which records the current resolution state. A class is *resolved*, if resolution for this class is in progress or done.

$$
\begin{array}{l}
res : Class \rightarrow ResolvedState \\
ResolvedState ::= Unresolved \mid Resolved \mid InProgress \\
resolved(state) = state \in \{InProgress, Resolved\}
\end{array}
$$

The JVM specification [6] uses symbolic references, namely field and method specifications, to support binary compatibility, cf. [4]. As a consequence, the calculation of field offsets and of method offsets is implementation dependent. Therefore, we keep the class field access abstract and define the storage function for class fields to be the same in Java$_{\mathcal{C}}$ and JVM$_{\mathcal{C}}$, namely

$$glo : FieldSpec \rightarrow Val.$$

The runs of JVM$_{\mathcal{C}}$ start with calling the class method *main* of a distinguished class *Main* being part of the environment. However, before *main* is executed, its class *Main* has to be initialized. Therefore, the frame stack initially has two entries: the *main* method at the bottom and the `<clinit>` method on the top. All classes are initially unresolved and all fields are set to their initial values. This initialization also refines the corresponding conditions imposed on Java$_{\mathcal{C}}$:

$$
\begin{array}{ll@{\qquad}l}
pcs & = start(clinit') \cdot start(main') & res = \{(c, Unresolved) \mid c \in dom(env)\} \\
locs & = \epsilon \cdot \epsilon & glo = \{(fs, fInitVal(fs)) \mid c \in dom(env), \\
opds & = \epsilon \cdot \epsilon & \qquad\qquad fs \in cfields(c)\} \\
mspecs & = clinit' \cdot main' &
\end{array}
$$

The method specifications *clinit'* and *main'* denote the class methods `<clinit>` and *main* of class *Main*. The macro *start* returns the first instruction of the code of the given method specification.

$$
\begin{array}{ll}
clinit' == proc(Main, \texttt{<clinit>}) & start(ms) \;==\; next_{unlab}(0, mCode(ms)) \\
main' == proc(Main, main) & proc(c, m) == (c, m, (\epsilon, \mathtt{void}))
\end{array}
$$

The following rules for JVM_C define the semantics of the new JVM instructions, provided the class of the field or method specification is already resolved. A `getstatic` instruction loads the value (one or two words), stored under the field specification in the global environment, on top of the operand stack. A `putstatic` instruction stores the top (two) word(s) of the operand stack in the global environment at the given field specification. An `invokestatic` instruction pops the arguments from the stack and sets *pc* to the next instruction. The arguments of the invoked method are placed in the local variables of the new stack frame, and execution continues at the first instruction of the new method. A `return` instruction is 'inverse' to `invokestatic`. It pops a value from the top of the stack and pushes it onto the operand stack of the invoker. All other items in the current stack are discarded. (If the return type is `void`, *split* returns the empty sequence as its value.)

$$
\begin{array}{l}
\textbf{if } pc \text{ is } \mathtt{getstatic}\ ((c,f),t) \wedge \\
\quad resolved(res(c)) \\
\textbf{then} \\
\quad opd := glo(c,f) \cdot opd \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{putstatic}\ ((c,f),t) \wedge \\
\quad resolved(res(c)) \wedge \\
\quad (v, opd') = split(t, opd) \\
\textbf{then} \\
\quad opd := opd' \\
\quad glo(c,f) := v \\
\quad proceed
\end{array}
$$

$$
\begin{array}{l}
\textbf{if } pc \text{ is } \mathtt{invokestatic}\ (c, m, (ts, t)) \wedge \\
\quad resolved(res(c)) \wedge (t_1, \ldots, t_n) = ts \wedge \\
\quad (v_n, \ldots, v_1, opd') = split(t_n, \ldots, t_1, opd) \\
\textbf{then} \\
\quad call(\, next(pc, code), v_1 \cdot \ldots \cdot v_n, \\
\qquad\quad opd', (c, m, (ts, t))) \\
\textbf{if } pc \text{ is } \mathtt{return}\ (t) \wedge \\
\quad (v, opd') = split(t, opd) \\
\textbf{then} \\
\quad return(v)
\end{array}
$$

The macros *call* and *return* update the frames as follows:

$$
\begin{array}{l}
call(pc, loc, opd, mspec) == \\
\quad \textbf{let } pc_0 \cdot pcs' = pcs \\
\qquad\; opd_0 \cdot opds' = opds \textbf{ in} \\
\quad pcs \;\;\;\;\; := start(mspec) \cdot pc \cdot pcs' \\
\quad locs \;\;\;\, := loc \cdot locs \\
\quad opds \;\;\; := \epsilon \cdot opd \cdot opds' \\
\quad mspecs := mspec \cdot mspecs
\end{array}
$$

$$
\begin{array}{l}
return(v) == \\
\quad \textbf{if } len(pcs) = 1 \textbf{ then} \\
\qquad pcs(0) := undef \\
\quad \textbf{else let } opd_0 \cdot opd_1 \cdot opds' = opds \textbf{ in} \\
\qquad pcs \;\;\;\;\; := pop(pcs) \\
\qquad locs \;\;\;\, := pop(locs) \\
\qquad opds \;\;\; := (v \cdot opd_1) \cdot opds' \\
\qquad mspecs := pop(mspecs)
\end{array}
$$

