Verified Bytecode Verifiers

Gerwin Klein, Tobias Nipkow

Technische Universität München,
Institut für Informatik, 80290 München, Germany

Abstract

Using the theorem prover Isabelle/HOL we have formalized and proved correct an executable bytecode verifier in the style of Kildall's algorithm for a significant subset of the Java Virtual Machine. First an abstract framework for proving correctness of data flow based type inference algorithms for assembly languages is formalized. It is shown that under certain conditions Kildall's algorithm yields a correct bytecode verifier. Then the framework is instantiated with our previous work about the JVM. Finally we demonstrate the flexibility of the framework by extending our previous JVM model and the executable bytecode verifier with object initialization.

Key words: Java, Bytecode Verification, Theorem Proving, Data Flow Analysis, Object Initialization

1 Introduction

Over the past few years there has been considerable interest in formal models for Java and the JVM, and in particular its bytecode verifier (*BCV*). So far most of the work has concentrated on abstract models of particularly tricky aspects of the JVM, specifically object initialization and the idiosyncratic notion of "subroutines". This paper complements those studies by focussing on machine-checked proofs of executable models. Its distinctive features are:

- The first machine-checked proof of correctness of a BCV implementation for a nontrivial subset of the JVM (including some of the tricky aspects mentioned above).

URLs: `http://www.in.tum.de/~kleing` (Gerwin Klein),
`http://www.in.tum.de/~nipkow` (Tobias Nipkow).

- The BCV is an almost directly executable functional program (which can be generated automatically from its Isabelle/HOL formalization). The few non-executable constructs (some choice functions) are easily implemented.
- The work is modular: the BCV (and its correctness proof!) becomes a simple instance of a general framework for data flow analysis.
- Almost all details of the model are presented. The complete formalization, including proofs, is available via the authors' home page.

Thus the novelty is not in the actual mathematics but in setting up a framework that matches the needs of the JVM, instantiating it with a description of the JVM, and doing this in complete detail, down to an executable program, and including all the proofs.

Moreover this is not an isolated development but it is integrated directly with the existing formalizations of Java and the JVM in Isabelle/HOL [1–3]. From them it also inherits the absence of subroutines and exception handling. Object initialization was not treated in [1–3] either; we will introduce this new feature here, and show how the framework deals with extensions.

Although our only application is the JVM, our modular framework is in principle applicable to other "typed assembly languages" such as MSIL [4] as well; hence the plural in the title.

What does the BCV do and what does it guarantee? The literature already contains a number of (partial) answers to this question. We follow the type system approach of Stata, Abadi and others [5–10]. The type systems check a program w.r.t. additional type annotations that provide the missing typing of storage locations for each instruction. These type systems are then shown to guarantee type soundness, i.e. absence of type errors during execution (which was verified formally by Pusch [3] for a subset of the system by Qian [8]). Only Qian [11] proves the correctness of an algorithm for turning his type checking rules into a data flow analyzer. However, his algorithm is still quite abstract.

Closely related is the work by Goldberg [12] who rephrases and generalizes the overly concrete description of the BCV given in [13] as an instance of a generic data flow framework. Work towards a verified implementation in the SPECWARE system is sketched by Coglio *et al.* [14]. Although we share the lattice-theoretic foundations with this work and it appears to consider roughly the same instruction set, it is otherwise quite different: whereas we solve the data flow problem directly, they generate constraints to be solved separately, which is not described. Furthermore, they state desired properties axiomatically but do not prove them. Casset and Lanet [15] also sketch work using the B method to model the JVM. However, their subset of the JVM is extremely simple (it does not even include classes), and it remains unclear what exactly they have proved. Based on the work of Freund and Mitchell [6] Bertot [16] has recently used the Coq system to prove the correctness of a bytecode verifier that handles object initialization, but again without classes. Barthe *et al.* [17,18] also use the Coq system; they cover the complete instruction set of the JavaCard platform, but do not handle object initialization.

The rest of the paper is structured as follows. The basic datatypes and semilattices required for data flow analysis are introduced in §2. Our abstract framework relating type systems, data flow analysis, and bytecode verification is set up in §3. It is proved that under certain conditions bytecode verification determines welltypedness. We also show that Kildall's algorithm, an iterative data flow analysis, is a bytecode verifier. In §4, the results of the previous sections are used to derive a bytecode verifier for the existing μJava formalization. Finally, in §5 we add the object initialization feature to the μJava bytecode verifier, show how to integrate it with Kildall's algorithm, and discuss its soundness.

2 Types, orders, and semilattices

This section introduces the basic mathematical concepts and their formalization in Isabelle/HOL. Note that HOL distinguishes types and sets: types are part of the meta-language and of limited expressiveness, whereas sets are part of the object language and very expressive.

2.1 Basic types

Isabelle's type system is similar to ML's. There are the basic types *bool*, *nat*, and *int*, and the polymorphic types α *set* and α *list* and a conversion function *set* from lists to sets. The "cons" operator on lists is the infix #, concatenation the infix @. The length of a list is denoted by *size*. The i-th element (starting with 0) of list xs is denoted by $xs \; ! \; i$. Overwriting the i-th element of a list xs with a new value x is written $xs[i := x]$. Recursive datatypes are introduced with the *datatype* keyword. The remainder of this section introduces the HOL-formalization of the basic lattice-theoretic concepts required for data flow analysis and its application to the JVM.

2.2 Partial orders

Partial orders are formalized as binary predicates. Based on the type synonym $\alpha \; ord = \alpha \Rightarrow \alpha \Rightarrow bool$ and the notations $x \leq_r y = r \; x \; y$ and

$x <_r y = (x \le_r y \wedge x \neq y)$ we say that r is a **partial order** iff the predicate *order* :: α *ord* $\Rightarrow$ *bool* holds for r:

$$\begin{aligned} \textit{order}\ r = {} & (\forall x.\ x \le_r x) \wedge (\forall x\ y.\ x \le_r y \wedge y \le_r x \longrightarrow x{=}y) \wedge \\ & (\forall x\ y\ z.\ x \le_r y \wedge y \le_r z \longrightarrow x \le_r z) \end{aligned}$$

We say that r satisfies the **ascending chain condition** if there is no infinite ascending chain $x_0 <_r x_1 <_r \ldots$ and call $\top$ a **top element** if $x \le_r \top$ for all x.

2.3 Semilattices

Based on the type synonyms α *binop* $= \alpha \Rightarrow \alpha \Rightarrow \alpha$ and α *sl* $= \alpha$ *set* $\times$ α *ord* $\times$ α *binop* and the supremum notation $x +_f y = f\ x\ y$ we say that (A,r,f) :: α *sl* is a **semilattice** iff the predicate *semilat* :: α *sl* $\Rightarrow$ *bool* holds:

$$\begin{aligned} \textit{semilat}\ (A,r,f) = {} & \textit{order}\ r \wedge \textit{closed}\ A\ f \wedge \\ & (\forall x\ y \in A.\ x \le_r x +_f y) \wedge (\forall x\ y \in A.\ y \le_r x +_f y) \wedge \\ & (\forall x\ y\ z \in A.\ x \le_r z \wedge y \le_r z \longrightarrow x +_f y \le_r z) \end{aligned}$$

where $\textit{closed}\ A\ f = \forall x\ y \in A.\ x +_f y \in A$

Data flow analysis is usually phrased in terms of infimum semilattices. We have chosen a supremum semilattice because it fits better with our intended application, where the ordering is the subtype relation and the join of two types is the least common supertype (if it exists).

We will now look at a few datatypes and the corresponding semilattices which are required for the construction of the JVM bytecode verifier. The definition of those semilattices follows a pattern: we lift an existing semilattice to a new semilattice with more structure. We do this by extending the carrier set, and by giving two functionals *le* and *sup* that lift the ordering and supremum operation to the new semilattice. In order to avoid name clashes, Isabelle provides separate names spaces for each *theory*, where a theory is like a module in a programming language. Qualified names are of the form *Theoryname.localname*, they apply to constant definitions and functions as well as type constructions. So, if we write *Err.sup* later on, we refer to the *sup* functional defined for the error type in §2.4.

