T U M

I N S T I T U T F Ü R I N F O R M A T I K

Formalizing the Java Virtual Machine in Isabelle/HOL

Cornelia Pusch

TUM-I9816
Juni 98

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I9816-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Formalizing the Java Virtual Machine in Isabelle/HOL

Cornelia Pusch[1]
Institut für Informatik, Technische Universität München
80290 München, Germany
`http://www.in.tum.de/~pusch`

[1]Research supported by DFG SPP *Deduktion*

Abstract

We present a formalization of the central parts of the Java Virtual Machine (JVM) with the theorem prover Isabelle/HOL. We formalize the class file format and give an operational semantics for a nontrivial subset of JVM instructions, covering the central parts of object oriented programming.

Contents

1 Introduction

The Java Virtual Machine (JVM) is an abstract machine consisting of a memory architecture and an instruction set. It is part of the Java language design developed by Sun Microsystems and serves as a basis for Java implementations. However, it also can be used as intermediate platform for other programming languages, since the JVM works independently from Java. The corresponding compiler then generates architecture-independent JVM code instead of machine code for a specific host platform. This approach allows automatic execution of compiled JVM code on any host platform that implements the JVM. However, this advantage does not come without risks. One can download any Java program from the World Wide Web and in general it is impossible to check the origin of the code and trust in its correctness. This is the reason why Java comes with several security mechanisms to protect the user from malicious code.

The Java Virtual Machine Specification [LY96] describes the operational semantics of JVM instructions as well as several static and structural constraints that have to be checked before the code may be executed. However, it is not a formal specification and it is in the nature of informal descriptions to contain ambiguities or even inconsistencies.

Our goal is to give a formal specification of the JVM that is not burdened with this problem. We think that this work can be useful in several aspects: on the one hand it allows the formal investigation of central concepts of the JVM, such as the correctness of the bytecode verifier and compiler verification; on the other hand it may serve as reference specification that is more precise than the informal description.

Formalizing a real life programming language is a very complex task and it is likely that an approach done with paper and pencil also will be susceptible to more or less grave errors. Therefore, tool assistance is required to reach a maximum amount of reliability: a theorem prover like Isabelle/HOL offers valuable support in developing consistent specifications and correct proofs.

Another important point is readability. If our specification is intended to serve as implementation guide, we have to keep it at a rather low level of abstraction. However, it is known that a high degree of abstraction simplifies the verification task. We therefore strive to find a satisfactory compromise in between.

1.1 JVM Subset

This paper presents a formalization of the JVM with Isabelle/HOL. We restrict our formalization to the central parts of object oriented programming. These include classes, interfaces, objects, methods, object fields and inheritance. We also model arrays and primitive values of type integer, but we do not treat the large amount of arithmetic instructions available in the JVM, since they are of minor interest to us. One significant feature of Java is exception handling. So far, we have included several predefined exceptions, but do not yet consider exception handling and user defined exceptions. We also do not yet treat multi-threading and synchronization.

Our runtime model of the JVM describes the operational semantics of the JVM instructions. Here, we abstract from several details, as for example the resolution of symbolic references. These aspects will be added in further refinement steps of our formalization. We also do not yet consider dynamic class loading, which revealed to introduce a problem of type safety [Sar97].

1.2 Related Work

Cohen [Coh97] has implemented in ACL2 a so called *defensive* JVM, where runtime checks are performed to guarantee a type-safe execution of the code. In contrast, our approach does not do type checking at runtime. To assure a type-safe execution, we need to check the code before execution using a bytecode verifier.

Hartel et al. [HBL98] describe the operational semantics of a Java Secure Processor (JSP), that is a derivate of the JVM designed to fit on smart cards. Their specification tool `latos` automatically generates executable code, allowing the validation of JVM programs. They do not consider bytecode verification and assume that JVM code is verified when translated to JVM code.

Qian [Qia98] gives a formal specification of the JVM instructions and describes a static type inference system. Then he proves that if a JVM program is statically well-typed, then the runtime data will be type-correct. He considers a large subset of the JVM including subroutine calls. The treatment of these instructions reveals to be the most complex during the bytecode verification process. In contrast to our work, the specification is done with paper and pencil and remains semi-formal in some parts.

Stata and Abadi [SA98] also present a type system and operational semantics for JVM instructions. They have concentrated on the formalization of subroutine calls and do not treat object orientation.

There are several efforts to formalize the Java source language. The work of Oheimb and Nipkow [NO98, ON98] is closely related to our work. They have formalized a large subset of Java (= Bali) together with its type system and operational semantics in Isabelle/HOL. The type-safety of Bali has been proved formally.

Drossopoulou and Eisenbach [DE97] have elaborated a proof on paper for the type soundness of Java, and Syme [Sym97] has formalized this work using the theorem prover DECLARE. The language subset treated in these approaches is similar to that of Bali, but the formalization differs in several aspects.

1.3 Overview

The rest of this paper is structured as follows. Section 2 gives a short overview of the theorem prover Isabelle and introduces some basic functions and types. In section 3 we introduce several datatypes for the description of the JVM class file format and the runtime environment. Section 4.2 presents the JVM instructions and defines an operational semantics for them, and section 5 summarizes the results and outlines future work.

2 Isabelle

Isabelle is a generic theorem prover that can be instantiated to different object-logics [Pau94, Isa]. The formalization described in this paper is based on the instantiation of Isabelle for higher-order logic, called Isabelle/HOL. We have chosen higher-order logic because of its expressiveness and the good proof support Isabelle offers for it.

A proof has to be conducted in the context of a theory containing the signature of all declared constants as well as a collection of definitions and axioms. Theories can be combined and extended with further signature specifications and additional definitions and axioms. Isabelle/HOL offers keywords to introduce new types, type synonyms and type classes. Non-recursive, primitive recursive and well-founded recursive function definitions can be given in special sections, assuring a conservative extension of the theory.

The basic types *bool*, *nat* and *int* are predefined, where the latter two are strictly distinguished. When n is a natural number, then *$#* n represents the corresponding integer value. The function int2nat converts a positive integer into a natural number and returns an unknown value arbitrary else. Natural numbers are constructed via 0 and the successor function Suc.

Isabelle/HOL also offers the polymorphic datatypes α *set* (with the usual set operators) and α *list*. The list constructors are [] (for the empty list) and $x\#xs$ (as 'cons' operator). We have $xs@ys$ for concatenation, map f xs to apply a function to all elements of a list, and the functions hd xs (for head), tl xs (for tail) and last xs. The function xs ! i returns the i-th list element, take n xs returns the first n list elements, drop n xs returns the rest of a list after removing n elements, and length xs computes the length of a list. rev xs reverts a list, and set xs converts a list into a set.

We have added a new function on lists to update the value of an indexed list element:

$$
\begin{aligned}
&\_[\_{:=}\_] :: [\alpha\ \mathit{list}, \mathit{nat}, \alpha] \Rightarrow \alpha\ \mathit{list}\\
&\qquad [\,][k{:=}v] = [\,]\\
&(x\#xs)[k{:=}v] = (\mathsf{case}\ k\ \mathit{of}\ \mathsf{0} \Rightarrow v\ \#\ xs \mid \mathsf{Suc}\ i \Rightarrow x\ \#\ (xs[i{:=}v]))
\end{aligned}
$$

Function types are denoted by $\tau \Rightarrow \tau'$, where $\tau_1 \Rightarrow \tau_2 \Rightarrow \ldots \Rightarrow \tau_n$ may be written as $[\tau_1,\tau_2,\ldots] \Rightarrow \tau_n$.

Optional values can be modeled using the predefined datatype

$$\alpha\ option = \mathsf{None} \mid \mathsf{Some}\ \alpha$$

It comes with an unpacking function

$$\mathsf{the} :: \alpha\ option \Rightarrow \alpha$$
$$\mathsf{the} \stackrel{\mathsf{def}}{=} \lambda y.\ \mathsf{case}\ y\ of\ \mathsf{None} \Rightarrow \mathsf{arbitrary} \mid \mathsf{Some}\ x \Rightarrow x$$

Function application is preferably written in curried style, although product types ($\alpha \times \beta$) are also available. Several definitions in our formalization are written in an uncurried style; this is due to restrictions of the TFL-package for well-founded recursive functions [Sli96, Sli97].

3 Java Virtual Machine Datatypes

This section describes the formalization of the components of a JVM class file and the JVM runtime environment.

The Java Virtual Machine Specification [LY96] describes the class file format using structured pseudocode. Several datatypes are used to represent data of different size. Array-like tables are used to store items of variable size.

In Isabelle/HOL, we model structures as product types or abstract datatypes, where we abstract from concrete data size.

3.1 An Abstract Store

The Java Virtual Machine Specification [LY96] does not require any specific implementation of the object heap. An abstract representation of a heap consists in a partial function mapping object references to object data. The Isabelle/HOL library offers a "map" type, which is defined as follows:

$$\alpha \leadsto \beta = \alpha \Rightarrow \beta\ option$$

The following functions represent the undefined function, function application, pointwise update, function merge, function composition and translation from lists into partial functions:

$$\mathsf{empty} :: \alpha \leadsto \beta$$
$$\mathsf{empty} \stackrel{\mathsf{def}}{=} \lambda x.\ \mathsf{None}$$

$$\_\ !!\ \_ :: [\alpha \leadsto \beta,\alpha] \Rightarrow \beta$$
$$t\ !!\ x \stackrel{\mathsf{def}}{=} \mathsf{the}\ (t\ x)$$

$$\_[\_ \mapsto \_] :: [\alpha \leadsto \beta, \alpha, \beta] \Rightarrow (\alpha \leadsto \beta)$$
$$m[a \mapsto b] \stackrel{\mathsf{def}}{=} \lambda x.\ \mathsf{if}\ x{=}a\ \mathsf{then\ Some}\ b\ \mathsf{else}\ m\ x$$

$$\_ \oplus \_ :: [\alpha \leadsto \beta, \alpha \leadsto \beta] \Rightarrow (\alpha \leadsto \beta)$$
$$m \oplus n \stackrel{\mathsf{def}}{=} \lambda x.\ \mathsf{case}\ n\ x\ of\ \mathsf{None} \quad \Rightarrow m\ x \mid \mathsf{Some}\ y \Rightarrow \mathsf{Some}\ y$$

$$\mathsf{map\_compose} :: [\beta \Rightarrow \gamma, \alpha \leadsto \beta] \Rightarrow (\alpha \leadsto \gamma)$$
$$\mathsf{map\_compose}\ f\ m \stackrel{\mathsf{def}}{=} \mathsf{option\_map}\ f\ (m\ k)$$

$$\mathsf{map\_of} :: (\alpha \times \beta)\mathit{list} \Rightarrow \alpha \leadsto \beta$$
$$\mathsf{map\_of}\ [] = \mathsf{empty}$$
$$\mathsf{map\_of}\ ((a,b)\#ls) = (\mathsf{map\_of}\ ls)[a \mapsto b]$$

For the representation of the object heap, we have added the function newref that returns an unused map key:

$$\mathsf{newref} :: (\alpha \leadsto \beta) \Rightarrow \alpha$$
$$\mathsf{newref}\ s \stackrel{\mathsf{def}}{=} \varepsilon v.\ s\ v = \mathsf{None}$$

In our formalization, we also use the type $\alpha \leadsto \beta$ to store the information of the field and method tables of a class file (see §3.7.3).