Execution starts in a state in which no class is resolved. A class is resolved, when it is first referenced. Before a class is resolved, its superclass is resolved (if any). Interfaces are not resolved at this time, although this is not specified in Java's language reference manual [4]. On the level of the JVM resolution leads to three rules. First, resolutions starts, i.e. the class method `<clinit>` is implicitely called, when the class referred to in a `get-`, `put-` or `invokestatic` instruction is not resolved. Second, the class initializer records the fact that class initialization is in progress and calls the superclass initializer recursively. Third, after having executed the class initializer, it is recorded that the class is resolved.

$$\begin{array}{l}
\textbf{if } (pc \text{ is } \mathtt{putstatic}\ ((c,_),_) \lor \\
\quad pc \text{ is } \mathtt{getstatic}\ ((c,_),_) \lor \\
\quad pc \text{ is } \mathtt{invokestatic}\ (c,_,_)) \land \\
\quad \neg resolved(res(c)) \\
\textbf{then} \\
\quad call(pc, \emptyset, opd, proc(c, \mathtt{<clinit>}))
\end{array}$$

$$\begin{array}{l}
\textbf{if } res(cclass) = Unresolved \\
\textbf{then} \\
\quad res(cclass) := InProgress \\
\quad \textbf{if } supers(cclass) \neq \epsilon \land \\
\qquad \neg resolved(res(super(cclass))) \\
\quad \textbf{then} \\
\qquad call(pc, \emptyset, opd, \\
\qquad\qquad proc(super(cclass), \mathtt{<clinit>})) \\
\textbf{if } pc \text{ is } \mathtt{return}\ (t) \land cmeth = \mathtt{<clinit>} \\
\textbf{then} \\
\quad res(cclass) := Resolved
\end{array}$$

Firing the second rule depends on the condition that the current class is *Unresolved*—this is the reason why we called the initializer in the first rule. To suppress the simultaneous firing of other rules we strengthen the macro 'is':

$$pc \text{ is } instr == code(pc) = instr \land resolved(res(cclass))$$

This guarantees that an instruction can only be executed, if the current class is resolved. Opposite to the second rule, the third rule fires simultaneously with the previously presented rule for the `return` instruction.

3.2 Compilation of Java$_\mathcal{C}$ Programs to JVM$_\mathcal{C}$ Code

The compilation of Java$_\mathcal{I}$ expressions is extended by defining the compilation of class field access, class field assignment, and by the compilation of calls of class methods.

$$\begin{array}{ll}
\mathcal{E}(fspec) & = \mathtt{getstatic}\ (fspec, \mathcal{T}(fspec)) \\
\mathcal{E}(fspec = e) & = \mathcal{E}e \cdot \mathtt{dup}\ (\mathcal{T}(e)) \cdot \mathtt{putstatic}\ (fspec, \mathcal{T}(fspec)) \\
\mathcal{E}(mspec(e_1, \ldots, e_n)) & = \mathcal{E}e_1 \cdot \ldots \cdot \mathcal{E}e_n \cdot \mathtt{invokestatic}\ (mspec)
\end{array}$$

We add the clause for `return` statements to the Java$_\mathcal{I}$ compilation.

$$\begin{array}{ll}
\mathcal{S}(\mathtt{return}\ e;) & = \mathcal{E}e \cdot \mathtt{return}\ (\mathcal{T}(e)) \\
\mathcal{S}(\mathtt{return};) & = \mathtt{return}\ (\mathtt{void})
\end{array}$$

To compile a class initializer (the *Init* phrase) means to compile its statement as the body of the static `<clinit>` method.

The extension of Java$_\mathcal{I}$/JVM$_\mathcal{I}$ to Java$_\mathcal{C}$/JVM$_\mathcal{C}$ is conservative, i.e. purely incremental. For the proof of the *Correctness Theorem for* Java$_\mathcal{C}$/JVM$_\mathcal{C}$ it therfore suffices to extend the theorem from Java$_\mathcal{I}$/JVM$_\mathcal{I}$ to the new expressions and statements occurring in Java$_\mathcal{C}$/JVM$_\mathcal{C}$.

4 $\mathrm{JVM}_{\mathcal{O}}$ and the Compilation of $\mathrm{Java}_{\mathcal{O}}$ Programs

In this section we extend the machine $\mathrm{JVM}_{\mathcal{C}}$ to $\mathrm{JVM}_{\mathcal{O}}$. This machine handles the object-oriented features of Java programs, namely instances, instance creation, instance field access, instance method calls with late binding, type casts and null pointers. We add the corresponding new phrases to the definition of the compilation function.