2.4 The error type and err-semilattices

Theory *Err* introduces an error element to model the situation where the supremum of two elements does not exist. We introduce both a datatype and an equivalent construction on sets:

$$\textit{datatype}\ \alpha\ \textit{err} = \textit{Err} \mid \textit{OK}\ \alpha \qquad \textit{err}\ A = \{\textit{Err}\} \cup \{\textit{OK}\ a \mid a \in A\}$$

An ordering r on α can be lifted to α *err* by making *Err* the top element:

$$\begin{array}{llll} \textit{le}\ r\ (\textit{OK}\ x) & (\textit{OK}\ y) & = & x \le_r y \\ \textit{le}\ r\ \_ & \textit{Err} & = & \textit{True} \\ \textit{le}\ r\ \textit{Err} & (\textit{OK}\ y) & = & \textit{False} \end{array}$$

We proved that *le* preserves the ascending chain condition.

The following lifting functional is frequently useful:

$$\begin{array}{llll} \multicolumn{4}{l}{\textit{lift2} :: (\alpha \Rightarrow \beta \Rightarrow \gamma\ \textit{err}) \Rightarrow \alpha\ \textit{err} \Rightarrow \beta\ \textit{err} \Rightarrow \gamma\ \textit{err}} \\ \textit{lift2}\ f\ (\textit{OK}\ x) & (\textit{OK}\ y) & = & f\ x\ y \\ \textit{lift2}\ f\ \quad \_ & \_ & = & \textit{Err} \end{array}$$

This brings us to the genuinely new notion of an err-semilattice. It is a variation of a semilattice with top element. Because the behaviour of the ordering and the supremum on the top element are fixed, it suffices to say how they behave on non-top elements. Thus we can represent a semilattice with top element *Err* compactly by a triple of type *esl*:

$$\alpha\ \textit{ebinop} = \alpha \Rightarrow \alpha \Rightarrow \alpha\ \textit{err} \qquad \alpha\ \textit{esl} = \alpha\ \textit{set} \times \alpha\ \textit{ord} \times \alpha\ \textit{ebinop}$$

Conversion between the types *sl* and *esl* is easy:

$$\begin{array}{ll} \textit{esl} :: \alpha\ \textit{sl} \Rightarrow \alpha\ \textit{esl} & \textit{sl} :: \alpha\ \textit{esl} \Rightarrow \alpha\ \textit{err}\ \textit{sl} \\ \textit{esl}(A,r,f) = (A,\ r,\ \lambda x\ y.\ \textit{OK}(f\ x\ y)) & \textit{sl}(A,r,f) = (\textit{err}\ A,\ \textit{le}\ r,\ \textit{lift2}\ f) \end{array}$$

Now we define $L :: \alpha\ \textit{esl}$ to be an **err-semilattice** iff *sl* L is a semilattice. It follows easily that *esl* L is an err-semilattice if L is a semilattice. The supremum operation of *sl*(*esl* L) is useful on its own:

$$\textit{sup}\ f = \textit{lift2}\ (\lambda x\ y.\ \textit{OK}(x +_f y))$$

In a strongly typed environment like HOL we found err-semilattices easier to work with than semilattices with top element.

2.5 The option type

Theory *Opt* introduces the new type *option* and the set *opt* as duals to type *err* and set *err*,

$$\textit{datatype}\ \alpha\ \textit{option} = \textit{None} \mid \textit{Some}\ \alpha \qquad \textit{opt}\ A = \{\textit{None}\} \cup \{\textit{Some}\ a \mid a \in A\}$$

an ordering that makes *None* the bottom element, and a corresponding supremum operation:

$$\begin{array}{llll} \mathit{le}\ r & (\mathit{Some}\ x) & (\mathit{Some}\ y) & = & x \le_r y \\ \mathit{le}\ r & \mathit{None} & \_ & = & \mathit{True} \\ \mathit{le}\ r & (\mathit{Some}\ x) & \mathit{None} & = & \mathit{False} \end{array}$$

$$\begin{array}{llll} \mathit{sup}\ f & (\mathit{Some}\ x) & (\mathit{Some}\ y) & = & \mathit{Some}(f\ x\ y) \\ \mathit{sup}\ f & \mathit{None} & z & = & z \\ \mathit{sup}\ f & z & \mathit{None} & = & z \end{array}$$

We proved that function $\mathit{sl}(A,r,f) = (\mathit{opt}\ A,\ \mathit{le}\ r,\ \mathit{sup}\ f)$ maps semilattices to semilattices and that *le* preserves the ascending chain condition.

The *option* datatype is not only useful in the semilattice context, but also serves as a generic way to extend a given datatype by a special element, e.g. to model maps in HOL. A frequently used function is the destructor *the* that satisfies $\mathit{the}\ (\mathit{Some}\ x) = x$.

2.6 Products

Theory *Product* provides what is known as the *coalesced* product, where the top elements of both components are identified. In terms of err-semilattices, this is

$\mathit{esl} :: \alpha\ \mathit{esl} \Rightarrow \beta\ \mathit{esl} \Rightarrow (\alpha \times \beta)\ \mathit{esl}$
$\mathit{esl}\ (A,r_A,f_A)\ (B,r_B,f_B) = (A \times B,\ \mathit{le}\ r_A\ r_B,\ \mathit{sup}\ f_A\ f_B)$

$\mathit{le} :: \alpha\ \mathit{ord} \Rightarrow \beta\ \mathit{ord} \Rightarrow (\alpha \times \beta)\ \mathit{ord}$
$\mathit{le}\ r_A\ r_B = \lambda(a_1,b_1)(a_2,b_2).\ a_1 \le_{r_A} a_2 \wedge b_1 \le_{r_B} b_2$

$\mathit{sup} :: \alpha\ \mathit{ebinop} \Rightarrow \beta\ \mathit{ebinop} \Rightarrow (\alpha \times \beta)\ \mathit{ebinop}$
$\mathit{sup}\ f\ g = \lambda(a_1,b_1)(a_2,b_2).\ \mathit{Err.sup}\ (\lambda x\ y.(x,y))\ (a_1 +_f a_2)\ (b_1 +_g b_2)$

Note that we use $\times$ both on the type and set level.

We have shown that if both L_1 and L_2 are err-semilattices, so is $\mathit{esl}\ L_1\ L_2$, and that if both r_A and r_B satisfy the ascending chain condition, so does $\mathit{le}\ r_A\ r_B$.

2.7 Lists of fixed length

Theory *Listn* provides the concept of lists of a given length over a given set. In HOL, this is formalized as a set rather than a type:

$$\textit{listn}\ n\ A = \{xs \mid \textit{size}\ xs = n \land \textit{set}\ xs \subseteq A\}$$

This set can be turned into a semilattice in a componentwise manner, essentially viewing it as an n-fold cartesian product:

$$\begin{array}{ll} \textit{sl} :: \textit{nat} \Rightarrow \alpha\ \textit{sl} \Rightarrow \alpha\ \textit{list sl} & \textit{le} :: \alpha\ \textit{ord} \Rightarrow \alpha\ \textit{list ord} \\ \textit{sl}\ n\ (A,r,f) = (\textit{listn}\ n\ A,\ \textit{le}\ r,\ \textit{map2}\ f) & \textit{le}\ r = \textit{list-all2}\ (\lambda x\ y.\ x \le_r y) \end{array}$$

where $\textit{map2} :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha\ \textit{list} \Rightarrow \beta\ \textit{list} \Rightarrow \gamma\ \textit{list}$ and $\textit{list-all2} :: (\alpha \Rightarrow \beta \Rightarrow \textit{bool}) \Rightarrow \alpha\ \textit{list} \Rightarrow \beta\ \textit{list} \Rightarrow \textit{bool}$ are the obvious functions. We introduce the notation $xs \le[r]\ ys\ =\ xs\ \le_{(le\ r)} ys$. We have shown (by induction on n) that if L is a semilattice, so is $\textit{sl}\ n\ L$, and that if r is a partial order and satisfies the ascending chain condition, so does $\textit{le}\ r$.