3.2 Identifiers

We assume a type *ident*, containing the (predefined or user-defined) class, interface, method and field names, but abstain from specifying it further; we just assume the existence of an identifier Object to refer to the predefined class of the same name. In certain contexts we also use the type synonyms *cname*, *mname* and *fname*, to make clear what kind of identifier is expected.

3.3 Field and Method Descriptors

The types of fields or methods are represented by so called *descriptors*. A field descriptor of type *field_desc* describes the type of a class or instance variable:

$$
\begin{array}{lll}
\mathit{field\_desc} & = \mathsf{I} & \text{(** integer **)} \\
 & \mid \mathsf{L}\ \mathit{cname} & \text{(** object type **)} \\
 & \mid \mathsf{A}\ \mathit{field\_desc} & \text{(** array type **)}
\end{array}
$$

The identifier *cname* of an object type can be the name of a class or an interface. This means that the type descriptor does not distinguish between class types and interface types. To get the exact type, the interface flag of the corresponding class file (see §3.7.3) has to be examined.

The parameter and return types of a method are described by a method descriptor of type *method_desc*:

$$
\begin{array}{lll}
\mathit{return\_desc} & = \mathsf{FT}\ \mathit{field\_desc} & \\
 & \mid \mathsf{V} & \text{(** void type **)} \\
\mathit{param\_desc} & = \mathit{field\_desc\ list} & \\
\mathit{method\_desc} & = \mathit{param\_desc} \times \mathit{return\_desc} &
\end{array}
$$

3.4 Field and Method References

The Java Virtual Machine Specification [LY96] does not require any particular structure for objects. In §3.8.2 we define object data as a partial mapping from field references to values. A class inherits all fields from its superclasses, where a new field declaration hides inherited fields with the same name. In certain cases, access to the hidden fields is allowed, therefore a field reference consists of the name of the defining class together with the field name:

$$\mathit{field\_loc} = \mathit{cname} \times \mathit{fname}$$

A method is referenced by its signature, that is method name and parameter descriptor. This reflects the fact that there can be several definitions for the same method name with distinct signatures. Note that the result type is not included:

$$\mathit{method\_loc} = \mathit{mname} \times \mathit{param\_desc}$$

3.5 Strings

A constant pool entry (see §3.6) contains either references to other entries or string values. We are not interested in formalizing the concrete encoding of strings; however, we must be able to distinguish whether a string represents an identifier (for a class, field or method name) or a type descriptor for a field or method. Therefore we introduce the following datatype:

$$
\begin{array}{ll}
\mathit{string} = & \mathsf{Id}\ \mathit{ident} \\
 & \mid \mathsf{Fd}\ \mathit{field\_desc} \\
 & \mid \mathsf{Md}\ \mathit{method\_desc}
\end{array}
$$

For each type constructor we define an appropriate destructor function, such that the following properties hold:

$$\mathsf{get\_Id} :: \mathit{string} \Rightarrow \mathit{ident}$$
$$\mathsf{get\_Id}\ (\mathsf{Id}\ \mathit{id}) = \mathit{id}$$

$$\mathsf{get\_Fd} :: \mathit{string} \Rightarrow \mathit{field\_desc}$$
$$\mathsf{get\_Fd}\ (\mathsf{Fd}\ \mathit{fd}) = \mathit{fd}$$

$$\mathsf{get\_Md} :: \mathit{string} \Rightarrow \mathit{method\_desc}$$
$$\mathsf{get\_Md}\ (\mathsf{Md}\ \mathit{md}) = \mathit{md}$$

If a destructor function is applied to an inappropriate constructor, it returns the value arbitrary.

Constant pool entries may contain strings representing the name of a class, array or interface name. The function type_of_str translates a string to the appropriate type descriptor:

$$
\begin{array}{ll}
\mathsf{type\_of\_str} :: \mathit{string} \Rightarrow \mathit{field\_desc} & \\
\mathsf{type\_of\_str}\ (\mathsf{Id}\ \mathit{idt}) & = \mathsf{L}\ \mathit{idt} \\
\mathsf{type\_of\_str}\ (\mathsf{Fd}\ \mathit{fd}) & = \mathsf{A}\ \mathit{fd} \\
\mathsf{type\_of\_str}\ (\mathsf{Md}\ \mathit{md}) & = \mathsf{arbitrary}
\end{array}
$$

3.6 The Constant Pool

The constant pool is part of a JVM class file (see §3.7.3). It is a kind of symbol table, containing class, field, and method references, as well as type information about fields and methods. Every entry of type *cp_info* is tagged with a keyword indicating the kind of information it stores. Constant pool entries are referenced by a numerical index. Thus we can model the constant pool as a list of *cp_info* values. For better readability we define several synonyms for a constant pool index to give an idea of what kind of entry is expected:

$$
\begin{array}{lll}
\mathit{cl\_idx} & = \mathit{nat} & \text{(** idx to Class entry **)} \\
\mathit{fr\_idx} & = \mathit{nat} & \text{(** idx to Fieldref entry **)} \\
\mathit{mr\_idx} & = \mathit{nat} & \text{(** idx to Methodref entry **)} \\
\mathit{im\_idx} & = \mathit{nat} & \text{(** idx to InterMethref entry **)} \\
\mathit{nm\_idx} & = \mathit{nat} & \text{(** idx to Utf8 string entry **)} \\
\mathit{cl\_idx} & = \mathit{nat} & \text{(** idx to Class entry **)} \\
\mathit{nt\_idx} & = \mathit{nat} & \text{(** idx to NameAndType entry **)}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{cp\_info} = & \mathsf{Class} & \mathit{nm\_idx} \\
& \mid \mathsf{Fieldref} & \mathit{cl\_idx}\ \mathit{nt\_idx} \\
& \mid \mathsf{Methodref} & \mathit{cl\_idx}\ \mathit{nt\_idx} \\
& \mid \mathsf{InterMethref} & \mathit{cl\_idx}\ \mathit{nt\_idx} \\
& \mid \mathsf{NameAndType} & \mathit{nm\_idx}\ \mathit{nm\_idx} \\
& \mid \mathsf{Utf8} & \mathit{string}
\end{array}
$$

$$
\mathit{cpool} \quad = \mathit{cp\_info}\ \mathit{list}
$$

As can be seen, a *cpool* entry may contain further references to the constant pool. For example, a Methodref entry describing a method contains one reference to the class information and another to the name and type information of the method. The constant pool entry describing the class must be a Class entry, containing again a reference to a Utf8 entry with the name of the class. The entry for the name and type information must be a

NameAndType entry, containing one reference to a Utf8 entry with the name of the method, and one to a Utf8 entry with the method descriptor. (The keyword Utf8 indicates the string encoding format used in the JVM). We define destructor functions, for which the following properties hold:

$\mathsf{get\_Class} :: \mathit{cp\_info} \Rightarrow \mathit{nm\_idx}$
$\mathsf{get\_Class}\ (\mathsf{Class}\ i) = i$

$\mathsf{get\_Fieldref} :: \mathit{cp\_info} \Rightarrow \mathit{cl\_idx} \times \mathit{nt\_idx}$
$\mathsf{get\_Fieldref}\ (\mathsf{Fieldref}\ i\ j) = (i,j)$

$\mathsf{get\_Methodref} :: \mathit{cp\_info} \Rightarrow \mathit{cl\_idx} \times \mathit{nt\_idx}$
$\mathsf{get\_Methodref}\ (\mathsf{Methodref}\ i\ i) = (i,j)$

$\mathsf{get\_InterMethref} :: \mathit{cp\_info} \Rightarrow \mathit{cl\_idx} \times \mathit{nt\_idx}$
$\mathsf{get\_InterMethref}\ (\mathsf{InterMethref}\ i\ j) = (i,j)$

$\mathsf{get\_NameAndType} :: \mathit{cp\_info} \Rightarrow \mathit{nm\_idx} \times \mathit{nm\_idx}$
$\mathsf{get\_NameAndType}\ (\mathsf{NameAndType}\ i\ j) = (i,j)$

$\mathsf{get\_Utf8} :: \mathit{cp\_info} \Rightarrow \mathit{string}$
$\mathsf{get\_Utf8}\ (\mathsf{Utf8}\ s) = s$

For a well-formed constant pool, the following functions extract data from nested constant pool references:

$$
\begin{array}{l}
\mathsf{extract\_Class} :: [\mathit{cpool},\mathit{cl\_idx}] \Rightarrow \mathit{string} \\
\mathsf{extract\_Class}\ \mathit{cp}\ \mathit{idx} \stackrel{\mathsf{def}}{=} \\
\quad (\mathsf{let}\ \mathit{n\_idx} = \mathsf{get\_Class}\ (\mathit{cp}\ !\ \mathit{idx}); \\
\qquad\quad \mathit{cstr} \quad = \mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{nm\_idx}) \\
\quad\ \mathsf{in} \\
\quad\ \mathit{cstr})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{extract\_Fieldref} :: [\mathit{cpool},\mathit{fr\_idx}] \Rightarrow (\mathit{cname} \times \mathit{fname} \times \mathit{field\_desc}) \\
\mathsf{extract\_Fieldref}\ \mathit{cp}\ \mathit{idx} \stackrel{\mathsf{def}}{=} \\
\quad (\mathsf{let}\ (\mathit{c\_idx},\mathit{nt\_idx}) \quad = \mathsf{get\_Fieldref}\ (\mathit{cp}\ !\ \mathit{idx}); \\
\qquad\quad \mathit{cid} \qquad\qquad\quad = \mathsf{get\_Id}\ (\mathsf{extract\_Class}\ \mathit{cp}\ \mathit{c\_idx}); \\
\qquad\quad (\mathit{n\_idx},\mathit{d\_idx}) \quad = \mathsf{get\_NameAndType}\ (\mathit{cp}\ !\ \mathit{nt\_idx}); \\
\qquad\quad \mathit{fid} \qquad\qquad\quad = \mathsf{get\_Id}\ (\mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{n\_idx})); \\
\qquad\quad \mathit{fd} \qquad\qquad\quad\ = \mathsf{get\_Fd}\ (\mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{d\_idx})) \\
\quad\ \mathsf{in} \\
\quad\ (\mathit{cid},\mathit{fid},\mathit{fd}))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{extract\_Methodref} :: [\mathit{cpool}, \mathit{mr\_idx}] \Rightarrow (\mathit{cname} \times \mathit{mname} \times \mathit{method\_desc}) \\
\mathsf{extract\_Methodref}\ \mathit{cp}\ \mathit{idx} \stackrel{\mathsf{def}}{=} \\
\quad (\mathsf{let}\ \begin{array}[t]{ll}
(\mathit{c\_idx}, \mathit{nt\_idx}) & = \mathsf{get\_Methodref}\ (\mathit{cp}\ !\ \mathit{idx}); \\
\mathit{cid} & = \mathsf{get\_Id}\ (\mathsf{extract\_Class}\ \mathit{cp}\ \mathit{c\_idx}); \\
(\mathit{n\_idx}, \mathit{d\_idx}) & = \mathsf{get\_NameAndType}\ (\mathit{cp}\ !\ \mathit{nt\_idx}); \\
\mathit{mid} & = \mathsf{get\_Id}\ (\mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{n\_idx})); \\
\mathit{md} & = \mathsf{get\_Md}\ (\mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{d\_idx}))
\end{array} \\
\quad\ \mathsf{in} \\
\quad\ (\mathit{cid}, \mathit{mid}, \mathit{md}))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{extract\_InterMethref} :: [\mathit{cpool}, \mathit{im\_idx}] \Rightarrow (\mathit{cname} \times \mathit{mname} \times \mathit{method\_desc}) \\
\mathsf{extract\_InterMethref}\ \mathit{cp}\ \mathit{idx} \stackrel{\mathsf{def}}{=} \\
\quad (\mathsf{let}\ \begin{array}[t]{ll}
(\mathit{c\_idx}, \mathit{nt\_idx}) & = \mathsf{get\_InterMethref}\ (\mathit{cp}\ !\ \mathit{idx}); \\
\mathit{cid} & = \mathsf{get\_Id}\ (\mathsf{extract\_Class}\ \mathit{cp}\ \mathit{c\_idx}); \\
(\mathit{n\_idx}, \mathit{d\_idx}) & = \mathsf{get\_NameAndType}\ (\mathit{cp}\ !\ \mathit{nt\_idx}); \\
\mathit{mid} & = \mathsf{get\_Id}\ (\mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{n\_idx})); \\
\mathit{md} & = \mathsf{get\_Md}\ (\mathsf{get\_Utf8}\ (\mathit{cp}\ !\ \mathit{d\_idx}))
\end{array} \\
\quad\ \mathsf{in} \\
\quad\ (\mathit{cid}, \mathit{mid}, \mathit{md}))
\end{array}
$$