We recall the grammar for the new expressions of $\mathrm{Java}_{\mathcal{O}}$ and define the corresponding $\mathrm{JVM}_{\mathcal{O}}$ instructions:

$$
\begin{array}{lll}
\mathit{Exp} & ::= & \dots \\
 & | & \mathtt{this} \\
 & | & \mathtt{new}\ \mathit{ConstrSpec}\,(\mathit{Exp}^*) \\
 & | & \mathit{ConstrSpec}\,(\mathit{Exp}^*) \\
 & | & \mathit{Exp}.\mathit{FieldSpec} \\
 & | & \mathit{Exp}.\mathit{FieldSpec} = \mathit{Exp} \\
 & | & \mathit{Exp}.\mathit{MethSpec}\{\mathit{CallKind}\}(\mathit{Exp}^*) \\
 & | & \mathit{Exp}\ \mathtt{instanceof}\ \mathit{Class} \\
 & | & (\mathit{Class})\ \mathit{Exp} \\
\mathit{ConstrSpec} & == & (\mathit{Class} \times \mathit{Typ}^*)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Instr} & ::= & \dots \\
 & | & \mathtt{new}\,(\mathit{Class}) \\
 & | & \mathtt{getfield}\,(\mathit{FieldSpec} \times \mathit{Typ}) \\
 & | & \mathtt{putfield}\,(\mathit{FieldSpec} \times \mathit{Typ}) \\
 & | & \mathtt{dup}\_(\mathit{Typ}^*) \\
 & | & \mathtt{invokeinstance}\,(\mathit{MethSpec} \times \mathit{CallKind}) \\
 & | & \mathtt{instanceof}\,(\mathit{Class}) \\
 & | & \mathtt{checkcast}\,(\mathit{Class}) \\
\mathit{CallKind} & ::= & \mathit{Constr} \mid \mathit{Nonvirtual} \\
 & | & \mathit{Virtual} \mid \mathit{Super}
\end{array}
$$

$\mathrm{Java}_{\mathcal{O}}$ uses constructor specifications to uniquely denote overloaded instance constructors. $\mathrm{JVM}_{\mathcal{O}}$ provides instructions to allocate a new instance, to access or assign its fields, to duplicate values, to invoke instance methods and to check instance types. $\mathrm{Java}_{\mathcal{O}}$ and $\mathrm{JVM}_{\mathcal{O}}$ use the universe *CallKind*, to distinguish the particular way in which instance methods are called.

4.1 The Machine $\mathrm{JVM}_{\mathcal{O}}$ for Object-Oriented Code

$\mathrm{JVM}_{\mathcal{O}}$ uses the same abstract class file as $\mathrm{JVM}_{\mathcal{C}}$. However, instance fields and instance methods—in opposite to class fields and class methods—are not static but dynamic. So we extend the universe *MemberKind* as follows:

$$\mathit{MemberKind} ::= \dots \mid \mathit{Dynamic}$$

The JVM specification [6] fixes the class file. However, the specification does not explain how instances are stored or instance methods are accessed. So we extend the signature of $\mathrm{JVM}_{\mathcal{C}}$ in $\mathrm{JVM}_{\mathcal{O}}$ in the same way as the signature of $\mathrm{Java}_{\mathcal{C}}$ is extended in $\mathrm{Java}_{\mathcal{O}}$. We introduce the following static functions (homonymy with $\mathrm{Java}_{\mathcal{O}}$ functions) that look up information in the global environment:

$$
\begin{array}{ll}
\mathit{dfields} & : \mathit{Class} \rightarrow \mathcal{P}\ \mathit{FieldSpec} \\
\mathit{dlookup} & : \mathit{Class} \times \mathit{MethSpec} \rightarrow \mathit{Class} \\
\mathit{compatible} & : \mathit{Class} \times \mathit{Class} \rightarrow \mathit{Bool}
\end{array}
$$

The function *dfields* determines the instance fields of a class and of all its superclasses (if any). The function *dlookup* returns the first (super) class for the given method specification, which implements this method. The expression *compatible*(*myType*, *tarType*) returns *true* if *myType* is assignment compatible with *tarType* [4]. Note that at the JVM level, there is no special lookup function for constructors. Instead, Java's constructors appear in the JVM as instance initialization methods with the special name `<init>`.

$\mathrm{JVM}_{\mathcal{O}}$ and $\mathrm{Java}_{\mathcal{O}}$ have the same dynamic functions for memorizing the class and the instance field values of a reference. In both machines they are initially empty. References can be obtained from the abstract universe *Ref*, which is assumed to be a subset of *Word*. (Likewise, we also assume that *null* is an element of *Word*.)

$$\begin{array}{llll} \mathit{classOf} & : \mathit{Ref} \to \mathit{Class} & \qquad \mathit{classOf} & = \emptyset \\ \mathit{dyn} & : \mathit{Ref} \times \mathit{FieldSpec} \to \mathit{Val} & \qquad \mathit{dyn} & = \emptyset \end{array}$$

The following rules define the semantics of the new instructions of $\mathrm{JVM}_{\mathcal{O}}$, provided that the involved class is resolved.

$$\begin{array}{l}
\textbf{if } pc \text{ is } \mathtt{new}\ (c) \wedge \\
\quad resolved(res(c)) \\
\textbf{then} \\
\quad \textbf{extend } Ref \textbf{ by } r \\
\quad\quad classOf(r) := c \\
\quad\quad \textbf{vary } fs \textbf{ over } dfields(c) \\
\quad\quad\quad dyn(r, fs) := fInitVal(fs) \\
\quad\quad opd := r \cdot opd \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{getfield}\ ((c, f), t) \wedge \\
\quad resolved(res(c)) \wedge \\
\quad r \cdot opd' = opd \wedge \\
\quad r \neq null \\
\textbf{then} \\
\quad opd := dyn(r, (c, f)) \cdot opd' \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{putfield}\ ((c, f), t) \wedge \\
\quad resolved(res(c)) \wedge \\
\quad (v, r, opd') = split(t, c, opd) \wedge \\
\quad r \neq null \\
\textbf{then} \\
\quad opd := opd' \\
\quad dyn(r, (c, f)) := v \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{dup_}(t_1, t_2) \wedge \\
\quad (v_2, v_1, opd') = split(t_2, t_1, opd) \\
\textbf{then} \\
\quad opd := v_2 \cdot v_1 \cdot v_2 \cdot opd' \\
\quad proceed
\end{array}$$