In case we want to combine lists of different lengths, or if the supremum on the elements of the list may return *Err*, we use the following function:

$$\begin{array}{l} \textit{sup} :: (\alpha \Rightarrow \beta \Rightarrow \gamma\ \textit{err}) \Rightarrow \alpha\ \textit{list} \Rightarrow \beta\ \textit{list} \Rightarrow \gamma\ \textit{list err} \\ \textit{sup}\ f\ xs\ ys = \textit{if size}\ xs = \textit{size}\ ys\ \ \textit{then coalesce}(\textit{map2}\ f\ xs\ ys)\ \textit{else Err} \end{array}$$

$$\begin{array}{l} \textit{coalesce}\ [] = \textit{OK}\ [] \\ \textit{coalesce}\ (e\#es) = \textit{Err.sup}\ (\lambda x\ xs.\ x\#xs)\ e\ (\textit{coalesce}\ es) \end{array}$$

This corresponds to the coalesced product. Below we also need the structure of all lists up to a specific length:

$$\begin{array}{l} \textit{uptoesl} :: \textit{nat} \Rightarrow \alpha\ \textit{esl} \Rightarrow \alpha\ \textit{list esl} \\ \textit{uptoesl}\ n\ (A,r,f) = (\bigcup_{i\ \le\ n} \textit{listn}\ i\ A,\ \textit{le}\ r,\ \textit{sup}\ f) \end{array}$$

We have shown that if L is an err-semilattice, so is $\textit{uptoesl}\ n\ L$.

3 Data flow analysis and type system

The purpose of this section is to set up an abstract framework for type checking and data flow analysis of machine code, i.e. lists of instructions. We assume that instructions may be typed (e.g. distinguishing integer from floating point addition) but storage locations are not necessarily typed and may also change their type during execution. Thus it is necessary to infer the type of each storage location at each instruction to see if the instruction will manipulate values of the required type. To keep things abstract, we do not fix the type system or the machine architecture. We simply assume that our model contains a type of **states** that characterizes the state of the machine. This **state type** is a parameter of our setup and will be represented by the type variable σ. Note that σ is intended not to represent values but their abstraction, types. For example, in a register machine σ would be a list of types, one for each register (roughly speaking). We can now define a **method type**, i.e. the type of a method, simply as a list of state types: each element in the list characterizes the state of the machine before execution of the corresponding instruction. We hasten to add that the name "method" is not significant, we chose it since this is what we will use the framework for later, but the framework itself is not restricted to Java methods. We could also call it program type, or procedure type, or whatever the entities to verify may be. In order to lessen the confusion between types in the programming language under consideration and types in our modeling language, the latter are sometimes referred to as **HOL types**. For example, σ is a HOL type (variable) that represents part of the programming language type system.

Before we continue keeping things as abstract as possible, let us take a look at a concrete example of how things work in the JVM.

The JVM is a stack machine where each method activation has its own expression stack and local variables. The types of operands and results of bytecode instructions are fixed (modulo subtyping), whereas the type of a storage location may differ at different points in the program.

instruction	stack	local variables
Load 0	*Some* ([],	[*Class B*, *Int*])
Store 1	*Some* ([*Class A*],	[*Class B*, *Err*])
Load 0	*Some* ([],	[*Class B*, *Class A*])
Getfield F A	*Some* ([*Class B*],	[*Class B*, *Class A*])
Goto −3	*Some* ([*Class A*],	[*Class B*, *Class A*])

Fig. 1. Example of a welltyping

In figure 1 on the left the instructions are shown and on the right the type of the stack elements and the local variables. The type information attached to an instruction characterizes the state *before* execution of that instruction. We assume that class B is a subclass of A and that A has a field F of type A. The *Some* before each of the type entries means that we were able to predict some type for each of the instructions. If one of the instructions had been unreachable, the type entry would have been *None*.

Execution starts with an empty stack and the two local variables hold a reference to an object of class B and an integer. The first instruction loads local variable 0, a reference to a B object, on the stack. The type information associated with the following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that usage of local variable 1 may produce an error. This means the type information has become less precise but is still correct: a B object is also an A object and an integer is now classified as unusable (*Err*). The reason for these more general types is that the predecessor of the *Store* instruction may have either been *Load 0* or

Goto -3. Since there exist different execution paths to reach *Store*, the type information of the two paths has to be "merged". The type of the second local variable is either *Int* or *Class A*, which are incompatible, i.e. the only common supertype is *Err*.

Bytecode verification is the process of inferring the types on the right from the instruction sequence on the left and some initial condition, and of ensuring that each instruction receives arguments of the correct type. The method type is the right-hand side of the table, a state type is one line of it. Type inference is the computation of a method type from an instruction sequence (usually by iteration), while type checking means checking that a given method type fits an instruction sequence.

With this intuition we turn our attention back to the formalization.

In this abstract setting, we do not yet have to talk about the instruction sequences themselves. They will be hidden inside functions that characterize their behaviour. These functions form the parameters of our model, namely the type system and the data flow analyzer. In the Isabelle formalization, these functions are parameters of everything. In this article, we often make them "implicit parameters", i.e. we pretend they are global constants, thus increasing readability.

We first introduce a generic framework oriented towards data flow analysis (§3.1). Within it we define and verify a standard data flow analyzer, Kildall's algorithm (§3.2). Finally we refine the framework towards a static type system (§3.3). In these following three sections let $(A,\ r,\ f)$ be a semilattice.

3.1 A generic data flow framework

Data flow analysis and type systems are based on an abstract view of the semantics of a program in terms of types instead of values. Since our programs are sequences of instructions the semantics can be characterized by two functions:

step $:: nat \Rightarrow \sigma \Rightarrow \sigma$ is the abstract execution function: *step p s* is the result of executing instruction at p starting in state s. In the literature *step p* is called the *transfer function* or *flow function* associated with instruction p.

succs $:: nat \Rightarrow nat\ list$ computes the possible successor instructions: *succs p* returns $[q_1, \ldots, q_k]$ iff execution of instruction p may transfer control to any of the instructions $q_1,\ \ldots,\ q_k$. We use lists instead of sets for reasons of executability.

We say that *succs* is **bounded by** n iff for all $p < n$ the elements of *succs* p are less than n. This expresses that from below instruction n, instruction n and beyond are unreachable, i.e. control never leaves the list of instructions below n.

Data flow analysis is concerned with solving data flow equations, i.e. systems of equations involving the flow functions over a semilattice. In our case *step* is the flow function and σ the semilattice. Instead of an explicit formalization of the data flow equation it suffices to consider certain prefixed points. To that end we define what it means that a method type *ss* is **stable at** p and **stable**:

$$\mathit{stable}\ ss\ p \equiv \forall\, q{\in}\mathit{set}(\mathit{succs}\ p).\ \mathit{step}\ p\ (ss!p) \le_r ss!q$$

$$\mathit{stables}\ ss\ \equiv \forall\, p{<}\mathit{size}\ ss.\ \mathit{stable}\ ss\ p$$

The stable method types are the prefixed points of the associated data flow equation for *step*

$$ss!p = \sum_{q\ \in\ \mathit{succs}^{-1}\ p} \mathit{step}\ q\ (ss!q)$$

where we assume that r is a semilattice and $\sum$ the corresponding supremum operation. Note that this equation has only been included here for clarity and is not part of our formalization. Our reference point will be the notion of stability. It induces the notion of a method type *ts* being a **welltyping w.r.t.** ***step***:

$$\mathit{wt\text{-}step}\ ts \equiv \forall\, p{<}\mathit{size}\ ts.\ ts!p \neq \top \wedge \mathit{stable}\ ts\ p$$

Here $\top$, another implicit parameter of *wt-step*, is assumed to be a special element in the state space indicating a type error. Usually $\top$ will be the top element of the ordering.