3.7 The Class File Format

The binary code generated by a Java compiler comes in a special format, called *class file*. This format is defined precisely in the Java Virtual Machine Specification [LY96], being the basis for class file and bytecode verification to detect ill-formed JVM programs.

To keep our first formalization small, we have omitted several components that do not concern the execution of the code directly (but would be important when considering class file verification). These are for example magic number or version number. Some sections like attributes or exceptions handling do not occur, because we do not yet consider these parts of the language.

3.7.1 Fields

Each field is described by a *field_info* entry, containing two pointers to the constant pool. The first one references the field name, the second one the field descriptor. Fields are modeled as a list of field data:

$$
\begin{array}{lll}
\mathit{field\_info} = \mathit{nm\_idx} \times & & \text{(** idx to Utf8 } \langle\text{field name}\rangle \text{ **)} \\
\qquad\qquad\quad \mathit{nm\_idx} & & \text{(** idx to Utf8 } \langle\text{field desc.}\rangle \text{ **)} \\
\mathit{fields} \quad\ = \mathit{field\_info\ list} & &
\end{array}
$$

3.7.2 Methods

Each method is described by a *method_info* entry, containing two pointers to the constant pool and the method code. The first pointer references the method name, the second one the method descriptor. Methods are modeled as a list of method data:

$$
\begin{array}{lll}
\textit{'instr method\_info} = & \textit{nm\_idx} \times & \text{(** idx to Utf8 } \langle\text{meth.name}\rangle \text{ **)} \\
 & \textit{nm\_idx} \times & \text{(** idx to Utf8 } \langle\text{meth.desc.}\rangle \text{ **)} \\
 & \textit{'instr list} & \\
\textit{'instr methods} = & (\textit{'instr method\_info})\ \textit{list} &
\end{array}
$$

The given types are parameterized over a type variable *'instr* that will be instantiated later. This allows us to formalize the JVM instruction set and its operational semantics in a modular way (see §4.2).

3.7.3 The Class File

A class file consists of a constant pool, a flag indicating whether the class file describes an interface or a class, pointers to constant pool entries returning the names of the current class, its superclass and direct superinterfaces, and the field and method descriptions:

$$
\begin{array}{lll}
\textit{'instr classfile} = & \textit{cpool} \times & \text{(** constant pool **)} \\
 & \textit{bool} \times & \text{(** is it an interface ? **)} \\
 & \textit{cl\_idx} \times & \text{(** idx to current class **)} \\
 & \textit{cl\_idx} \times & \text{(** idx to direct superclass **)} \\
 & \textit{cl\_idx list} \times & \text{(** idxs to direct superints **)} \\
 & \textit{fields} \times & \text{(** field table **)} \\
 & \textit{'instr methods} & \text{(** method table **)}
\end{array}
$$

We define selector functions to access the individual parts of a class file. Thus we can easily extend our formalization with further components without having to change much code:

$$
\begin{array}{l}
\mathsf{get\_cpool} :: \textit{'instr classfile} \Rightarrow \textit{cpool} \\
\mathsf{get\_cpool} \overset{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms).\ cp
\end{array}
$$

$$
\begin{array}{l}
\mathsf{is\_interface} :: \textit{'instr classfile} \Rightarrow \textit{bool} \\
\mathsf{is\_interface} \overset{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms).\ a
\end{array}
$$

$$
\begin{array}{l}
\mathsf{get\_thisclass} :: \textit{'instr classfile} \Rightarrow \textit{cname} \\
\mathsf{get\_thisclass} \overset{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms).\ \mathsf{get\_Id}\ (\mathsf{extract\_Class}\ cp\ t)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{get\_superclass} :: \textit{'instr classfile} \Rightarrow \textit{cname} \\
\mathsf{get\_superclass} \overset{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms).\ \mathsf{get\_Id}\ (\mathsf{extract\_Class}\ cp\ s)
\end{array}
$$

If the class file describes a class, the list of direct superinterfaces contains the superinterfaces that are directly implemented by that class. If the class file contains an interface description, that interface extends the given superinterfaces. The function get_superinterfaces returns the set of referenced interface names:

$$\begin{array}{l}
\mathsf{get_superinterfaces} :: \textit{'instr classfile} \Rightarrow \textit{cname set} \\
\mathsf{get_superinterfaces} \stackrel{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms). \\
\quad \mathsf{set}\ (\mathsf{map}\ (\mathsf{get_Id} \circ (\mathsf{extract_Class}\ cp))\ is)
\end{array}$$

The selector functions for methods and fields convert the lists of method and field data to mappings. This makes method and field access more comfortable:

$$\begin{array}{l}
\mathsf{get_methods} :: \textit{'instr classfile} \Rightarrow (\textit{method_loc} \rightsquigarrow \textit{return_desc} \times \textit{'instr list}) \\
\mathsf{get_methods} \stackrel{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms). \\
\quad \mathsf{map_of}\ (\mathsf{map}\ (\lambda(mn,md,ins). \\
\qquad\qquad \mathsf{let}\ (pd,rd) = \mathsf{get_Md}\ (\mathsf{get_Utf8}\ (cp\ !\ md)) \\
\qquad\qquad \mathsf{in} \\
\qquad\qquad ((\mathsf{get_Id}\ (\mathsf{get_Utf8}\ (cp\ !\ mn)),\ pd)\ ,\ (rd,ins))) \\
\qquad ms)
\end{array}$$

$$\begin{array}{l}
\mathsf{get_fields} :: \textit{'instr classfile} \Rightarrow (\textit{field_loc} \rightsquigarrow \textit{field_desc}) \\
\mathsf{get_fields} \stackrel{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms). \\
\quad \mathsf{map_of}\ (\mathsf{map}\ (\lambda(fn,fd). \\
\qquad\qquad ((\mathsf{get_Id}\ (\mathsf{extract_Class}\ cp\ t),\ \mathsf{get_Id}\ (\mathsf{get_Utf8}\ (cp\ !\ fn))), \\
\qquad\qquad \mathsf{get_Fd}\ (\mathsf{get_Utf8}\ (cp\ !\ fd)))) \\
\qquad fs)
\end{array}$$

To extract the code area of a method, we use the following function:

$$\begin{array}{l}
\mathsf{get_code} :: [\textit{'instr classfile}, \textit{method_loc}] \Rightarrow \textit{'instr list} \\
\mathsf{get_code}\ cf\ mid \stackrel{\mathsf{def}}{=} \mathsf{let}\ (rd,ins) = (\mathsf{get_methods}\ cf)\ !!\ mid\ \mathsf{in}\ ins
\end{array}$$

The current class does not have a superclass, if the constant pool reference for the superclass is zero. This is checked by the following predicate:

$$\begin{array}{l}
\mathsf{no_super} :: \textit{'instr classfile} \Rightarrow \textit{bool} \\
\mathsf{no_super} \stackrel{\mathsf{def}}{=} \lambda(cp,a,t,s,is,fs,ms).\ s{=}0
\end{array}$$

The JVM works on a set of class files. In our formalisation, we represent this by a mapping from class names to class files:

$$\textit{'instr classfiles} = \textit{cname} \rightsquigarrow \textit{'instr classfile}$$

3.7.4 Class Relations

The Java concept defines several relations between classes and interfaces that are formalized in this subsection.

First of all, it must be checked, whether an identifier represents a class or an interface. In both cases, there must exist a corresponding class file, with the interface flag set appropriately:

$$\mathsf{is\_class} :: [\textit{'instr classfiles}, \textit{cname}] \Rightarrow \textit{bool}$$
$$\mathsf{is\_class}\ \textit{CFS}\ \textit{cn} \stackrel{\mathsf{def}}{=} \textit{CFS}\ \textit{cn} \neq \mathsf{None} \wedge \neg\mathsf{is\_interface}\ (\textit{CFS}\ !!\ \textit{cn})$$

$$\mathsf{is\_inter} :: [\textit{'instr classfiles}, \textit{cname}, \textit{cname}] \Rightarrow \textit{bool}$$
$$\mathsf{is\_inter}\ \textit{CFS}\ \textit{cn} \stackrel{\mathsf{def}}{=} \textit{CFS}\ \textit{cn} \neq \mathsf{None} \wedge \mathsf{is\_interface}\ (\textit{CFS}\ !!\ \textit{cn})$$

The direct superclass relation is then defined as follows:

$$\mathsf{d\_superclass\_rel} :: \textit{'instr classfiles} \Rightarrow (\textit{cname} \times \textit{cname})\ \textit{set}$$
$$\begin{aligned}&\mathsf{d\_superclass\_rel}\ \textit{CFS} \stackrel{\mathsf{def}}{=}\\ &\quad \{(sc,cn).\ \mathsf{is\_class}\ \textit{CFS}\ sc \wedge \mathsf{is\_class}\ \textit{CFS}\ cn \wedge cn \neq \mathsf{Object} \wedge\\ &\qquad\qquad \mathsf{get\_superclass}\ (\textit{CFS}\ !!\ cn) = sc\}\end{aligned}$$

We do not formalize an extra subclass relation, since it is just the transitive closure of the inverted direct superclass relation. Hence, a class cn is a subclass of class sc, if $(sc,cn) \in (\mathsf{d\_superclass\_rel}\ \textit{CFS})^+$ holds. In some cases, we also use the reflexive transitive closure $(\mathsf{d\_superclass\_rel}\ \textit{CFS})^*$.