$$\begin{array}{l}
\textbf{if } pc \text{ is } \mathtt{invokeinstance}\ ((c, m, (ts, t)), k) \wedge \\
\quad resolved(res(c)) \wedge (t_1, \ldots t_n) = ts \wedge \\
\quad (v_n, \ldots, v_1, r, opd') = \\
\quad\quad split(t_n, \ldots, t_1, c), opd) \wedge \\
\quad r \neq null \\
\textbf{then} \\
\quad call(\, next(pc, code), r \cdot v_1 \cdot \ldots \cdot v_n, \\
\quad\quad opd', (c', m, (ts, t)) \\
\quad \textbf{where} \\
\quad c' = \textbf{case } k \textbf{ of} \\
\quad\quad \begin{array}{ll} Constr & : c \\ Nonvirtual & : cclass \\ Virtual & : dlookup(\, classOf(r), \\ & \qquad\qquad m, (ts, t)) \\ Super & : dlookup(\, super(cclass), \\ & \qquad\qquad m, (ts, t)) \end{array} \\
\textbf{if } pc \text{ is } \mathtt{instanceof}\ (c) \wedge \\
\quad resolved(res(c)) \wedge \\
\quad r \cdot opd' = opd \\
\textbf{then} \\
\quad opd := (r \neq null \wedge \\
\quad\quad\quad compatible(classOf(r), c) \cdot opd' \\
\quad proceed \\
\textbf{if } pc \text{ is } \mathtt{checkcast}\ (c) \wedge \\
\quad resolved(res(c)) \wedge \\
\quad r \cdot opd' = opd \wedge \\
\quad (r = null \vee compatible(classOf(r), c)) \\
\textbf{then} \\
\quad proceed
\end{array}$$

A `new` instruction allocates a fresh reference using the domain extension update of ASMs. The *classOf* the reference is set to the given class, the class instance fields are set to default values, and the new reference is pushed on the operand stack. A `getfield` instruction pops the target reference from the stack, retrieves the value of the field identified by the given field specification from the dynamic store and pushes one or two words on the operand stack. A `putfield` instruction pops a value and the target reference from the stack and sets the dynamic store at the point of the target reference and the given field specification to the popped value. A `dup_` instruction duplicates the top value and inserts the duplicate below the top value on the stack. An `invokeinstance` instruction pops the arguments and the target reference (which denotes the instance whose method is being called) from the stack and sets *pc* to the next instruction. The method's implementing class is being located. If the call kind is

- *Constr*, the method specification denotes a constructor; its code is located in the given class. (The given method m must be `<init>`.)
- *Nonvirtual*, the method specification denotes a private method; its code is located in the current class. (The given class c must be *cclass*.)
- *Virtual*, the implementing class is looked up dynamically, starting at the class of the target reference.
- *Super*, the method is looked up dynamically, starting at the superclass of the current class. (The given class c must be $super(cclass)$.)

Once a method has been located, `invoke` calls the method: The arguments for the invoked method are placed in the local variables of the new stack frame, placing the target reference r (denoting `this` in Java) in $loc(0)$. Execution continues at the first instruction of the new method. An `instanceof` instruction pops a reference from the operand stack. If the reference is not *null* and assignment compatible with the required class, the integer 1 is pushed on the operand stack, otherwise 0 is pushed. A `checkcast` instruction checks that the top value on the stack is an instance of the given class.

If the class c of a field or method specification or if the explicitly given class c of a `new`, an `instanceof` or a `checkcast` instruction is not resolved, the JVM first resolves c, i.e. calls c's `<clinit>` method, before the instruction is executed.

$$
\begin{array}{l}
\textbf{if}\ \ (pc \text{ is } \mathtt{new}\,(c) \lor pc \text{ is } \mathtt{putfield}\,((c,\_),\_) \lor pc \text{ is } \mathtt{getfield}\,((c,\_),\_) \lor \\
\quad pc \text{ is } \mathtt{invokeinstance}\,((c,\_,\_),\_) \lor pc \text{ is } \mathtt{instanceof}\,(c) \lor pc \text{ is } \mathtt{checkcast}\,(c)) \land \\
\quad \neg resolved(res(c)) \\
\textbf{then} \\
\quad call(pc, \emptyset, opd, proc(c, \mathtt{<clinit>}))
\end{array}
$$

4.2 Compilation of $\text{Java}_\mathcal{O}$ Programs to $\text{JVM}_\mathcal{O}$ Code

Since there are no new statements in $\text{Java}_\mathcal{O}$, only the compilation of $\text{Java}_\mathcal{C}$ expressions has to be extended to the new $\text{Java}_\mathcal{O}$ expressions. The reference `this` is implemented as the distinguished local variable number 0.