A welltyping is a witness of welltypedness in the sense of stability. Now we turn to the problem of computing such a witness. This is precisely the task of a bytecode verifier: it computes a method type such that the absence of $\top$ in the result means the method is welltyped. Formally, a function $\mathit{bcv} :: \sigma\ \mathit{list} \Rightarrow \sigma\ \mathit{list}$ is a **bytecode verifier** (w.r.t. $n :: \mathit{nat}$ and $A :: \sigma\ \mathit{set}$, see below) iff

$$\forall\, ss \in \mathit{list}\ n\ A.\ (\forall\, p{<}n.\ (\mathit{bcv}\ ss)!p \neq \top) = (\exists\, ts \in \mathit{list}\ n\ A.\ ss \le[r]\ ts \wedge \mathit{wt\text{-}step}\ ts)$$

The notation $\le[r]$ lifts $\le_r$ to lists (see §2.7). In practice, *bcv ss* itself will be the welltyping, and it will also be the least welltyping. However, it is simpler not to require this.

The introduction of a subset A of the state space σ is necessary to make distinctions beyond HOL's type system: for example, when representing a list of registers, σ is likely to be a HOL list type; but the fact that in any particular program the number of registers is fixed cannot be expressed as a HOL type, because it requires dependent types to formalize lists of a fixed length. We use sets to express such fine grained distinctions.

3.2 Kildall's algorithm

This section first defines and then verifies a functional version of Kildall's algorithm [19,20], a standard data flow analysis tool. In fact, the description of bytecode verification in the official JVM specification [13, pages 129–130] is essentially Kildall's algorithm, an iterative computation of the solution to a data flow problem. The main loop operates on a method type ss and a **worklist** w :: *nat set*. The worklist contains the indices of those elements of ss that have changed and whose changes still need to be propagated to their successors. Each iteration picks an element p from w, executes instruction number p, and propagates the new state to the successor instructions of p. Iteration terminates once w becomes empty: in each iteration, p is removed but new elements can be added to w. For reasons detailed below, the algorithm is expressed in terms of a predefined *while*-combinator of type $(\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$ which satisfies the recursion equation

$$while\ b\ c\ s = (if\ b\ s\ then\ while\ b\ c\ (c\ s)\ else\ s)$$

Clearly *while* $(\lambda s.\ b\ s)$ $(\lambda s.\ c\ s)$ is the functional counterpart of the imperative program `while b(s) do s := c(s)`. The main loop can now be expressed as

$$\begin{aligned}
iter\ ss\ w = while\ & (\lambda(ss,w).\ w \neq \{\}) \\
& (\lambda(ss,w).\ let\ p = SOME\ p.\ p \in w;\ t = step\ p\ (ss!p) \\
& \qquad\qquad in\ propa\ (succs\ p)\ t\ ss\ (w - \{p\})) \\
& (ss,w)
\end{aligned}$$

As in §3.1 the functions *step* and *succs* are implicit parameters.

The choice of which position to consider next is made by Hilbert's ε-operator, written *SOME* above: $SOME\ x.\ P\ x$ is some arbitrary but fixed x such that $P\ x$ holds; if there is no such x, then the value of $SOME\ x.\ P\ x$ is arbitrary but still defined. Since the choice in *iter* is guarded by $w \neq \{\}$, we know that $p \in w$. An implementation is free to choose whichever element it wants.

Propagating the result t of executing instruction number p to all successors is expressed by the primitive recursive function *propa*:

$$\begin{aligned}
&propa\ []\ t\ ss\ w = (ss,w) \\
&propa\ (q\#qs)\ t\ ss\ w = (let\ u = t +_f ss!q; \\
&\qquad\qquad\qquad\qquad\qquad w' = (if\ u = ss!q\ then\ w\ else\ \{q\} \cup w) \\
&\qquad\qquad\qquad\qquad in\ propa\ qs\ t\ (ss[q := u])\ w')
\end{aligned}$$

In the terminology of the official JVM specification[13, page 130] t is "merged" with the state of all successor instructions q, i.e. the supremum is computed. If this results in a change of $ss\ !\ q$ then q is inserted into w. The supremum operation f is another implicit parameter of the algorithm.

Kildall's algorithm is simply a call to *iter* where the worklist is initialized with the set of unstable indices; upon termination we project on the first (*fst*) component:

$$\mathit{kildall}\ ss = \mathit{fst}(\mathit{iter}\ ss\ \{p.\ p < \mathit{size}\ ss \wedge \neg \mathit{stable}\ ss\ p\})$$

Essentially the same algorithm was presented in [21], with one important difference: we did not show the actual definition of *iter*, because that was an opaque well-founded recursion. The recursion equation shown instead was what one would have written in a programming language with partial functions, but in HOL this is impossible, because *iter* is partial. It is only now that the *while*-combinator has become available that we can really define *iter* in a natural and directly executable form. The definition of *while* itself and the derivation of the corresponding recursion equation is not completely trivial; for details see [22].

The key theorem is that Kildall's algorithm is a bytecode verifier as defined above:

Theorem 1 *If (A, r, f) is a semilattice, r meets the ascending chain condition, step is monotone (w.r.t. A and n) and preserves A upto n, and succs is bounded by n, then kildall is a bytecode verifier.*

This theorem is a corollary to the following lemma about *iter*:

Lemma 2 *Assuming the same preconditions as in Theorem 1, if ss_0 is stable at all $p < n$ that are not in w_0, $ss_0 \in \mathit{list}\ n\ A$, w_0 is bounded in size by n, and iter ss_0 $w_0 = (ss',w')$, then*

$$\begin{array}{l} ss' \in \mathit{list}\ n\ A \wedge \mathit{stables}\ ss' \wedge ss_0 \le[r]\ ss' \wedge \\ (\forall ts{\in}\mathit{list}\ n\ A.\ ss_0 \le[r]\ ts \wedge \mathit{stables}\ ts \longrightarrow ss' \le[r]\ ts) \end{array}$$

This lemma is proved with the help of the usual proof rule for *while*-loops rephrased functionally in terms of the *while*-combinator. That theorem is already part of the Isabelle library. The invariant is the following predicate on ss and w:

$$\begin{array}{l} ss \in \mathit{list}\ n\ A \wedge (\forall p{<}n.\ p \notin w \longrightarrow \mathit{stable}\ ss\ p) \wedge ss_0 \le[r]\ ss \wedge \\ (\forall ts{\in}\mathit{list}\ n\ A.\ ss_0 \le[r]\ ts \wedge \mathit{stables}\ ts \longrightarrow ss \le[r]\ ts) \wedge (\forall p{\in}w.\ p < n) \end{array}$$

Except for the last conjunct, this is identical to the desired postcondition. Hence it is easy to see that upon termination ($w = \{\}$) the postcondition holds. Similarly, it is easy to see that the invariant holds initially, i.e. for $ss = ss_0$ and $w = w_0$. Preservation of the invariant requires a number of lemmas about *propa* which are not interesting enough to warrant showing them here. Termination is proved with the help of the lexicographic product of two well-founded orderings: with every iteration, either *ss* increases w.r.t. $\leq[r]$ (hence we need the ascending chain condition), or *ss* stays unchanged but *w* decreases (and *w* is finite because it is bounded by *n*).

3.3 Refining step

We close this section with a little refinement of the generic *step*. In practice, *step* will be composed of two functions: one that checks the applicability of the instruction in the current state, and one that carries out the instruction assuming it is applicable. These two functions will be called *app* and *eff*. Furthermore the state space σ will be of the form τ *err* for a suitable type τ. In which case the error element $\top$ above is *Err* itself. Given functions $app :: nat \Rightarrow \tau \Rightarrow bool$ and $eff :: nat \Rightarrow \tau \Rightarrow \tau$, *step* is easily defined:

$$step\ p \equiv lift\ (\lambda t.\ if\ app\ p\ t\ then\ OK(eff\ p\ t)\ else\ Err)$$

Similarly, we can refine the notion of welltyping w.r.t. *step* to welltyping w.r.t. *app* and *eff*:

$$wt\text{-}app\text{-}eff\ ts \equiv \forall\, p{<}size\ ts.\ app\ p\ (ts!p) \wedge (\forall\, q{\in}set(succs\ p).\ eff\ p\ (ts!p) \leq_r ts!q)$$

This is very natural: every instruction is applicable in its start state, and the effect is compatible with the state expected by all successor instructions.