The definition of the direct superinterface relation between two interfaces is analogous:

$$\mathsf{d\_superinterface\_rel} :: \textit{'instr classfiles} \Rightarrow (\textit{cname} \times \textit{cname})\ \textit{set}$$
$$\begin{aligned}&\mathsf{d\_superinterface\_rel}\ \textit{CFS} \stackrel{\mathsf{def}}{=}\\ &\quad \{(si,i).\ \mathsf{is\_inter}\ \textit{CFS}\ si\ \wedge \mathsf{is\_inter}\ \textit{CFS}\ i \wedge\\ &\qquad\quad si \in (\mathsf{get\_superinterfaces}\ (\textit{CFS}\ !!\ i))\}\end{aligned}$$

The general (non-direct) superinterface relation is obtained by the transitive closure $(\mathsf{d\_superinterface\_rel}\ \textit{CFS})^+$.

The implementation relation between a class and an interface is more complex: a class cn implements an interface si if cn directly implements sc, or if cn directly implements an interface si' that has si as superinterface, or if the direct superclass sc implements si. This recursive definition is only well-defined, if there are no cyclic superclass relations. This is checked by the predicate WF_classfiles described below. Technically, we have defined the function implements using well-founded recursion and proved that the following function equation holds under the given condition:

$\mathsf{d\_implements} :: [\textit{'instr classfiles}, \textit{cname}, \textit{cname}] \Rightarrow \textit{bool}$
$\mathsf{d\_implements}\ \mathit{CFS}\ \mathit{cn}\ \mathit{si} \stackrel{\mathsf{def}}{=}$
$\quad \mathsf{is\_class}\ \mathit{CFS}\ \mathit{cn} \wedge \mathsf{is\_inter}\ \mathit{CFS}\ \mathit{si} \wedge \mathit{si} \in \mathsf{get\_superinterfaces}\ (\mathit{CFS}\ !!\ \mathit{cn})$

$\mathsf{implements} :: \textit{'instr classfiles} \times \mathit{cname} \times \mathit{cname} \Rightarrow \mathit{bool}$
$\mathsf{WF\_classfiles}\ \mathit{CFS}\ \longrightarrow$
$\mathsf{implements}\ (\mathit{CFS},\ \mathit{cn},\ i) =$
$\quad (\mathsf{d\_implements}\ \mathit{CFS}\ \mathit{cn}\ i) \vee$
$\quad (\exists \mathit{si}.\ \mathsf{d\_implements}\ \mathit{CFS}\ \mathit{cn}\ \mathit{si} \wedge (i,\mathit{si}) \in (\mathsf{d\_superinterface\_rel}\ \mathit{CFS})^{+}) \vee$
$\quad (\exists \mathit{sc}.\ (\mathit{sc},\mathit{cn}) \in \mathsf{d\_superclass\_rel}\ \mathit{CFS} \wedge \mathsf{implements}\ (\mathit{CFS},\ \mathit{sc},\ i))$

3.7.5 Well-formedness of Class Files

Correct machine programs (i.e. sets of class files) have to conform to several syntactic and structural constraints that are checked by the JVM before execution. Well-formed class files will satisfy the following properties: the number of considered class files is finite; the class Object is defined and it does have neither superclass nor fields; for all defined class files, the constant pool entry for the current class contains the correct class name; further properties depend on whether the interface flag is set or not, i.e. whether the class file represents a class or an interface:

$\mathsf{WF\_classfiles} :: \textit{'instr classfiles} \Rightarrow \mathit{bool}$
$\mathsf{WF\_classfiles}\ \mathit{CFS} \stackrel{\mathsf{def}}{=}$
$\quad (\mathsf{finite}\ \{\mathit{cn}.\ \mathit{CFS}\ \mathit{cn} \neq \mathsf{None}\}) \wedge$
$\quad (\mathsf{is\_class}\ \mathit{CFS}\ \mathsf{Object} \wedge \mathsf{no\_super}\ (\mathit{CFS}\ !!\ \mathsf{Object})) \wedge$
$\quad (\mathsf{get\_fields}\ (\mathit{CFS}\ !!\ \mathsf{Object}) = \mathsf{empty}) \wedge$
$\quad (\forall \mathit{cn}.\ \mathit{CFS}\ \mathit{cn} \neq \mathsf{None}$
$\qquad\qquad \longrightarrow (\mathsf{get\_thisclass}\ (\mathit{CFS}\ !!\ \mathit{cn}) = \mathit{cn}) \wedge$
$\qquad\qquad\quad (\mathsf{is\_inter}\ \mathit{CFS}\ \mathit{cn} \longrightarrow \mathsf{WF\_inter}\ \mathit{CFS}\ \mathit{cn}) \wedge$
$\qquad\qquad\quad (\mathsf{is\_class}\ \mathit{CFS}\ \mathit{cn} \longrightarrow \mathsf{WF\_class}\ \mathit{CFS}\ \mathit{cn}))$

The class file of a well-formed class fulfills the following conditions: if the current class does not have a superclass, its name is Object. In that case, the class does not have any superinterfaces. Otherwise, superclass and superinterfaces are also defined, and there are no cyclic dependencies. Correct method overriding and implementation is checked by the predicates method_over_ok and method_impl_ok that are described below:

$$
\begin{array}{l}
\mathsf{WF\_class} :: [\textit{'instr classfiles,cname}] \Rightarrow \textit{bool} \\
\mathsf{WF\_class}\ \textit{CFS cn} \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if\ no\_super}\ (\textit{CFS}\ !!\ \textit{cn}) \\
\quad\quad \mathsf{then}\ \textit{cn}{=}\mathsf{Object} \wedge \\
\quad\quad\quad\quad \mathsf{get\_superinterfaces}\ (\textit{CFS}\ !!\ \textit{cn}){=}\{\} \\
\quad\quad \mathsf{else\ let}\ \textit{sc} = \mathsf{get\_superclass}\ (\textit{CFS}\ !!\ \textit{cn}) \\
\quad\quad\quad\quad \mathsf{in} \\
\quad\quad\quad\quad \mathsf{is\_class}\ \textit{CFS sc} \wedge \\
\quad\quad\quad\quad\ (\textit{cn,sc}) \notin (\mathsf{d\_superclass\_rel}\ \textit{CFS})^* \wedge \\
\quad\quad\quad\quad\ (\forall \textit{sc'}.\ (\textit{sc',cn}) \in (\mathsf{d\_superclass\_rel}\ \textit{CFS})^+ \\
\quad\quad\quad\quad\quad\quad\quad\quad \longrightarrow \mathsf{method\_over\_ok}\ \textit{CFS cn sc'}) \wedge \\
\quad\quad\quad\quad\ (\forall \textit{si} \in \mathsf{get\_superinterfaces}\ (\textit{CFS}\ !!\ \textit{cn}). \\
\quad\quad\quad\quad\quad\quad \mathsf{is\_inter}\ \textit{CFS si} \wedge \\
\quad\quad\quad\quad\quad\quad (\forall \textit{si'}.\ (\textit{si',si}) \in (\mathsf{d\_superinterface\_rel}\ \textit{CFS})^* \\
\quad\quad\quad\quad\quad\quad\quad\quad \longrightarrow \mathsf{method\_impl\_ok}\ \textit{CFS cn si'}))
\end{array}
$$

The class file of a proper interface refers to the class Object as superclass. All superinterfaces have to be defined, where there must not be cyclic dependencies:

$$
\begin{array}{l}
\mathsf{WF\_inter} :: [\textit{'instr classfiles,cname}] \Rightarrow \textit{bool} \\
\mathsf{WF\_inter}\ \textit{CFS i} \stackrel{\mathsf{def}}{=} \\
\ (\mathsf{get\_superclass}\ (\textit{CFS}\ !!\ \textit{i}) = \mathsf{Object}) \wedge \\
\ (\forall \textit{si} \in \mathsf{get\_superinterfaces}\ (\textit{CFS}\ !!\ \textit{i}). \\
\quad\quad \mathsf{is\_inter}\ \textit{CFS si} \wedge \\
\quad\quad (\textit{i,si}) \notin (\mathsf{d\_superinterface\_rel}\ \textit{CFS})^*)
\end{array}
$$

We do not require explicitely correct method overriding for interfaces, since this can be derived from correct interface implementation[1].

If a method defined in some superclass *sc* is overridden in the current class *cn*, the return descriptor must remain the same:

$$
\begin{array}{l}
\mathsf{method\_over\_ok} :: [\textit{'instr classfiles,cname,cname}] \Rightarrow \textit{bool} \\
\mathsf{method\_over\_ok}\ \textit{CFS cn sc} \stackrel{\mathsf{def}}{=} \\
\ \ \forall \textit{ml rd ins rd' ins'}. \\
\quad\quad \mathsf{get\_methods}\ (\textit{CFS}\ !!\ \textit{cn})\ \textit{ml} = \mathsf{Some}\ (\textit{rd}\ ,\textit{ins}\ ) \wedge \\
\quad\quad \mathsf{get\_methods}\ (\textit{CFS}\ !!\ \textit{sc})\ \textit{ml} = \mathsf{Some}\ (\textit{rd'},\textit{ins'}) \\
\quad\quad\quad \longrightarrow \textit{rd}{=}\textit{rd'}
\end{array}
$$

If a method is declared in some superinterface *si*, the implementing class *cn* must contain a method of the appropriate type. This means, that there

[1] The Java Virtual Machine Specification [LY96] does not talk about correct method overriding for interfaces. At a later point, we found that the Java Language Specification [GJS96] explicitly requires this. However, we have kept our formalization, since the given predicates suffice to derive correct method overriding.

must be a method definition in the class or some superclass[2]:

$$
\begin{array}{l}
\mathsf{method\_impl\_ok} :: [\textit{'instr classfiles,cname,cname}] \Rightarrow \textit{bool} \\
\mathsf{method\_impl\_ok}\ \textit{CFS cn si} \stackrel{\mathsf{def}}{=} \\
\quad \forall \textit{ml rd ins}. \\
\qquad\quad \mathsf{get\_methods}\ (\textit{CFS} \,!!\, \textit{si})\ \textit{ml} = \mathsf{Some}\ (\textit{rd,ins}) \\
\qquad\qquad \longrightarrow (\exists \textit{cn' ins'}.\ (\textit{cn',cn}) \in (\mathsf{d\_superclass\_rel}\ \textit{CFS})^* \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathsf{get\_methods}\ (\textit{CFS} \,!!\, \textit{cn'})\ \textit{ml} = \mathsf{Some}\ (\textit{rd,ins'}))
\end{array}
$$

3.8 The JVM Runtime Environment

3.8.1 JVM Runtime Data

Like the Java language, the JVM operates on two different types of values, primitive values and reference values. Among the primitive values, we consider only those of type integer. The reference values are pointers to objects, to denote a null pointer there exists a special null reference.