$$
\begin{array}{ll}
\mathcal{E}(\mathtt{this}) & = \mathtt{load}\,(0, \mathcal{T}(\mathtt{this})) \\
\mathcal{E}(\mathtt{new}\,(c, ts)\,(e_1, \dots, e_n)) & = \mathtt{new}\,(c) \cdot \mathtt{dup}\,(c) \cdot \mathcal{E}\,e_1 \cdot \dots \cdot \mathcal{E}\,e_n \cdot \\
& \quad \mathtt{invokeinstance}\,((c, \mathtt{<init>}, (ts, \mathtt{void})), \mathit{Constr}) \\
\mathcal{E}((c, ts)\,(e_1, \dots, e_n)) & = \mathtt{load}\,(0, \mathcal{T}(\mathtt{this})) \cdot \mathcal{E}\,e_1 \cdot \dots \cdot \mathcal{E}\,e_n \cdot \\
& \quad \mathtt{invokeinstance}\,((c, \mathtt{<init>}, (ts, \mathtt{void})), \mathit{Constr}) \\
\mathcal{E}(e.\mathit{fspec}) & = \mathcal{E}\,e \cdot \mathtt{getfield}\,(\mathit{fspec}, \mathcal{T}(\mathit{fspec})) \\
\mathcal{E}(e_1.\mathit{fspec} = e_2) & = \mathcal{E}\,e_1 \cdot \mathcal{E}\,e_2 \cdot \\
& \quad \mathtt{dup\_}(\mathcal{T}(e_1), \mathcal{T}(e_2)) \cdot \mathtt{putfield}\,(\mathit{fspec}, \mathcal{T}(\mathit{fspec})) \\
\mathcal{E}(e.\mathit{mspec}\{k\}(e_1, \dots, e_n)) & = \mathcal{E}\,e \cdot \mathcal{E}\,e_1 \cdot \dots \cdot \mathcal{E}\,e_n \cdot \mathtt{invokeinstance}\,(\mathit{mspec}, k) \\
\mathcal{E}(e\;\mathtt{instanceof}\;c) & = \mathcal{E}\,e \cdot \mathtt{instanceof}\,(c) \\
\mathcal{E}((c)\,e) & = \mathcal{E}\,e \cdot \mathtt{checkcast}\,(c)
\end{array}
$$

Due to the conservativity of the extension of Java$_{\mathcal{C}}$/JVM$_{\mathcal{C}}$ to Java$_{\mathcal{O}}$/JVM$_{\mathcal{O}}$, for the proof of the *Correctness Theorem for* Java$_{\mathcal{O}}$/JVM$_{\mathcal{O}}$ it suffices to extend the theorem from Java$_{\mathcal{C}}$/JVM$_{\mathcal{C}}$ to the new expressions occurring in Java$_{\mathcal{O}}$/JVM$_{\mathcal{O}}$.

The definitions of class initialization for Java$_{\mathcal{O}}$ in [4] and resolution for JVM$_{\mathcal{O}}$ in [6] do not match because `instanceof` and class cast expressions in Java do not call the initialization of classes. In opposite, the JVM effect is to execute the initialization of the related class if it is not initialized yet. Under the assumption that also in Java these instructions trigger class initialization, these instructions preserve the theorem for Java$_{\mathcal{O}}$/JVM$_{\mathcal{O}}$.

5 JVM$_{\mathcal{E}}$ and the Compilation of Exception Treatment

In this section we extend JVM$_{\mathcal{O}}$ to JVM$_{\mathcal{E}}$ that handles exceptions. We add the compilation of the new Java$_{\mathcal{E}}$ statements and refine the compilation of jump and `return` statements.

The following grammars list the new statements of Java$_{\mathcal{E}}$ and the new JVM$_{\mathcal{E}}$ instructions. JVM$_{\mathcal{E}}$ provides instructions to raise an exception, to jump to and to return from subroutines embedded in methods.

$$
\begin{array}{llcll}
\mathit{Stm} ::= & \dots & \qquad & \mathit{Instr} ::= & \dots \\
\mid & \mathtt{throw}\;\mathit{Exp}; & & \mid & \mathtt{athrow} \\
\mid & \mathtt{try}\;\mathit{Stm}\;\mathtt{catch}\;(\mathit{Typ}, \mathit{Var}, \mathit{Stm})^* & & \mid & \mathtt{jsr}\;(\mathit{Lab}) \\
\mid & \mathtt{try}\;\mathit{Stm}\;\mathtt{finally}\;\mathit{Stm} & & \mid & \mathtt{ret}\;(\mathit{Varnum})
\end{array}
$$

5.1 The JVM$_{\mathcal{E}}$ Machine for Executing Exceptions

The JVM supports `try/catch` or `try/finally` by exception tables that list the exceptions of a method. When an exception is raised this table is searched for the handler. Exception tables refine the notion of method body as follows:

$$
\begin{array}{l}
\mathit{MethDec} == (\mathit{mKind} : \mathit{MemberKind} \times \mathit{mBody} : [\mathit{Code} \times \mathit{Exception}^*]) \\
\mathit{Exception} == (\mathit{from}, \mathit{to}, \mathit{handle} : \mathit{Lab} \times \mathit{catchTyp} : [\mathit{Class}])
\end{array}
$$

The labels *from* and *to* define the range of the protected code; *handle* starts the exception handler for the optional type *catchTyp*. If no *catchTyp* is given (as is the case for `finally` statements), any exception is caught. We refine the function *mCode* from JVM$_\mathcal{C}$ and introduce a new function *mExcs*, which returns the exceptions of the given method specification.