It is almost but not quite the case that *wt-step* and *wt-app-eff* coincide. We have proved the following two lemmas:

Lemma 3 *If succs is bounded by size ts and every instruction has a successor, then wt-step ts implies wt-app-eff (map ok-val ts) where ok-val (OK x) = x.*

Why the requirement that every instruction must have a successor? Otherwise *ts* may be a welltyping w.r.t. *step* for the trivial reason that *stable ts p* is vacuously true if $succs\ p = \{\}$, even though $app\ p\ (ts!p)$ may be false. In the other direction, this problem disappears:

Lemma 4 *If succs is bounded by size ts, then wt-app-eff ts implies that wt-step (map OK ts).*

In a sense *wt-app-eff* is more precise than *wt-step*. This is not surprising: it has more structure to work with. We can still model everything that is captured by *wt-app-eff* with *wt-step*, though: lemma 3 tells us that they agree when each instruction has an successor, so for instructions with no successor we just define *succs* $p = \{p\}$ and *eff* $p = id$. This does not change the behaviour of the BCV and makes sure that both formulations are equivalent.

4 JVM

In the sections above we have presented an abstract definition of welltyping and also a means to calculate one. We will now instantiate these abstract notions for the μJava Virtual Machine. The instantiation presented here is a bit different from the one we already gave in [21]. The version we will discuss here is a direct integration with our existing μJava formalization. The theory in [21] was based on μJava in that it used the same concepts, instruction set etc, but formally it was a separate theory. We will first give an overview of the μJava formalization—see [2] for an in depth introduction to a slightly earlier version of it—and then discuss how it works together with the definitions of the bytecode verifier framework.

The μJava model is a downsized version of the real Java language and JVM. It does not contain threads, interfaces, packages, or visibility modifiers like `public` or `private`. It does contain a small but representative instruction set, classes, inheritance, and object oriented method invocation. The formalization includes the source language, its operational semantics and type system together with a proof of type safety, and also the bytecode language together with its operational semantics, type system, proof of type safety, and now also an executable verified bytecode verifier. To keep things abstract and manageable, the source and bytecode language share as many notions as possible, i.e. program structure, wellformedness of programs, large parts of the type system, etc.

4.1 The μJava VM

Since source and target languages share the same model of program structure, the virtual machine of μJava does not have a notion of class files and constant pool like Sun's JVM has. A program is a list of classes and their declaration information, a class a list of methods and fields. For bytecode verification we will only be concerned with the method level, so we only describe the types at this level in more detail:

$$\begin{array}{rcl} \gamma\ \mathit{mdecl} & = & \mathit{sig} \times \mathit{ty} \times \gamma \\ \mathit{sig} & = & \mathit{mname} \times \mathit{ty}\ \mathit{list} \end{array}$$

Since they are used for both the source and the bytecode level, μJava method declarations are parameterized: they consist of a signature (name and parameter types), the return type (*ty* are the Java types), and a method body of type γ (which in our case will be the bytecode).

Method declarations come with a method dictionary lookup function *method*, such that *method* (Γ, C) *sig* looks up a method with signature *sig* in class C of program Γ. It yields a value of type (*cname* $\times$ *ty* $\times$ γ) *option* indicating whether a method with that signature exists, in which class it is defined (it could be a superclass of C since *method* takes inheritance and overriding into account), and also the rest of the declaration information: the return type and body.

The runtime environment, i.e. the state space of the μJVM, is modeled a bit more closely after the real thing. The state consists of a heap, a stack of call frames, and a flag whether an exception was raised (and if yes, which one).

The heap is simple: a partial function from locations to objects, in Isabelle *aheap* = *loc* $\Rightarrow$ *obj option*. Here, *loc* is the type of addresses and *obj* is short for *cname* $\times$ (*vname* $\times$ *cname* $\Rightarrow$ *val option*), i.e. each object is a pair consisting of a class name (the class the object belongs to) and a mapping for the fields of the object (taking the name and defining class of a field, and yielding its value if such a field exists, *None* otherwise).

In the μJVM each method execution gets its own call frame, containing its own operand stack (a list of values), its own set of local variables (also a list of values), and its own program counter. Since we later need to know which method each call frame belongs to, we also store the class and signature (i.e. name and parameter types) of the method and arrive at:

$$\begin{array}{rcl} \mathit{frame} & = & \mathit{opstack} \times \mathit{locvars} \times \mathit{cname} \times \mathit{sig} \times \mathit{nat} \\ \mathit{opstack} & = & \mathit{val\ list} \\ \mathit{locvars} & = & \mathit{val\ list} \end{array}$$

The μJVM state is then:

$$\begin{array}{rcl} \mathit{jvm\text{-}state} & = & \mathit{xcpt\ option} \times \mathit{aheap} \times \mathit{frame\ list} \\ \mathit{xcpt} & = & \mathit{NullPointer} \mid \mathit{ClassCast} \mid \mathit{OutOfMemory} \end{array}$$

4.2 Operational semantics

This section sketches the state transition relation of the μJava VM. We will be relatively brief here and only concentrate on the parts we need for the BCV.

Figure 2 shows the instruction set. Method bodies are lists of such instructions together with two integers *mxs* and *mxl* containing the maximum operand stack size and the number of local variables (not counting the *this* pointer and parameters of the method which get stored in the first *0* to *n* registers). So the type parameter γ for method bodies gets instantiated with $\mathit{nat} \times \mathit{nat} \times \mathit{instr\ list}$, i.e. $\mathit{mdecl} = \mathit{sig} \times \mathit{ty} \times \mathit{nat} \times \mathit{nat} \times \mathit{instr\ list}$.

datatype instr =

	Load nat	load from local variable
|	*Store nat*	store into local variable
|	*LitPush val*	push a literal (constant)
|	*New cname*	create object on heap
|	*Getfield vname cname*	fetch field from object
|	*Putfield vname cname*	set field in object
|	*Checkcast cname*	check if object is of given type
|	*Invoke cname mname (ty list)*	invoke instance method
|	*Return*	return from method
|	*Dup*	duplicate top element
|	*IAdd*	integer addition
|	*Goto int*	goto relative address
|	*Ifcmpeq int*	branch if top elements are equal

Fig. 2. The μJava instruction set

The state transition relation $s \xrightarrow{\text{jvm}} t$ is built on a function *exec* describing one-step execution:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{exec} :: \mathit{jvm\text{-}state} \Rightarrow \mathit{jvm\text{-}state\ option}} \\
\mathit{exec}\ (\mathit{xp},\ \mathit{hp},\ []) & = & \mathit{None} \\
\mathit{exec}\ (\mathit{Some\ xp},\ \mathit{hp},\ \mathit{frs}) & = & \mathit{None} \\
\mathit{exec}\ (\mathit{None},\ \mathit{hp},\ f\#\mathit{frs}) & = & \mathit{let}\ (\mathit{stk},\mathit{loc},C,\mathit{sig},\mathit{pc}) = f; \\
 & & \quad i = (\mathit{5th}\ (\mathit{the}\ (\mathit{method}\ (\Gamma,C)\ \mathit{sig})))\ !\ \mathit{pc} \\
 & & \mathit{in\ Some}\ (\mathit{exec\text{-}instr}\ i\ \mathit{hp}\ \mathit{stk}\ \mathit{loc}\ C\ \mathit{sig}\ \mathit{pc}\ \mathit{frs})
\end{array}
$$

It says that execution halts if the call frame stack is empty or an exception has occurred. In all other cases execution is defined, *exec* decomposes the top call frame, looks up the current method, retrieves the instruction list (the 5th element) of that method, and delegates the rest to *exec-instr* which defines the execution rules for single instructions. As throughout the rest of this article, the program Γ is treated as global parameter.