The realization of object references is kept abstract: we model them by an opaque type *loc* that is not further specified. We formalize JVM values as follows:

$$
\begin{array}{l}
\textit{val} = \mathsf{Intg}\ \textit{int} \\
\qquad \mid \mathsf{Addr}\ \textit{loc} \\
\qquad \mid \mathsf{Null}
\end{array}
$$

The destructor functions have the following properties:

$$
\begin{array}{l}
\mathsf{get\_Intg} :: \textit{val} \Rightarrow \textit{int} \\
\mathsf{get\_Intg}\ (\mathsf{Intg}\ i) = i \\
\\
\mathsf{get\_Addr} :: \textit{val} \Rightarrow \textit{loc} \\
\mathsf{get\_Addr}\ (\mathsf{Addr}\ l) = l
\end{array}
$$

Additionally, we define:

$$
\begin{array}{l}
\mathsf{get\_Nat} :: \textit{val} \Rightarrow \textit{nat} \\
\mathsf{get\_Nat}\ x \stackrel{\mathsf{def}}{=} \mathsf{int2nat}\ (\mathsf{get\_Intg}\ x)
\end{array}
$$

The Java Virtual Machine Specification [LY96] does not require values to be tagged with their runtime types, whereas in our formalization, the constructors Addr and Intg contains additional type information (Null is indeed a distinct value). We need this information to state and prove the correctness of the bytecode verifier, where the runtime types are checked against the

[2] Again, the description in the Java Virtual Machine Specification [LY96] might lead to unintentional interpretations: it is not made clear that the method implementation also may be inherited from a superclass

static type information. However, our approach does not impose restrictions on possible implementations, because the values are only accessed via the destructor functions. Thus, the type tags do not influence the operational semantics of a statically well-typed JVM program. The type information simply may be thrown away in a concrete implementation.

3.8.2 The JVM Heap

The heap contains the runtime data of all objects and arrays. The Java Virtual Machine Specification [LY96] does not require any specific representation of objects. In our formalization, an object consists of the corresponding class name and a data area. Object data is represented by a mapping from field references to values. An array consists of the type descriptor for its components and a data area. Array data is represented as a list that can be accessed by a numerical index. The heap is then modeled by a mapping from object references to objects.

$$
\begin{array}{lll}
\mathit{odata} & = & \mathit{field\_loc} \leadsto \mathit{val} \\
\mathit{adata} & = & \mathit{val\ list} \\
\mathit{obj} & = & \mathsf{Obj}\ \mathit{cname}\ \mathit{odata} \\
 & \mid & \mathsf{Arr}\ \mathit{field\_desc}\ \mathit{adata} \\
\mathit{heap} & = & \mathit{loc} \leadsto \mathit{obj}
\end{array}
$$

The characteristic properties of the destructor functions are

$$
\begin{array}{l}
\mathsf{get\_Obj} :: \mathit{obj} \Rightarrow (\mathit{cname} \times \mathit{odata}) \\
\mathsf{get\_Obj}\ (\mathsf{Obj}\ \mathit{cn}\ \mathit{od}) = (\mathit{cn},\mathit{od})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{get\_Arr} :: \mathit{obj} \Rightarrow (\mathit{field\_desc} \times \mathit{adata}) \\
\mathsf{get\_Arr}\ (\mathsf{Arr}\ \mathit{fd}\ \mathit{ad}) = (\mathit{fd},\mathit{ad})
\end{array}
$$

A class instance contains the name of its class, and an array contains the type of its components. To get the type of the whole object, we define the following function:

$$
\begin{array}{l}
\mathsf{get\_obj\_type} :: \mathit{obj} \Rightarrow \mathit{field\_desc} \\
\mathsf{get\_obj\_type}\ (\mathsf{Obj}\ \mathit{cn}\ \mathit{od}) = \mathsf{L}\ \mathit{cn} \\
\mathsf{get\_obj\_type}\ (\mathsf{Arr}\ \mathit{fd}\ \mathit{ad})\ \ = \mathsf{A}\ \mathit{fd}
\end{array}
$$

In the course of instance method invocation (see §4.11), the called method depends on the runtime type of the current object. In the case of a class instance, the search for the method starts in the class file of that class, in the case of an array the class Object is inspected. The function get_obj_class returns the name of the class, where search starts:

$$
\begin{array}{l}
\mathsf{get\_obj\_class} :: \mathit{obj} \Rightarrow \mathit{cname} \\
\mathsf{get\_obj\_class}\ (\mathsf{Obj}\ \mathit{cn}\ \mathit{od}) = \mathit{cn} \\
\mathsf{get\_obj\_class}\ (\mathsf{Arr}\ \mathit{fd}\ \mathit{ad})\ \ = \mathsf{Object}
\end{array}
$$

3.8.3 Object Initialization

If a new class instance is created (see §4.5), it will contain data for all fields, those that have been declared in the corresponding class file, and those of all superclasses. The function get_all_fields successively merges the fields of all superclasses with the fields of the current class file. Again, the function is defined by well-founded recursion. It has the following property that has been proved as a theorem from its definition:

$$
\begin{array}{l}
\mathsf{get\_all\_fields} :: \mathit{'instr\ classfiles} \times \mathit{cname} \Rightarrow \mathit{field\_loc} \leadsto \mathit{field\_desc} \\
\mathsf{WF\_classfiles}\ \mathit{CFS} \wedge \mathsf{is\_class}\ \mathit{CFS}\ \mathit{cn} \ \longrightarrow \\
\mathsf{get\_all\_fields}\ (\mathit{CFS,cn}) = \\
\quad (\mathsf{let}\ \mathit{fs} = \mathsf{get\_fields}\ (\mathit{CFS}\ !!\ \mathit{cn}) \\
\quad\ \ \mathsf{in} \\
\quad\ \ \mathsf{if}\ \mathit{cn}{=}\mathsf{Object}\ \mathsf{then}\ \mathit{fs} \\
\qquad \mathsf{else}\ (\mathsf{let}\ \mathit{sc} = \mathsf{get\_superclass}\ (\mathit{CFS}\ !!\ \mathit{cn}) \\
\qquad\qquad \mathsf{in} \\
\qquad\qquad (\mathsf{get\_all\_fields}\ (\mathit{CFS},\ \mathit{sc}))\ \oplus\ \mathit{fs}))
\end{array}
$$

The object and array fields are initialized to their default values:

$$
\begin{array}{l}
\mathsf{default\_val} :: \mathit{field\_desc} \Rightarrow \mathit{val} \\
\mathsf{default\_val}\quad \mathsf{I}\ = \mathsf{Intg}\ (\$\#0) \\
\mathsf{default\_val}\ (\mathsf{L}\ i) = \mathsf{Null} \\
\mathsf{default\_val}\ (\mathsf{A}\ f) = \mathsf{Null}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{init\_obj} :: [\mathit{'instr\ classfiles,cname}] \Rightarrow \mathit{odata} \\
\mathsf{init\_obj}\ \mathit{CFS}\ \mathit{cn} \stackrel{\mathsf{def}}{=} \\
\ (\mathsf{let}\ \mathit{fs} = \mathsf{get\_all\_fields}\ \mathit{CFS}\ \mathit{cn}\ \mathsf{in}\ \mathsf{map\_compose}\ \mathsf{default\_val}\ \mathit{fs})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{init\_arr} :: [\mathit{val,int}] \Rightarrow \mathit{adata} \\
\mathsf{init\_arr}\ \mathit{val}\ n \stackrel{\mathsf{def}}{=} \mathsf{replicate}\ (\mathsf{int2nat}\ n)\ \mathit{val}
\end{array}
$$

The function replicate n v creates a list $[v,\ \ldots\ ,v]$ of length n.

3.8.4 The JVM Frame Stack

Each time a method is invoked, a new frame is created on the frame stack containing the following components: the operand stack *opstack* is used to store partial results and arguments of further instructions, the local variables *locvars* contain the arguments the method has been called with, and a reference to the object the method has been called on. The class name *cname* indicates the defining class of the method, and *meth_loc* gives a (symbolic) reference to the method code within this class. The program counter *p_count* points to the instruction in the method code to be executed next. In our formalization, the program counter is local to the code area of a method, that means, it just determines the offset in the method code.

$$
\begin{array}{lll}
opstack & = val\ list & \\
locvars & = val\ list & \\
p\_count & = nat & \\
frame & = opstack \times & \text{(** operand stack **)} \\
 & \quad locvars \times & \text{(** local variables **)} \\
 & \quad cname \times & \text{(** defining class **)} \\
 & \quad meth\_loc \times & \text{(** ref. to code **)} \\
 & \quad p\_count & \text{(** program counter **)}
\end{array}
$$

3.8.5 The Exception Flag

During the execution of the JVM, several exceptions can occur. We consider a set of predefined exceptions, but do not allow user defined exceptions.

$$
\begin{array}{ll}
xcpt = & \mathsf{NullPointer} \\
 & \mid \mathsf{NegArrSize} \\
 & \mid \mathsf{IndOutBound} \\
 & \mid \mathsf{ArrStore} \\
 & \mid \mathsf{ClassCast} \\
 & \mid \mathsf{InstantiationError} \\
 & \mid \mathsf{OutOfMemory}
\end{array}
$$

Since we do not yet treat exception handling, execution stops immediately once an exception is thrown.

3.8.6 The JVM Runtime State

The runtime state of the JVM is formed by the following components:

$$
\begin{array}{lll}
jvm\_state = & xcpt\ option \times & \text{(** exception flag **)} \\
 & heap \times & \text{(** object heap **)} \\
 & frame\ list & \text{(** frame stack **)}
\end{array}
$$

3.8.7 Type Compatibility

Casting the type of a value to a another type requires valid type conversions. Sometimes a cast can be checked at compile time, but in general exhaustive type checking cannot be done before runtime. The compiler will then introduce a cast checking instruction (see §4.8) that throws a runtime exception if the type does not fit. The predicate $\mathsf{compatible}\ s\ t$ checks, if type s can be cast to t:

```
compatible :: ['instr classfiles,field_desc,field_desc] ⇒ bool
compatible CFS I t = (t=I)
compatible CFS (L cn) t =
  (case t of I       ⇒ False
           | (L cn') ⇒ if is_interface (CFS !! cn')
                        then implements (CFS,cn,cn')
                        else (cn',cn) ∈ (d_superclass_rel CFS)*
           | (A fd)  ⇒ False)
compatible CFS (A fd) t =
  (case t of I       ⇒ False
           | (L cn)  ⇒ cn=Object
           | (A fd') ⇒ compatible CFS fd fd')
```

3.8.8 Dynamic Method Lookup

If a method is invoked, the constant pool returns the name of the class, where the method has been declared statically (see §4.11). But this is not necessarily the dynamically invoked method. To find that method, we have to search beginning from the current class in all superclasses. The function dyn_class is defined by well-founded recursion and we have proved the following property for it:

```
dyn_class :: 'instr classfiles × method_loc × cname ⇒ cname
WF_classfiles CFS ∧ is_class CFS cn ⟶
dyn_class (CFS, ml, cn) =
      (let ms = get_methods (CFS !! cn)
       in
       if ms ml ≠ None then cn
          else if cn = Object then arbitrary
                 else (let sc = get_superclass (CFS !! cn)
                       in
                       dyn_class (CFS, ml, sc)))
```

4 Java Virtual Machine Instructions

This section describes the instruction set of the JVM we have considered so far, and gives an operational semantics for them.