$$\begin{aligned}
&mCode(c, m, f) = fst(mBody(mTab(env(c))(m, f)))\\
&mExcs(c, m, f) = snd(mBody(mTab(env(c))(m, f)))
\end{aligned}$$

If a class initializer raised an exception, which is not handled within the method, Java and therefore the JVM require that the method's class must be labeled as erroneous. So we extend the domain of *ResolvedState* in the same way as we did for Java:

$$ResolvedState ::= \ldots \mid Error$$

If the thrown exception is not an `Error` or one of its subclasses, then Java$_\mathcal{E}$ and JVM$_\mathcal{E}$ throw an `ExceptionInInitializerError`. If a class should be resolved but is marked as erroneous, Java and therefore implicitely the JVM require that a `NoClassDefFoundError` is reported.

We formalize the run-time system search for a handler of an exception by a recursively defined function *catch*. This function first searches the active method using *catch'*. If no handler is found (the exception handler list is empty), the current method frame is discarded, the invoker frame is reinstated and *catch* is called recursively. A handler is found if the *pc* is protected by some brackets *from* and *to*, and the thrown exception is compatible with the *catchType*. In this case the operand stack is reduced to the exception and execution continues at the address of the exception handler. When *catch'* returns from a `<clinit>` method, the method has thrown an uncaught exception; according to the strategy presented above the method's class must be labeled as erroneous.

$$\begin{array}{l}
catch(r, ((pc \cdot pcs, loc \cdot locs, opd \cdot opds, mspec \cdot mspecs), res)) =\\
\quad catch'(mExcs(mspec))\ \mathbf{where}\\
\quad\quad catch'(\epsilon) =\\
\quad\quad\quad \mathbf{if}\ pcs = \epsilon\ \mathbf{then}\\
\quad\quad\quad\quad\quad ((undef \cdot pcs, loc \cdot locs, opd \cdot opds, mspec \cdot mspecs), res)\\
\quad\quad\quad \mathbf{else\ let}\ (c, m, _) = mspec\\
\quad\quad\quad\quad\quad\quad res' = \mathbf{if}\ m = \texttt{<clinit>}\ \mathbf{then}\ res \oplus \{(c, Error)\}\ \mathbf{else}\ res\ \mathbf{in}\\
\quad\quad\quad\quad\quad catch(r, ((pcs, locs, opds, mspecs), res'))\\
\quad\quad catch'((from, to, handle, catchTyp) \cdot excs) =\\
\quad\quad\quad \mathbf{if}\ jump(from, mCode(mspec)) \leq pc < jump(to, mCode(mspec)) \wedge\\
\quad\quad\quad\quad (catchTyp = [] \vee compatible(classOf(r), catchTyp))\ \mathbf{then}\\
\quad\quad\quad\quad\quad ((jump(handle, mCode(mspec)) \cdot pcs, loc \cdot locs, r \cdot opds, mspec \cdot mspecs), res)\\
\quad\quad\quad \mathbf{else}\ catch'(excs)
\end{array}$$

The following rules define the semantics of JVM$_\mathcal{E}$ instructions. The `athrow` instruction pops a reference from the stack and throws the exception represented by that reference. The `jsr` instruction is used to implement Java's `finally` clause. This instruction pushes the address of the next instruction on the operand stack and jumps to the given label. This requires that the universe *Pc* (called

ReturnAddress in the JVM specification) is embedded in *Word*. The address, which is put on top of the stack, is used by ret to return from the subroutine, wherefore the return address first has to be stored in a local variable.

if *pc* is **athrow** $\wedge$
  $r \cdot opd' = opd \wedge$
  $r \neq null$
then
  $(frames, res) := catch(r, (frames, res))$

if *pc* is jsr (lab)
then
  $opd := next(pc, code) \cdot opd$
  $pc := goto(lab)$

if *pc* is ret (x)
then
  $pc := loc(x)$

if $res(cclass) = Error$
then
  fail(NoClassDefFoundError)

If the current class is erroneous, the last rule throws a NoClassDefFoundError using the macro *fail*(c). This macro replaces the following instruction sequence:

new (c), **dup** , **invokeinstance** $((c,$ <init>$, (\epsilon,$ void$)), Constr)$, **athrow**

Whether or not the constructor is called is semantically irrelevant, as long as the constructors only call superclass constructors.

We refine in the obvious way rules that raise run-time exceptions. A typical representative of this rule kind is the refinement of bapply. It throws an ArithmeticException, if the operator is an integer or long division or remainder operator and the right operand is 0.

if *pc* is bapply $(\otimes) \wedge (0, v_1, opd') = split(\mathcal{A}(\otimes), opd) \wedge (\otimes \in DivMods)$
then
  fail(ArithmeticException)

JVM$_\mathcal{E}$ throws a NullPointerException if the target reference of a getfield, putfield or invokeinstance instruction is *null*, or if the reference of the athrow instruction is *null*. The machine throws a ClassCastException, if the reference on top of stack is neither *null* nor assignment compatible with the required type.

5.2 Compilation of Java$_\mathcal{E}$ Statements to JVM$_\mathcal{E}$ Instructions

Since there are no new expression in Java$_\mathcal{E}$, only the compilation of Java$_\mathcal{O}$ statements has to be extended to the compilation of the new Java$_\mathcal{E}$ statements.