The state transition relation is the reflexive transitive closure of the defined part of *exec*:

$$s \xrightarrow{\text{jvm}} t = (s,t) \in \{(s,t) \mid \mathit{exec}\ s = \mathit{Some}\ t\}^*$$

The definition of *exec-instr* is straightforward for most instructions. In the *Load idx* case for instance, we just take the value at position *idx* in the local variables and put it on top of the stack. Apart from incrementing the program counter the rest remains untouched:

```
exec-instr (Load idx) hp stk vars Cl sig pc frs =
   (None, hp, ((vars ! idx) # stk, vars, Cl, sig, pc+1)#frs)
```

Since they will recur in a slightly different form in §5.2 we also show the rules for *New*, *Invoke* and *Return*:

```
exec-instr (New C) hp stk vars Cl sig pc frs =
let (ref,xp') = new-Addr hp;
          hp' = if xp'=None then hp(ref := Some (blank C)) else hp;
         stk' = if xp'=None then (Addr ref)#stk else stk
in (xp', hp', (stk', vars, Cl, sig, pc+1)#frs)
```

The *New* rule uses two helper functions *new-Addr* :: *aheap* ⇒ *loc* × *xcpt option* and *blank* :: *cname* ⇒ *obj*. The former returns a new, unused address on the heap or an *OutOfMemory* exception. The latter constructs a blank object with all fields set to their default values. With these, the *New C* instruction either sets the exception flag to *OutOfMemory* or constructs a new object on the heap and puts its address on top of the stack.

Invoke on a method with name *mn* of class *C* with formal parameter types *ps* is a bit more involved:

```
exec-instr (Invoke C mn ps) hp stk vars Cl sig pc frs =
let                  n = size ps;
                  args = take n stk;
                   ref = stk!n;
                   xp' = raise-xcpt (ref=Null) NullPointer;
                     D = fst(the(hp(the-Addr ref)));
     (dc, _, _, mxl, _) = the (method (Γ,D) (mn,ps));
                  frs' = if xp'≠None then [] else
                         [([],ref#rev args@replicate mxl arbitrary,dc,(mn,ps),0)]
in  (xp', hp, frs'@(stk, vars, Cl, sig, pc)#frs)
```

The first *n* values on the stack are the actual parameters, the next value after that is the reference on which to invoke the method. If it is *Null*, the flag is set to *NullPointer* exception and execution will stop. Otherwise, we look up the dynamic type of the object (which could be a subclass of *C*) on the heap, and do a method lookup with this dynamic type to find out in which class *dc* the method really is defined.

With these values we construct a new call frame for the new method: initially, it has an empty stack, a reference to *this* in local variable *0*, the actual parameters in variables *1* to *n*, and the rest of the local variables filled with an arbitrary value. We fill in the defining class *dc*, the signature of the method, and set the *pc* to instruction *0*. The new call frame is put in front of the current one which remains unchanged until the newley invoked method returns.

If execution of the new method ends, it must end with a *Return* instruction:

```
exec-instr Return G hp stk$_0$ vars Cl sig$_0$ pc frs =
  if frs=[] then (None, hp, []) else
  let val = hd stk$_0$;
      (stk,loc,C,sig,pc) = hd frs;
      n = length (snd sig$_0$)
  in (None, hp, (val#(drop (n+1) stk),loc,C,sig,pc+1)#tl frs)
```

Return does nothing if the call frame is already empty. If it is not, the return value is the top of stack in the current call frame, the current call frame is removed and the frame directly beneath (the calling method) is updated: we drop the parameters and object reference of the *Invoke* instruction from the stack, push the return value, and increment the *pc*.

These rules obviously do not implement a defensive VM. Nowhere do we check if values have the right type, if the operand stack contains values at all when we refer to its top, if a local variable with a given index even exists, etc. Since HOL is a logic of total functions, execution in all these cases is still defined. However, we do not know anything about such defined but unspecified states, and we will certainly not be able to prove any interesting properties like type conformance about them.

4.3 Bytecode verification and type safety

This section presents the BCV specification for μJava, and its integration with the executable BCV framework of §3.2.

We will proceed as follows: first, we again take a look at the state space—this time with types instead of values since we are interested in abstract execution only. Then, we will turn this state space into a semilattice which gives us an ordering on it and a supremum operation. We will use the ordering to describe welltypings statically. This part is taken from the existing μJava formalization, it comes with a type safety result about welltypings. The supremum operation is the part we additionally need for Kildall's algorithm. We will see that the result of Kildalls algorithm agrees with the specification of welltypings. This finally implies that the type safety results for μJava welltypings also apply to the executable BCV.

The μJava BCV specification describes how welltypings of operand stack and local variables of a method look like (as in the example figure 1 at the very beginning). Values on the operand stack must always contain a known type, the type of values in the local variables may be unknown (encoded by *Err*):

$$\mathit{state\text{-}type} \;=\; \mathit{ty\ list} \times \mathit{ty\ err\ list}$$

$$\begin{array}{lcl}
\mathit{datatype\ ty} & = & \mathit{PrimT\ prim\text{-}ty} \mid \mathit{RefT\ ref\text{-}ty} \\
\mathit{datatype\ prim\text{-}ty} & = & \mathit{Void} \mid \mathit{Bool} \mid \mathit{Int} \\
\mathit{datatype\ ref\text{-}ty} & = & \mathit{NullT} \mid \mathit{ClassT\ cname}
\end{array}$$

In this definition, *ty* is the HOL type describing the μJava type system. It consists of primitive types like *Int* or *Void*, and references types. We abbreviate the reference type *RefT* (*ClassT C*) by *Class C* and *RefT NullT*, the type of the value *Null*, by *NT*.

The μJava types come with a subtype ordering $\preceq$ that builds on the direct subclass relation *subcls* Γ induced by the program Γ. It satisfies:

$$\begin{array}{rcll}
T & \preceq & T & \\
NT & \preceq & \mathit{RefT}\ T & \\
\mathit{Class}\ C & \preceq & \mathit{Class}\ D & \text{if } (C,D) \in (\mathit{subcls}\ \Gamma)^*
\end{array}$$

where $(C,D) \in (\mathit{subcls}\ \Gamma)^*$ means that C is a subclass of D.

For every class hierarchy, that is for every program, this subtype ordering may be a different one. In the Isabelle formalization the ordering $\preceq$ therefore has Γ as an additional parameter, in this paper Γ is an implicit parameter of everything.

To handle unreachable code, the BCV will not directly work on *state-type*, but on *state-type option* instead. In the dataflow analysis *None* will indicate the state type of instructions that have not been reached yet. If *None* occurs in the welltyping, the corresponding instruction is unreachable.

4.3.1 Java Types as Semilattice

If we want to instantiate Kildall's algorithm we need to turn first *ty* and then *state-type* into a semilattice.

Theory *JType* concerns the semilattice structure of *ty*: the ordering we already have with $\preceq$. For the carrier set *types* we take the set of all primitive types and all subclasses of *Object* defined in the program. The supremum operation *sup* is also relatively straightforward:

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathit{sup} :: \mathit{ty} \Rightarrow \mathit{ty} \Rightarrow \mathit{ty\ err}} \\
\mathit{sup}\ NT & (\mathit{Class}\ C) & = & \mathit{OK}\ (\mathit{Class}\ C) \\
\mathit{sup}\ (\mathit{Class}\ C) & NT & = & \mathit{OK}\ (\mathit{Class}\ C) \\
\mathit{sup}\ (\mathit{Class}\ C) & (\mathit{Class}\ D) & = & \mathit{OK}\ (\mathit{Class}\ (\mathit{some\text{-}lub}\ C\ D)) \\
\mathit{sup}\ t_1 & t_2 & = & \mathit{if}\ t_1 = t_2\ \mathit{then}\ \mathit{OK}\ t_1\ \mathit{else}\ \mathit{Err}
\end{array}$$

The auxiliary function *some-lub* used in the computation of the supremum of two classes is defined non-constructively (as some least upper bound, using Hilbert's ε-operator). Of course we also prove that (under suitable conditions) least upper bounds are uniquely determined and exist. Thus our work is independent of the particular algorithm used for this calculation.

Theorem 5 *The triple esl* $\equiv$ (*types*, $\preceq$, *sup*) *is an err-semilattice provided the class hierarchy subcls* Γ *is* ***single valued*** *(each subclass has at most one direct superclass, i.e.* Γ *represents a single inheritance hierarchy) and subcls* Γ *is acyclic.*

Univalence and acyclicity together imply that *subcls* Γ is a set of trees, and *types* focuses on the subtree below *Object*.