4.1 Description of JVM instructions

The Java Virtual Machine Specification [LY96] describes the operational semantics for each instruction in the context of a JVM state, where several constraints hold, e.g. there must be an appropriate number of arguments on the operand stack, or the operands must be of a certain type. If the constraints are not satisfied, the behaviour of the JVM is undefined.

One way of defining this partiality is to give a set of conditional execution rules, where the premises only hold for correct states. However, this method does not make clear by construction, whether the definition is complete, i.e. whether execution of a (correct) program will not get stuck before a final state is reached. Besides that, it has to be proved, that the behaviour is deterministic.

In our approach, we formalize the behaviour of a JVM instruction in a functional style, thus guaranteeing by definition the complete execution of a program. If a state is not correct with respect to the current instruction, e.g. the operand stack is empty in case of a *pop* instruction, the result of popping an element from the empty stack will be an arbitrary value. Remember, that this is not a special error value, but means that we do not know any properties of the returned value. The bytecode verifier has to check before execution of the code, that all constraints will be satisfied.

4.2 JVM Execution

We have structured the instructions in several groups of related instructions, describing each by a its own execution function. This makes the operational semantics easier to understand, since every function only takes the parameters that are needed for the corresponding group of instructions. The single instructions are described below. We need an additional datatype to construct the entire instruction set:

$$
\begin{aligned}
\mathit{instr} = \; & \mathsf{LAS}\ \mathit{load\_and\_store} \\
\mid\; & \mathsf{CO}\ \mathit{create\_object} \\
\mid\; & \mathsf{MO}\ \mathit{manipulate\_object} \\
\mid\; & \mathsf{MA}\ \mathit{manipulate\_array} \\
\mid\; & \mathsf{CH}\ \mathit{check\_object} \\
\mid\; & \mathsf{MI}\ \mathit{meth\_inv} \\
\mid\; & \mathsf{MR}\ \mathit{meth\_ret} \\
\mid\; & \mathsf{OS}\ \mathit{op\_stack} \\
\mid\; & \mathsf{CB}\ \mathit{cond\_branch} \\
\mid\; & \mathsf{UB}\ \mathit{uncond\_branch}
\end{aligned}
$$

Within the context of a JVM program, (i.e. set of class files), JVM execution consists in iterated transformation of the machine state according to the current instruction. If the frame stack is empty or an exception is raised, execution terminates. If the machine has not yet reached a final state, the function exec performs a single execution step: it calls an appropriate execution function and incorporates the result in the new machine state. If execution has reached a final state, exec does not return a new state. This is modeled by embedding the result state in an *option* type:

```
exec :: instr classfiles × jvm_state ⇒ jvm_state option
exec (CFS, None, hp, []) = None
exec (CFS, None, hp, (stk,loc,cls,ml,pc)#frs) = Some
 (case ((get_code (CFS !! cls) ml)) ! pc of
   LAS ins ⇒ (let (stk',loc',pc') = exec_las ins stk loc pc
              in
              (None,hp,(stk',loc',cls,ml,pc')#frs))
 | CO  ins ⇒ (let (xp',hp',stk',pc') = exec_co ins CFS cls hp stk pc
              in
              (xp',hp',(stk',loc,cls,ml,pc')#frs))
 | MO  ins ⇒ (let (xp',hp',stk',pc') = exec_mo ins CFS cls hp stk pc
              in
              (xp',hp',(stk',loc,cls,ml,pc')#frs))
 | MA  ins ⇒ (let (xp',hp',stk',pc') = exec_ma ins CFS cls hp stk pc
              in
              (xp',hp',(stk',loc,cls,ml,pc')#frs))
 | CH  ins ⇒ (let (xp',pc') = exec_ch ins CFS cls hp stk pc
              in
              (xp',hp,(stk,loc,cls,ml,pc')#frs))
 | MI  ins ⇒ (let (xp',frs') = exec_mi ins CFS cls hp (stk,loc,cls,ml,pc)
              in
              (xp',hp,frs'@frs))
 | MR  ins ⇒ (let frs' = exec_mr ins stk frs
              in
              (None,hp,frs'))
 | OS  ins ⇒ (let (stk',pc') = exec_os ins stk pc
              in
              (None,hp,(stk',loc,cls,ml,pc')#frs))
 | CB  ins ⇒ (let (stk',pc') = exec_cb ins stk pc
              in
              (None,hp,(stk',loc,cls,ml,pc')#frs))
 | UB  ins ⇒ (let pc' = exec_ub ins pc
              in
              (None,hp,(stk,loc,cls,ml,pc')#frs)))
exec (CFS, Some xp, hp, frs) = None
```

Execution of an entire JVM program consists in repeated application of exec, as long as the result is not None. The relation $CFS \vdash \sigma \longrightarrow^* \sigma'$ maps a given set of class files CFS and a JVM state σ to a new state σ', where the pair (σ,σ') is in the reflexive transitive closure of successful execution steps:

$$\_ \vdash \_ \longrightarrow^* \_ :: \mathit{instr\ classfiles} \Rightarrow \mathit{jvm\_state} \Rightarrow \mathit{jvm\_state} \Rightarrow \mathit{bool}$$
$$CFS \vdash \sigma \longrightarrow^* \sigma' \stackrel{\mathsf{def}}{=} (\sigma,\sigma') \in \{(s,t).\ \mathsf{exec}\ (CFS,s) = \mathsf{Some}\ t\}^*$$

4.3 Instruction Types

Most of the JVM instructions carry type information about their operators. For example, there are different instructions to load an integer value (*iload*) or a reference value (*aload*) from the local variables. Often, the operational behaviour of these instructions is identical or just differs in certain details. To avoid redundant definitions (and thus redundant proofs), we compactify the representation of those instructions in the following way: instead of defining two different instructions, we represent them by one instruction that has an additional argument carrying the type of the instruction. Thus, the operational semantics for both cases can be expressed all in one. This description style not only reveals to be advantageous for further proof tasks; putting similar things together also improves readability and eases understanding.

4.4 Load and Store

Load and store instructions transfer a value between a local variable and the operand stack. We define a new datatype *ins_type* indicating the expected type of the transferred value:

$$\mathit{ins\_type} = \mathsf{I\_} \mid \mathsf{A\_}$$

We consider then the following set of load and store instructions:

$$\begin{aligned}\mathit{load\_and\_store} = {} & \mathsf{LAload}\ \mathit{ins\_type}\ \mathit{nat} \\ & \mid \mathsf{LAstore}\ \mathit{ins\_type}\ \mathit{nat} \\ & \mid \mathsf{Bipush}\ \mathit{int} \\ & \mid \mathsf{Aconst\_null}\end{aligned}$$

The operational semantics of these instructions is given by the function

$$\begin{aligned}\mathsf{exec\_las} :: {} & [\mathit{load\_and\_store}, \mathit{opstack}, \mathit{locvars}, \mathit{p\_count}] \\ & \Rightarrow (\mathit{opstack} \times \mathit{locvars} \times \mathit{p\_count})\end{aligned}$$

$\mathsf{LAload}\ \mathsf{I\_}$ loads an integer, $\mathsf{LAload}\ \mathsf{A\_}$ loads a reference value from a local variable onto the operand stack:

$$\mathsf{exec\_las}\ (\mathsf{LAload}\ X\ \mathit{idx})\ \mathit{stk}\ \mathit{vars}\ \mathit{pc} = ((\mathit{vars}\ !\ \mathit{idx})\ \#\ \mathit{stk}, \mathit{vars}, \mathit{pc}{+}1))$$

$\mathsf{LAstore}\ X$ stores an integer or reference value into a local variable:

$$\mathsf{exec\_las}\ (\mathsf{LAstore}\ X\ \mathit{idx})\ \mathit{stk}\ \mathit{vars}\ \mathit{pc} = (\mathsf{tl}\ \mathit{stk}\ ,\ \mathit{vars}[\mathit{idx}{:=}\mathsf{hd}\ \mathit{stk}]\ ,\ \mathit{pc}{+}1)$$

You will note that in these two cases the operation succeeds, even if the type of the local variable does not correspond to the type of the instruction. This does not cause any problems, since bytecode verification assures, that there will be a value of legitimate type.

Bipush loads an integer value onto the operand stack:

exec_las (Bipush *ival*) *stk vars pc* = (Intg *ival* # *stk,vars,pc*+1)

Aconst_null loads the null reference onto the operand stack:

exec_las Aconst_null *stk vars pc* = (Null # *stk,vars,pc*+1)

4.5 Object and Array Creation

The Java Virtual Machine Specification [LY96] defines distinct instructions for the creation of a new array of primitive or of reference type. Both work nearly identical, they just differ in the way the type of the array components is determined: for primitive types, *newarray* has as argument a special type indicator *atype*. For arrays of reference type, *anewarray* carries an index *cl_idx* into the constant pool, pointing to the type information. We subsume the representation of these two instructions in one single case. Therefore, we define the following datatypes, where we actually consider only integers as primitive array type:

atype = T_INT
arr_type = PA *atype* | AA *cl_idx*

The following functions convert these values to type descriptors:

type_of_atype :: *atype* ⇒ *field_desc*
type_of_atype *at* $\stackrel{\text{def}}{=}$ I

type_of_arr_type :: [*'instr classfiles,cpool,arr_type*] ⇒ *field_desc*
type_of_arr_type *CFS cpool aty* $\stackrel{\text{def}}{=}$
 case *aty of*
  (PA *at*) ⇒ type_of_atype *at*
 | (AA *idx*) ⇒ type_of_str (extract_Class *cpool idx*)

Now, we define the following set of instructions for object creation:

create_object =
  New *cl_idx* (** Create new object **)
 | IAnewarray *arr_type* (** Create new array of prim/ref type **)

The operational semantics of these instructions is given by the function

exec_co :: [*create_object,'instr classfiles,cname,heap,opstack*]
  ⇒ (*xcpt option* × *heap* × *opstack* × *p_count*)

New creates a new class instance[3], whose type cn is extracted from the constant pool. The value a is a new reference to the heap, where the instance variables of the new object are initialized to their default values. If cn is an interface or if there is no more memory available on the heap, an exception is thrown. The new reference is stored on the operand stack:

```
exec_co (New idx) CFS cls hp stk pc =
 (let cpool = get_cpool (CFS !! cls);
      cn    = get_Id (extract_Class cpool idx);
      a     = newref hp;
      hp'   = hp[a ↦ Obj cn (init_obj CFS cn)];
      xp'   = if is_interface (CFS !! cn) then Some InstantiationError
                else if ∀x. hp x ≠ None then Some OutOfMemory
                     else None
  in (xp',hp',(Addr a)#stk,pc+1))
```

You will note that components of the state change, even if an exception has been thrown. This does not matter, since in our formalization execution stops if an exception is thrown. When extending it to exception handling, we have to return the unchanged state in case of an exception. In fact, the Java Virtual Machine Specification [LY96] does not make clear that point.