For try/catch statements, the compiled try clause is followed by a jump to the end of the compiled statement. Next the handlers are generated. Each handler stores the exception into the 'catch' parameter, followed by the code of the catch clause and a jump to the end of the compiled statement. For try/finally statements s, the try clause is compiled followed by a call to the embedded subroutine, which is generated for the finally clause. The subroutine first stores the return address into a fresh variable $ret(s)$, and finally calls ret $(ret(s))$. The handler for exceptions that are thrown in the try clause starts at $lab_3(s)$. The handler saves an exception of class Throwable, which is left on the operand stack, into the fresh local variable $exc(s)$, calls the subroutine, and rethrows the exception. Variable providing functions *exc*, *ret* and also *val* that is used below, return for occurences of statements *fresh* variable numbers. This means that any returned variable number must be unused when the exception, return address or return value is stored, and this variable definition must reach its corresponding use.

$$
\begin{array}{l}
\mathcal{S}(\texttt{throw}\ e;) = \mathcal{E}e \cdot \texttt{athrow} \\
\mathcal{S}(s\ \textbf{as}\ \texttt{try}\ s_0\ \texttt{catch}\,(c_1, x_1, s_1)\ldots(c_m, x_m, s_m)) = \\
\quad \texttt{label}\,(lab_1(s)) \cdot \mathcal{S}s_0 \cdot \texttt{goto}\,(lab_3(s)) \cdot \texttt{label}\,(lab_2(s)) \cdot \\
\quad \texttt{label}\,(lab_{3+1}) \cdot \texttt{store}\,(\overline{x_1}, c_1) \cdot \mathcal{S}s_1 \cdot \texttt{goto}\,(lab_3(s)) \cdot \ldots \cdot \\
\quad \texttt{label}\,(lab_{3+m}) \cdot \texttt{store}\,(\overline{x_m}, c_m) \cdot \mathcal{S}s_m \cdot \texttt{goto}\,(lab_3(s)) \cdot \\
\quad \texttt{label}\,(lab_3(s)) \\
\mathcal{S}(s\ \textbf{as}\ \texttt{try}\ s_1\ \texttt{finally}\ s_2) = \\
\quad \texttt{label}\,(lab_1(s)) \cdot \mathcal{S}s_1 \cdot \texttt{jsr}\,(lab_2(s)) \cdot \texttt{goto}\,(lab_4(s)) \cdot \\
\quad \texttt{label}\,(lab_2(s)) \cdot \texttt{store}\,(ret(s), \texttt{ReturnAddress}) \cdot \mathcal{S}s_2 \cdot \texttt{ret}\,(ret(s)) \cdot \\
\quad \texttt{label}\,(lab_3(s)) \cdot\ \texttt{store}\,(exc(s), \texttt{Throwable}) \cdot \texttt{jsr}\,(lab_2(s)) \cdot \\
\qquad\qquad\qquad\quad \texttt{load}\,(exc(s), \texttt{Throwable}) \cdot \texttt{athrow}\ \cdot \\
\quad \texttt{label}\,(lab_4(s))
\end{array}
$$

If a jump statement is nested inside a `try` clause of a `try/finally` statement and its corresponding target statement contains `try/finally` statements, then all `finally` clauses between the jump statement and the target have to be executed in innermost order. The compilation uses the function *takeFinallyUntilTarget* : $Stm \times Lab \rightarrow Stm^*$, which given an occurrence of a statement and a label, returns in innermost order all occurrences of `try/finally` statements up to the target statement. For `return` e the compiler stores the result of the compiled expression e in a *fresh* temporary variable *val*. The compiler then generates code to jump to all outer `finally` statements in this method using the static function *takeFinally* : $Stm \rightarrow Stm^*$. Thereafter, the local variable *val* is pushed back onto the operand stack and the intended `return` instruction is executed.

$$
\begin{array}{l}
\mathcal{S}(s\ \textbf{as}\ \texttt{break}\ lab;) = \textbf{let}\ (s_1, \ldots, s_m) = takeFinallyUntilTarget(s, lab)\ \textbf{in} \\
\quad \texttt{jsr}\,(lab_2(s_1)) \cdot \ldots \cdot \texttt{jsr}\,(lab_2(s_m)) \cdot \texttt{goto}\,(lab_2(target(s, lab))) \\
\mathcal{S}(s\ \textbf{as}\ \texttt{continue}\ lab;) = \textbf{let}\ (s_1, \ldots, s_m) = takeFinallyUntilTarget(s, lab)\ \textbf{in} \\
\quad \texttt{jsr}\,(lab_2(s_1)) \cdot \ldots \cdot \texttt{jsr}\,(lab_2(s_m)) \cdot \texttt{goto}\,(lab_1(target(lab, s))) \\
\mathcal{S}(s\ \textbf{as}\ \texttt{return}\ e;) = \textbf{let}\ (s_1, \ldots, s_m) = takeFinally(s)\ \textbf{in} \\
\quad \mathcal{E}e \cdot \texttt{store}\,(val(s), \mathcal{T}(e)) \cdot \\
\quad \texttt{jsr}\,(lab_2(s_1)) \cdot \ldots \cdot \texttt{jsr}\,(lab_2(s_m)) \cdot \texttt{load}\,(val(s), \mathcal{T}(e)) \cdot \texttt{return}\,(\mathcal{T}(e)) \\
\mathcal{S}(s\ \textbf{as}\ \texttt{return};) = \textbf{let}\ (s_1, \ldots, s_m) = takeFinally(s)\ \textbf{in} \\
\quad \texttt{jsr}\,(lab_2(s_1)) \cdot \ldots \cdot \texttt{jsr}\,(lab_2(s_m)) \cdot \texttt{return}\,(\texttt{void})
\end{array}
$$