Because any infinite subtype chain would induce an infinite subclass chain we also obtain

Lemma 6 *If* $(subcls\ \Gamma)^{-1}$ *is wellfounded (there is no infinite ascending subclass chain* $(C_i, C_{i+1}) \in subcls\ \Gamma$*) then* $\preceq$ *satisfies the ascending chain condition.*

We call Γ **wellformed** if *subcls* Γ is single valued and $(subcls\ \Gamma)^{-1}$ is wellfounded.

Note that by incorporating a fixed class hierarchy into our model we assume that all required classes have been loaded, i.e. we model an eager class loader. Although Sun's JVM is a bit lazier, the JVM specification [13, p. 127] does permit eager loading.

Now we come to the core of the type system, namely the semilattice structure τ on *state-type option*. Turning *state-type option* into a semilattice is easy, because all of its constituent types are (err-)semilattices. The expression stacks form an err-semilattice because the supremum of stacks of different size is *Err*; the list of registers forms a semilattice because the number *mxr* of registers is fixed (it is the number *mxl* of local variables plus the number of parameters of the method plus one for the *this* pointer):

$stk\text{-}esl :: nat \Rightarrow ty\ list\ esl$
$stk\text{-}esl\ mxs \equiv upto\text{-}esl\ mxs\ (JType.esl)$

$reg\text{-}sl :: nat \Rightarrow ty\ err\ list\ sl$
$reg\text{-}sl\ mxr \equiv Listn.sl\ mxr\ (Err.sl\ (JType.esl))$

The stack and registers are combined in a coalesced product via *Product.esl* and then embedded into *option* and *err* (as in §3.3) to form the final semilattice for the state space σ = *state-type option err*:

$sl :: nat \Rightarrow nat \Rightarrow state\text{-}type\ option\ err\ sl$
$sl\ mxs\ mxr \equiv Err.sl(Opt.esl(Product.esl\ (stk\text{-}esl\ mxs)\ (Err.esl(reg\text{-}sl\ mxr))))$

The three components of *sl* are called *states* (the carrier set), *le* (the ordering), and *sup* (the supremum).

Combining the theorems about the various (err-)semilattice constructions involved in the definition of *sl* (starting from Theorem 5), it is easy to prove

Corollary 7 *If* Γ *is wellformed then sl is a semilattice.*

It is trivial to show that $\top = Err$ is the top element of this semilattice.

4.3.2 Type checking

The semilattice construction above gives us an ordering

$\le' :: state\text{-}type\ option \Rightarrow state\text{-}type\ option \Rightarrow bool$

which is already sufficient to describe type checking.

Type checking is what the original μJava BCV specification does: it describes welltypings by stating conditions for correct method types. In contrast to the type inference in Kildall's Algorithm, we do not need a supremum operatiom for this.

Since the μJava specification predates our work about executable bytecode verifiers, it looks a bit different from our definition of welltypings *wt-app-eff* in §3.3. Nevertheless Theorem 10 will connect *wt-app-eff* to *wt-method* and show that it exactly describes welltypings φ :: *state-type option list* of methods:

$wt\text{-}instr\ i\ \varphi\ pc \equiv$
$\quad app\ i\ (\varphi!pc) \wedge$
$\quad (\forall pc' \in set\ (succs\ i\ pc).\ pc' < size\ ins \wedge (eff\ i\ (\varphi!pc) \le' \varphi!pc'))$

$wt\text{-}start\ \varphi \equiv$
$\quad Some\ ([],(OK\ (Class\ C))\#(map\ OK\ ps)@(replicate\ mxl\ Err)) \le' \varphi!0$

$wt\text{-}method\ \varphi \equiv$
$\quad 0 < size\ ins \wedge wt\text{-}start\ \varphi \wedge (\forall pc.\ pc < size\ ins \longrightarrow wt\text{-}instr\ (ins!pc)\ \varphi\ pc)$

Again we have some implicit parameters: *C* is the class where the method under scrutiny is defined, *ps* the types of its formal parameters, *ins* the instruction list (method body), and *mxl* the number of local variables. Apart from *wt-start* and the additional condition that the instruction list may not be empty, *wt-instr* and *wt-method* together coincide nicely with *wt-app-eff* and the fact that *succs* is bounded by *size ins*. Corresponding to the start value of the iteration in Kildall's algorithm, *wt-start* places a restriction on the welltyping of instruction *0*.

The type checking specification is defined in terms of the functions *app*, *eff* and *succs* as introduced in §3.3. They can directly be reused later for Kildall's algorithm.

The definition of *succs* is straightforward:

$$\begin{array}{lcl} \textit{succs} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{nat list} & & \\ \textit{succs}\ (\textit{Ifcmpeq}\ b)\ \textit{pc} & = & [\textit{pc}{+}\textit{1},\ \textit{nat}\ (\textit{pc}{+}b)] \\ \textit{succs}\ (\textit{Goto}\ b)\ \textit{pc} & = & [\textit{nat}\ (\textit{pc}{+}b)] \\ \textit{succs Return pc} & = & [\textit{pc}] \\ \textit{succs instr pc} & = & [\textit{pc}{+}\textit{1}] \end{array}$$

Since the offset b in branching instructions may be negative, we need to turn the result of *pc*+*b* back to type *nat* by explicit conversion. The successor of *Return* is again *pc* since lemma 3 requires *succs* to be nonempty for all instructions. The last equation is the default: we just increment the *pc* by one.

Figure 3 and 4 show the definitions of *app* and *eff*. Both are first defined via *app'* and *eff'* on *state-type* and then lifted to *state-type option*. They use the program Γ, the maximum size of the operand stack *mxs*, and the return type *rt* of the current method as implicit parameters.

In *app'* a few new functions occur: *field* is analogous to *method* and looks up declaration information of object fields (defining class and type); apart from that there are only the obvious functions on lists $\textit{rev} :: \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}$, $\textit{zip} :: \alpha\ \textit{list} \Rightarrow \beta\ \textit{list} \Rightarrow (\alpha \times \beta)\ \textit{list}$, and $\textit{take} :: \textit{nat} \Rightarrow \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}$.

In *eff'* we use $\textit{typeof} :: \textit{val} \Rightarrow \textit{ty option}$ returning *None* for addresses, and the type of the value otherwise. The *method* expression for *Invoke* merely determines the return type of the method in question. The lifting functional $\textit{option-map} :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ \textit{option} \Rightarrow \beta\ \textit{option}$ satisfies the two equations $\textit{option-map}\ f\ \textit{None} = \textit{None}$ and $\textit{option-map}\ f\ (\textit{Some}\ x) = \textit{Some}\ (f\ x)$.

4.3.3 Type inference

If we construct *step* as in §3.3 from *app* and *eff*, we can instantiate Kildall's algorithm for the μJava VM:

$$\begin{array}{l} \textit{kiljvm} :: \textit{instr list} \Rightarrow \textit{state-type option err list} \Rightarrow \textit{state-type option err list} \\ \textit{kiljvm ins} \equiv \textit{kildall le sup}\ (\textit{step ins})\ (\lambda \textit{pc}.\ \textit{succs}\ (\textit{ins}!\textit{pc})\ \textit{pc}) \end{array}$$