IAnewarray creates a new array, whose components are of primitive or reference type, depending on the value of X. It works similarly to the creation of a new class instance. On top of the operand stack there must be an integer value *count*, indicating the length of the array. If that value is negative or there is no more memory available on the heap, an exception is thrown. The components are initialized to the default value corresponding to the component type *fd*:

```
exec_co (IAnewarray X) CFS cls hp stk pc =
 (let cpool = get_cpool (CFS !! cls);
      fd    = type_of_arr_type CFS cpool X;
      count = get_Intg (hd stk);
      xp'   = if count < $#0 then Some NegArrSize
                else if ∀x. hp x ≠ None then Some OutOfMemory
                     else None;
      aref  = newref hp;
      hp'   = hp[aref ↦ Arr fd (init_arr (default_val fd) count)]
  in
  (xp',hp',(Addr aref)#(tl stk),pc+1))
```

[3]In the real JVM, the New instruction does not completely create a new instance, a special instance initialization method has to be invoked. We do not yet consider these initialization methods.

4.6 Object Manipulation

Object data is accessed via the following instructions:

$$\begin{aligned} \textit{manipulate\_object} &= \mathsf{Getfield}\ \textit{fr\_idx} \\ &\mid \mathsf{Putfield}\ \textit{fr\_idx} \end{aligned}$$

The operational semantics of these instructions is given by the function

$$\begin{aligned} \mathsf{exec\_mo} :: [&\textit{manipulate\_object}, \textit{'instr classfiles}, \textit{cname}, \textit{heap}, \textit{opstack}] \\ &\Rightarrow (\textit{xcpt option} \times \textit{heap} \times \textit{opstack} \times \textit{p\_count}) \end{aligned}$$

Getfield fetches a field from an object. The object is determined by the reference a on top of the operand stack, the field descriptor is extracted from the constant pool, containing defining class *fc* and field name *fn*. These components form an index into the field table. If the reference is Null, an exception is thrown:

```
exec_mo (Getfield idx) CFS cls hp stk pc =
 (let a          = hd stk;
      (cl,fs)    = get_Obj (hp !! (get_Addr a));
      cpool      = get_cpool (CFS !! cls);
      (fc,fn,fd) = extract_Fieldref cpool idx;
      xp'        = if a=Null then Some NullPointer else None
  in
  (xp',hp,(fs !! (fc,fn))#(tl stk),pc+1))
```

Putfield stores the top operand stack element v in the field of an object. It works analogous to Getfield:

```
exec_mo (Putfield idx) CFS cls hp stk pc =
 (let (v,a)      = (hd stk,hd (tl stk));
      (cl,fs)    = get_Obj (hp !! (get_Addr a));
      cpool      = get_cpool (CFS !! cls);
      (fc,fn,fd) = extract_Fieldref cpool idx;
      hp'        = hp[get_Addr a ↦ Obj cl (fs[(fc,fn) ↦ v])];
      xp'        = if a=Null then Some NullPointer else None
  in
  (xp',hp',tl (tl stk),pc+1))
```

4.7 Array Manipulation

Instructions for array manipulation load an array component onto the operand stack, or store a value from the operand stack as an array component. We use again an additional argument of type *inst_type* to give a compact representation of the instructions:

$$
\begin{aligned}
\mathit{manipulate\_array} = \;& \mathsf{IAaload}\ \ \mathit{ins\_type}\\
\mid\;& \mathsf{IAastore}\ \mathit{ins\_type}
\end{aligned}
$$

The operational semantics of these instructions is given by the function

$$
\begin{aligned}
\mathsf{exec\_ma} :: \;& \mathit{manipulate\_array} \Rightarrow \mathit{'instr\ classfiles} \Rightarrow \mathit{cname} \Rightarrow\\
& \mathit{heap} \Rightarrow \mathit{opstack} \Rightarrow (\mathit{xcpt\ option} \times \mathit{heap} \times \mathit{opstack} \times \mathit{p\_count})
\end{aligned}
$$

IAaload loads an integer resp. reference from an array, where the array reference a and the index i into the array are popped from the operand stack. If the reference is Null or the index is not within the bounds of the referenced array, an exception is thrown:

$$
\begin{array}{l}
\mathsf{exec\_ma}\ (\mathsf{IAaload}\ X)\ \mathit{CFS\ cls\ hp\ stk\ pc} =\\
\quad (\mathsf{let}\ (i,a) \quad = (\mathsf{hd}\ \mathit{stk}, \mathsf{hd}\ (\mathsf{tl}\ \mathit{stk}));\\
\qquad\quad (\mathit{fd},\mathit{ad}) = \mathsf{get\_Arr}\ (\mathit{hp}\ !!\ (\mathsf{get\_Addr}\ a));\\
\qquad\quad \mathit{xp} \qquad = \mathsf{if}\ a{=}\mathsf{Null}\ \mathsf{then\ Some\ NullPointer}\\
\qquad\qquad\qquad\quad \mathsf{else\ if\ length}\ \mathit{ad} \le \mathsf{get\_Nat}\ i\ \mathsf{then\ Some\ IndOutBound}\\
\qquad\qquad\qquad\qquad \mathsf{else\ None}\\
\quad \mathsf{in}\\
\quad (\mathit{xp},\mathit{hp},(\mathit{ad}\ !\ (\mathsf{get\_Nat}\ i)\ \#(\mathsf{tl}\ (\mathsf{tl}\ \mathit{stk})),\mathit{pc}{+}1))
\end{array}
$$

If a value of reference type is stored in an array, runtime type checking always has to be performed. Therefore, we define a predicate fd_compatible that checks whether the type of a value *val* is assignment compatible with a given type *fd*. According to our representation of instruction types, an additional argument X indicates the expected type of *val*. For integer types (I_) no runtime type checking is necessary, therefore the result is trivially True. A Null value may be assigned to any reference type[4], for reference values, the type *fd'* of the referenced object must be compatible with *fd'*:

$$
\begin{array}{l}
\mathsf{fd\_compatible} :: [\mathit{'instr\ classfiles},\mathit{heap},\mathit{ins\_type},\mathit{val},\mathit{field\_desc}] \Rightarrow \mathit{bool}\\
\mathsf{fd\_compatible}\ \mathit{CFS\ hp\ X\ val\ fd} \stackrel{\mathsf{def}}{=}\\
\quad \mathsf{case}\ X\ \mathit{of}\ \ \mathsf{I\_} \Rightarrow \mathsf{True}\\
\qquad\qquad \mid \mathsf{A\_} \Rightarrow \mathsf{if}\ \mathit{val}{=}\mathsf{Null}\ \mathsf{then}\ \mathit{fd}{\neq}\mathsf{I}\\
\qquad\qquad\qquad\quad \mathsf{else\ let}\ \mathit{fd'} = \mathsf{get\_obj\_type}\ (\mathit{hp}\ !!\ (\mathsf{get\_Addr}\ \mathit{val}))\\
\qquad\qquad\qquad\qquad \mathsf{in}\\
\qquad\qquad\qquad\qquad \mathsf{compatible}\ \mathit{CFS\ fd'\ fd}
\end{array}
$$

IAastore stores an integer resp. reference value v into an array. It works analogous to IAaload. Additionally an exception is thrown, if the dynamic type of the value is not compatible with the component type *fd* of the referenced array:

[4] In the Java Virtual Machine Specification [LY96], the description of the *aastore* instruction does not mention this case, whereas in §2.6.6 the definition is complete

```
exec_ma (IAastore X) CFS cls hp stk pc =
 (let (v,i,a)   = (hd stk,hd (tl stk),hd (tl (tl stk)));
      (fd,ad)   = get_Arr (hp !! (get_Addr a));
      hp'       = hp[get_Addr a ↦ Arr fd (ad[get_Nat i := v])];
      xp        = if a=Null then Some NullPointer
                     else if length ad ≤ get_Nat i then Some IndOutBound
                          else if ¬(fd_compatible CFS hp X v fd)
                               then Some ArrStore
                               else None
 in
 (xp,hp',tl (tl (tl stk)),pc+1))
```

4.8 Check Object

We consider one instruction to check object properties:

check_object = Checkcast *cl_idx*

The operational semantics of this instruction is given by the function

exec_ch :: [*check_object*, *'instr classfiles*, *cname*, *heap*] ⇒ *xcpt option*

Checkcast checks, if an object is of a given type. If the object reference a is not Null and the object type ot is not compatible with the type ct that is extracted from the constant pool, an exception is thrown. The operand stack remains unchanged:

```
exec_ch (Checkcast idx) CFS cls hp stk pc =
 (let a      = hd stk;
      cpool  = get_cpool (CFS !! cls);
      ct     = type_of_str (extract_Class cpool idx);
      ot     = get_obj_type (hp !! (get_Addr a));
      xp     = if a=Null then None
                  else if ¬(compatible CFS ot ct) then Some ClassCast
                       else None
  in
  (xp,pc+1))
```

4.9 Control Transfer

Conditional control transfer causes the JVM to proceed execution at a given offset from the current program counter. There are several integer comparisons of the top stack element against the integer value 0. We subsume them by one instruction If_, whose additional argument of type *comp0* specifies the compare operation. The set of instructions is then formalized as follows:

$$
\begin{aligned}
\mathit{comp0} &= \mathit{int} \Rightarrow \mathit{bool}\\
\mathit{cond\_branch} &= \mathsf{If\_}\ \mathit{comp0}\ \mathit{int}\\
&\mid \mathsf{Ifnull}\ \mathit{int}\\
&\mid \mathsf{Ifiacmpeq}\ \mathit{ins\_type}\ \mathit{int}
\end{aligned}
$$

The execution of these instructions is described as follows:

$\mathsf{exec\_cb} :: [\mathit{cond\_branch}, \mathit{opstack}, \mathit{nat}] \Rightarrow \mathit{opstack} \times \mathit{p\_count}$

$\mathsf{If\_}$ causes a jump, if *cmp*-comparison of the top element of the operand stack succeeds:

```
exec_cb (If_ cmp i) stk pc =
 (let val  = hd stk;
      pc'  = if cmp (get_Intg val) then int2nat($#pc+i) else pc+1
  in
  (tl stk,pc'))
```

Ifnull branches, if the top stack element equals Null:

```
exec_cb (Ifnull i) stk pc =
 (let val  = hd stk;
      pc'  = if val=Null then int2nat($#pc+i) else pc+1
  in
  (tl stk,pc'))
```

$\mathsf{Ifiacmpeq}$ causes a jump, if the two top operand stack elements (that are both integer resp. reference values, depending on X) are equal.

```
exec_cb (Ifiacmpeq X i) stk pc =
  (let (val1,val2) = (hd stk, hd (tl stk));
       pc'         = if val1 = val2 then int2nat($#pc+i) else pc+1
   in
   (tl (tl stk),pc'))
```

In this case again, bytecode verification will assure that there will be values of legitimate type on the operand stack.