In the generation of an exception table inner `try` phrases are concatenated before the outer ones. This guarantees that exceptions are searched in innermost order.

$$
\begin{array}{ll}
\mathcal{X}(s \textbf{ as } \mathtt{try}\, s_0\, \mathtt{catch}\,(c_1, x_1, s_1), \ldots (c_m, x_m, s_m)) = & \\
\qquad\qquad \mathcal{X} s_0 \cdot (lab_1(s), lab_2(s), lab_{3+1}, c_1) \cdot \mathcal{X} s_1 \cdot \ldots \cdot & \\
\qquad\qquad\qquad (lab_1(s), lab_2(s), lab_{3+m}, c_m) \cdot \mathcal{X} s_m & \\
\mathcal{X}(s \textbf{ as } \mathtt{try}\, s_1\, \mathtt{finally}\, s_2) & = \mathcal{X} s_1 \cdot (lab_1(s), lab_2(s), lab_3(s), []) \cdot \mathcal{X} s_2 \\
\mathcal{X}(\{\, s_1 \ldots s_n \}) & = \mathcal{X} s_1 \cdot \ldots \cdot \mathcal{X} s_n \\
\mathcal{X}(\mathtt{if}\,(e)\, s_1\, \mathtt{else}\, s_2) & = \mathcal{X} s_1 \cdot \mathcal{X} s_2 \\
\mathcal{X}(\mathtt{while}\,(e)\, s) & = \mathcal{X} s \\
\mathcal{X}(lab : s) & = \mathcal{X} s \\
\mathcal{X}(\_) & = \epsilon
\end{array}
$$

If during execution of a class initializer an exception is thrown and this is not an `Error` or one of its subclasses, then Java$_{\mathcal{E}}$ and JVM$_{\mathcal{E}}$ throw an `ExceptionInInitializerError`. We refine the compilation of the phrase *Init* as follows:

$$
\begin{array}{l}
\mathcal{S}(\mathtt{static}\, s) = \\
\quad \mathcal{S}(\mathtt{try}\, s\, \mathtt{catch}\ (\mathtt{Exception}, x, \\
\qquad\qquad\qquad \mathtt{throw\ new}\,(\mathtt{ExceptionInInitializerError}, (\epsilon, \mathtt{void}))\,();))
\end{array}
$$

Due to the conservativity of the extension of Java$_{\mathcal{O}}$/JVM$_{\mathcal{O}}$ to Java$_{\mathcal{E}}$/JVM$_{\mathcal{E}}$, for the proof of the *Correctness Theorem for* Java$_{\mathcal{E}}$/JVM$_{\mathcal{E}}$ it suffices to extend the theorem from Java$_{\mathcal{O}}$/JVM$_{\mathcal{O}}$ to expression and statement execution in finally and error handling code, and to prove the following

Exception Lemma. The execution of code in Java$_{\mathcal{E}}$ and the execution of the corresponding compiled code in JVM$_{\mathcal{E}}$ produce exceptions at corresponding values of the program counters in Java$_{\mathcal{E}}$ and JVM$_{\mathcal{E}}$, for the same reasons, with the same failure classes (if any) and trigger the same exception handling.

6 Conclusion

We have presented implementation independent, rigorous yet easy to understand abstract code for the JVM as target machine for compilation of Java programs. Our definition captures faithfully the corresponding explanations of the Java Virtual Machine specification [6] and provides a practical basis for the mathematical analysis and comparison of different implementations of the machine. In particular it allowed us to prove the correctness of a general scheme for compiling Java programs into JVM code. Additionally, we have validated our work by a successful implementation in the functional programming language Haskell. The extended version of this paper [1] includes the proof details, the instruction refinement, an extensive bibliography and the discussion of related work. In an accompanying study [2] we refine the present JVM model to a defensive JVM, where we also isolate the bytecode verifier and the resolution component (including dynamic loading) of the JVM. This JVM can be used to execute compiled Java code as well as any bytecode that is loaded from the net.

Acknowledgment. We thank Ton Vullinghs for comments on this work. The first author thanks the IRIN (Institut de Recherche en Informatique de Nantes, Université de Nantes & École Centrale), in particular the *Équipe Génie logiciel, Méthodes et Spécifications formelles* for the good working environment offered during the last stage of the work on this paper.

References

[1] E. Börger and W. Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. Technical report, Universität Ulm, Fakultät für Informatik. Ulm, Germany, 1998.

[2] E. Börger and W. Schulte. A modular design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods.* Springer LNCS, to appear, 1998.

[3] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java(tm),* Springer LNCS, to appear. 1998.

[4] J. Gosling, B. Joy, and G. Steele. *The Java(tm) Language Specification.* Addison Wesley, 1996.

[5] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[6] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification.* Addison Wesley, 1996.