There is one unconditional branch instruction in our formalization:

$\mathit{uncond\_branch} = \mathsf{Goto}\ \mathit{int}$

Its operational semantics is defined as follows:

```
exec_ub :: [uncond_branch,nat] => p_count
exec_ub (Goto i) pc = int2nat($#pc+i)
```

4.10 Operand Stack

The JVM has several instructions for the direct manipulation of the operand stack. We have modeled the following subset:

$$
\begin{aligned}
\mathit{op\_stack} = \ & \mathsf{Pop} \\
\mid \ & \mathsf{Dup} \\
\mid \ & \mathsf{Swap}
\end{aligned}
$$

The definition of the execution function is straightforward:

$$\mathsf{exec\_os} :: [\mathit{op\_stack}, \mathit{opstack}, \mathit{p\_count}] \Rightarrow (\mathit{opstack} \times \mathit{p\_count})$$

Pop removes the top element from the operand stack.

$$\mathsf{exec\_os}\ \mathsf{Pop}\ \mathit{stk}\ \mathit{pc} = (\mathsf{tl}\ \mathit{stk}, \mathit{pc}{+}1)$$

Dup duplicates the top element of the operand stack.

$$\mathsf{exec\_os}\ \mathsf{Dup}\ \mathit{stk}\ \mathit{pc} = ((\mathsf{hd}\ \mathit{stk})\#\mathit{stk}, \mathit{pc}{+}1))$$

Swap interchanges the two top words of the operand stack.

$$
\begin{aligned}
& \mathsf{exec\_os}\ \mathsf{Swap}\ \mathit{stk}\ \mathit{pc} = \\
& \ (\mathsf{let}\ (\mathit{val1}, \mathit{val2}) = (\mathsf{hd}\ \mathit{stk}, \mathsf{hd}\ (\mathsf{tl}\ \mathit{stk})) \\
& \quad \mathsf{in} \\
& \quad (\mathit{val2}\#\mathit{val1}\#(\mathsf{tl}\ (\mathsf{tl}\ \mathit{stk})), \mathit{pc}{+}1))
\end{aligned}
$$

4.11 Method Invocation

The invocation of instance or interface methods works nearly identical. The only difference is an additional argument to the *invokeinterface* instruction, indicating the number of arguments. This argument is rather redundant, because it could be derived from the method descriptor just as for the *invokevirtual* instruction. We have modeled this redundancy according to the Java Virtual Machine Specification [LY96], but have omitted the last operand of *invokeinterface*, because it is unused by the instruction itself and exists only to reserve space for further optimizations in Sun's implementation of the JVM. We do not yet consider special methods like initialization methods or class methods:

$$
\begin{aligned}
\mathit{meth\_inv} = \ & \mathsf{Invokevirtual}\ \mathit{mr\_idx} \\
\mid \ & \mathsf{Invokeinterface}\ \mathit{im\_idx}\ \mathit{nat}
\end{aligned}
$$

To avoid redundant definitions, we define a function extract_inv_methref, that extracts the method information from the constant pool for both cases:

```
extract_inv_methref :: [cpool,inv_type]
                         ⇒ (cname × method_loc × return_desc × nat)
extract_inv_methref cp (Invokevirtual idx) =
 (let (cn,mn,(pd,rd)) = extract_Methodref cp idx
  in
  (cn,(mn,pd),rd,length pd+1))

extract_inv_methref cp (Invokeinterface idx n) =
 (let (cn,mn,(pd,rd)) = extract_InterMethref cp idx
  in
  (cn,(mn,pd),rd,n))
```

The operational semantics is then given by the function

```
exec_mi :: [meth_inv,'instr classfiles,cname,heap,frame]
           ⇒ (xcpt option × frame list)
```

Defining class *cn*, signature *ml*, return descriptor *rd* and number of arguments *n* of the invoked method are extracted from the constant pool. Then, dynamic method lookup is performed: starting from the object class of the object referenced by *a*, dyn_class searches in the class and its superclasses, until a method matching the signature is found. Object reference and arguments are popped from the operand stack and a new stack frame is created, where these values are incorporated as local variables.

```
exec_mi inv_com CFS cls hp (stk,loc,cls,met,pc) ≝
 (let cp          = get_cpool (CFS !! cls);
      (cn,ml,rd,n) = extract_inv_methref cp (get_Invoke inv_com);
      xs          = take n stk;
      a           = last xs;
      dyn_cn      = dyn_class (CFS, ml, get_obj_class (hp !! get_Addr a));
      frs'        = [([],rev xs,dyn_cn,ml,0),(drop n stk,loc,cls,met,pc+1)];
      xp'         = if a=Null then Some NullPointer else None
  in
  (xp',frs'))
```

The Java Virtual Machine Specification [LY96] emphasizes the fact that there exist no requirements on the representation of objects. However, the description for method invocation instructions *invokevirutal* and *invokeinterface* refer to method-tables as a part of an object. This is indeed an implementation-dependent optimization that is not part of the abstract specification. In our formalization we do not use method tables.

4.12 Method Return

The JVM has different return instructions, depending on the type of the returned value (if any). We chose again a compressed representation with an additional *ins_type* argument:

$$\begin{aligned} \textit{meth\_ret} = {} & \mathsf{IAreturn}\ \textit{ins\_type} \\ & \mid \mathsf{Return} \end{aligned}$$

The operational semantic of these instructions is defined by the function

$\mathsf{exec\_mr} :: [\textit{meth\_ret},\textit{opstack},\textit{frame list}] \Rightarrow \textit{frame list}$

IAreturn returns an integer or reference, depending on the value of X. The value *val* on top of the current operand stack *stk* is pushed onto the operand stack *stk'* of the frame of the invoker, and the current frame is deleted.

$$\begin{array}{l} \mathsf{exec\_mr}\ (\mathsf{IAreturn}\ X)\ \textit{stk}\ \textit{frs} = \\ \quad (\mathsf{let}\ \textit{val} \qquad\qquad\qquad\qquad = \mathsf{hd}\ \textit{stk} \\ \qquad\quad (\textit{stk'},\textit{loc'},\textit{cls'},\textit{met'},\textit{pc'}) = \mathsf{hd}\ \textit{frs} \\ \quad\ \ \mathsf{in} \\ \quad\ \ (\textit{val}\#\textit{stk'},\textit{loc'},\textit{cls'},\textit{met'},\textit{pc'})\#\mathsf{tl}\ \textit{frs}) \end{array}$$

Return returns from a method that has return type void. In this case, there is no value to be returned; the current frame *fr* is deleted and the remaining frames *frs* are unchanged:

$\mathsf{exec\_mr}\ \mathsf{Return}\ \textit{stk}\ \textit{frs} = \textit{frs}$

5 Results and Further Work

We have given a formalization of the central parts of the JVM in Isabelle/HOL. The theory files comprise nearly 1100 lines of code.

Isabelle/HOL turned out to be an adequate instrument to model real life programming languages such as Java (see also [NO98, ON98]). It is obvious that we had to make certain restrictions in this first approach to formalize the JVM. For example we do not consider the size of instructions and its operands and use instead abstract datatypes. These abstractions can be refined in further development steps of our formalization. Besides those "low level" points, our formalization of the operational semantics corresponds closely to the informal description given in the Java Virtual Machine Specification [LY96].

We have succeeded in finding a description style, that is suitable for theoretical investigations on the one hand and practical implementation guide on the other hand. In contrast to an informal description, we can trust by construction in the soundness of our formalization and profit from precise definitions. The result can be used for machine-checked verification tasks.

Our work has revealed several lacks and inconsistencies of the official JVM description: the latter claims to give an abstract specification that must be followed by any concrete implementation. However, in many cases it confounds the reader by referring to implementation details: for example it uses the notion of method tables for method invocation. In this context, it is curious that the design of the *invokeinterface* instruction depends on an optimization in Sun's implementation of the JVM.

Another point of criticism concerns the relation between Java source level and JVM: the Java Virtual Machine Specification [LY96] emphasizes the fact, that the JVM knows nothing about the Java programming language and can be used as platform for any other programming language whose functionality can be translated to JVM class files. It is therefore annoying that throughout the whole book, the descriptions refer to the Java source language instead of giving independent definitions. Even worse, Java Language Specification [GJS96] and Java Virtual Machine Specification [LY96] differ in the description of some basic Java concepts (e.g. interface method overriding and interface implementation, see §3.7.5). This leads to confusion, since it is not immediately clear whether these differences are intentional or just are resulting from inexact reproduction. Another example is the duplicate definition of of *assignment compatibility* (see §4.7), where one version is again incomplete.

Apart from these results, this work serves as basis for further formal treatment of Java and the JVM:

Bytecode Verifier: A correct Java compiler generates correct code that can be executed safely on the JVM. However, the VM cannot know, how the code has been generated and if it has been generated properly, because the platform-independent design of Java makes it possible to download arbitrary (ill-formed) code from the World Wide Web.
Consequently the JVM needs to check any untrusted code before executing it. This process is known as bytecode verification. Relying on the work of Qian [Qia98], we have formalized the bytecode verifier in Isabelle/HOL. We then have proved that if a program has been checked by the bytecode verifier, then the runtime data of the program will be type-correct. The results of this work are described in [Pus98].

Compiler correctness: The JVM is often considered as an operational semantics for Java. But it is not as easy as that, since Java programs have to be compiled into JVM code. Our main goal is the formal verification of compiler correctness for Java. We will formalize a Java compiler translating Java source programs in JVM class files. Then we will prove the equivalence between source program and compiled code.

Acknowledgments. I would like to thank Tobias Nipkow, David von Oheimb, and Zhenyu Qian for helpful discussions about this topic. Thanks are also owed to Franz Regensburger who read a draft version of this paper.

References

[Coh97] Richard M. Cohen. The defensive Java Virtual Machine specification. Technical report, Computational Logic Inc., 1997. Draft version.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lect. Notes in Comp. Sci.*, pages 389–418. Springer-Verlag, 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[HBL98] Pieter Hartel, Michael Butler, and Moshe Levy. The Operational Semantics of a Java Secure Processor. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1??? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998. To appear.

[Isa] The Isabelle library. Available on the World Wide Web at URL `http://www.in.tum.de/~isabelle/library/`.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[NO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[ON98] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1??? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998. To appear.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[Pus98] Cornelia Pusch. Proving the Soundness of the Java Bytecode Verifier. Technical Report TUM-I98??, Institut für Informatik, Technische Universiät München, 1998. To appear.

[Qia98] Zhenyu Qian. A Formal Specification of Java Virtual Machine instructions for Objects, Methods and Subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1??? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998. To appear.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages.* ACM Press, 1998. To appear.

[Sar97] Vijay Saraswat. Java is not type-safe. . `http://www.research.att.com/~vj/main.html`, August 1997.

[Sli96] Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 381–397. Springer-Verlag, 1996.

[Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 275–290. Springer-Verlag, 1997.

[Sym97] Donald Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997.

Index