The proof of the following theorem is easy:

$$
\begin{array}{ll}
\mathit{app}' :: \mathit{instr} \times \mathit{state\text{-}type} \Rightarrow \mathit{bool} & \\
\mathit{app}'\ (\mathit{Load}\ \mathit{idx},\ (\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size}\ \mathit{lt} \land \mathit{lt}!\mathit{idx} \neq \mathit{Err} \land \\
 & \quad \mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Store}\ \mathit{idx},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size}\ \mathit{lt} \\
\mathit{app}'\ (\mathit{LitPush}\ v,\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{st} < \mathit{mxs} \land \mathit{typeof}\ v \neq \mathit{None} \\
\mathit{app}'\ (\mathit{Getfield}\ F\ C,\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \land\ t \preceq \mathit{Class}\ C \land \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,t')) \\
\mathit{app}'\ (\mathit{Putfield}\ F\ C,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \land \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}(C,t') \land \\
 & \quad t_2 \preceq \mathit{Class}\ C \land t_1 \preceq t') \\
\mathit{app}'\ (\mathit{New}\ C,\ (\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \land \mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Checkcast}\ C,\ (\mathit{RefT}\ t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \\
\mathit{app}'\ (\mathit{Dup},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{1}+\mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{IAdd},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = \mathit{PrimT}\ \mathit{Int} \land t_2 = \mathit{PrimT}\ \mathit{Int} \\
\mathit{app}'\ (\mathit{Ifcmpeq}\ b,\ (t\#t'\#\mathit{st},\mathit{lt})) & = (t = t') \lor \\
 & \quad (\exists\, r\ r'.\ t = \mathit{RefT}\ r \land t' = \mathit{RefT}\ r') \\
\mathit{app}'\ (\mathit{Goto}\ b,\ s) & = \mathit{True} \\
\mathit{app}'\ (\mathit{Return},\ (t\#\mathit{st},\mathit{lt})) & = t \preceq \mathit{rt} \\
\mathit{app}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st} \land \\
 & \quad \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \land \\
 & \quad \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st}); \\
 & \quad \quad t \ \ = \mathit{st}!\mathit{size}\ \mathit{ps} \\
 & \quad \mathit{in}\ t \preceq \mathit{Class}\ C \land \mathit{is\text{-}class}\ \Gamma\ C \land \\
 & \quad \quad (\forall\,(a,f) \in \mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq f) \\
\mathit{app}'\ (i,s) & = \mathit{False}
\end{array}
$$

$$
\begin{array}{l}
\mathit{app} :: \mathit{instr} \Rightarrow \mathit{state\text{-}type}\ \mathit{option} \Rightarrow \mathit{bool} \\
\mathit{app}\ i\ s \equiv \mathit{case}\ s\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{True} \mid \mathit{Some}\ t \Rightarrow \mathit{app}'\ (i,t)
\end{array}
$$

Fig. 3. Applicability of instructions

Lemma 8 *step is monotone and preserves the carrier set.*

With Theorem 1 we therefore get

Corollary 9 *If* Γ *is wellformed and succs is bounded by the number of instructions then kiljvm is a bytecode verifier.*

We can further define

$$
\begin{array}{l}
\mathit{wt\text{-}kil} \equiv \mathit{bounded}\ (\lambda pc.\ \mathit{succs}\ (\mathit{ins}!pc)\ pc)\ (\mathit{size}\ \mathit{ins}) \land \mathit{0} < \mathit{size}\ \mathit{ins} \land \\
\qquad \mathit{let}\ S_0 = \mathit{Some}\ ([],(\mathit{OK}\ (\mathit{Class}\ C))\#(\mathit{map}\ \mathit{OK}\ \mathit{ps})@(\mathit{replicate}\ \mathit{mxl}\ \mathit{Err})); \\
\qquad \quad \varphi_0 = (\mathit{OK}\ S_0)\#(\mathit{replicate}\ (\mathit{size}\ \mathit{ins}-\mathit{1})\ (\mathit{OK}\ \mathit{None})); \\
\qquad \mathit{in}\ \forall\, n < \mathit{size}\ \mathit{ins}.\ (\mathit{kiljvm}\ \varphi_0)!n \neq \mathit{Err}
\end{array}
$$

This definition is again in the context of a method, i.e. it uses the implicit parameters *ins* (instruction sequence), C (defining class), *ps* (formal parameter

$$
\begin{array}{lcl}
\mathit{eff}' :: \mathit{instr} \times \mathit{state\text{-}type} \Rightarrow \mathit{state\text{-}type} & & \\
\mathit{eff}'\ (\mathit{Load\ idx},\ (\mathit{st},\mathit{lt})) & = & (\mathit{ok\text{-}val}\ (\mathit{lt}!\mathit{idx})\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Store\ idx},\ (t\#\mathit{st},\mathit{lt})) & = & (\mathit{st},\mathit{lt}[\mathit{idx}{:=}\ \mathit{OK}\ t]) \\
\mathit{eff}'\ (\mathit{LitPush}\ v,\ (\mathit{st},\mathit{lt})) & = & (\mathit{the}\ (\mathit{typeof}\ v)\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Getfield}\ F\ C,\ (t\#\mathit{st},\mathit{lt})) & = & (\mathit{snd}\ (\mathit{the}\ (\mathit{field}\ (\Gamma,C)\ F))\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Putfield}\ F\ C,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = & (\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{New}\ C,\ (\mathit{st},\mathit{lt})) & = & (\mathit{Class}\ C\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Checkcast}\ C,\ (t\#\mathit{st},\mathit{lt})) & = & (\mathit{Class}\ C\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Dup},\ (t\#\mathit{st},\mathit{lt})) & = & (t\#t\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{IAdd},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = & (\mathit{PrimT\ Int}\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Ifcmpeq}\ b,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = & (\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Goto}\ b,\ s) & = & s \\
\mathit{eff}'\ (\mathit{Return},\ s) & = & s \\
\mathit{eff}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = & \mathit{let}\quad \mathit{st}' = \mathit{drop}\ (1{+}\mathit{size}\ \mathit{ps})\ \mathit{st}; \\
 & & (\_,\mathit{rt},\_,\_,\_) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps})) \\
 & & \quad \mathit{in}\ (\mathit{rt}\#\mathit{st}',\mathit{lt})
\end{array}
$$

$$
\begin{array}{l}
\mathit{eff} :: \mathit{instr} \Rightarrow \mathit{state\text{-}type\ option} \Rightarrow \mathit{state\text{-}type\ option} \\
\mathit{eff}\ i \equiv \mathit{option\text{-}map}\ (\lambda s.\ \mathit{eff}'\ (i,s))
\end{array}
$$

Fig. 4. Effect of instructions on the state type

types), and *mxl* (number of local variables). From corollary 9 together with lemmas 3 and 4 we obtain the following theorem.

Theorem 10 *If the program* Γ *is wellformed, if* φ_0 *is defined as in wt-kil, and if wt-kil holds for a method, then wt-method* ($\mathit{kiljvm}\ \varphi_0$) *holds for the same method.*

Although useful on its own in the form above, theorem 10 becomes clearer rephrased: if Γ is wellformed and *wt-kil* holds for a method, then this method is type correct (there exists a welltyping φ such that *wt-method* φ). It connects the executable BCV with the static description of welltypings.

4.3.4 Type safety

The most interesting property we have proved about *wt-method* is type safety. So far we have considered an abstraction of the JVM that works on types rather than values. It remains to show that this abstraction is faithful, i.e. that welltypedness on this abstract level actually guarantees the absence of type errors in the concrete μJVM operating on values.

The original μJava BCV specification is by Pusch [3] who showed that it implies type soundness of the concrete μJVM. Roughly speaking, she proved that during execution of welltyped code, all values conform to the types given in the welltyping. That is, she defined a conformance relation *correct-state* $s\ \Phi$

between a concrete machine state s (of HOL type *jvm-state*) and a program type Φ :: *cname* $\Rightarrow$ *sig* $\Rightarrow$ *state-type option list* (φ lifted to programs, see also §A.1 for the definition conformance). And she proved

$$\textit{wt-jvm-prog}\ \Phi \wedge \textit{correct-state}\ s\ \Phi \wedge s \xrightarrow{\text{jvm}} t \longrightarrow \textit{correct-state}\ t\ \Phi$$

where *wt-jvm-prog* is again *wt-method* lifted to programs, i.e. *wt-jvm-prog* holds if *wt-method* holds for all methods in all classes of Γ. The theorem says that a welltyping guarantees type safe execution.

We can also characterize concrete starting states s for which execution is type safe without referring to the calculated welltyping at all: if *wt-kil* holds then it suffices to look at the start value φ_0 defined in *wt-kil*. States s satisfying *correct-state* s Φ_0 (where Φ_0 is again φ_0 lifted to programs) will also satisfy *correct-state* s Φ since the result Φ of Kildall's algorithm lies above its start value and *correct-state* is monotone.

5 Object Initialization

With *object initialization* we address a particular feature of the Java bytecode verifier: the test whether each new object is properly initialized before it is used. This not only includes a guarantee that for each object its constructor is called before its fields or methods are accessed, but also that each constructor calls the constructor of the object's superclass before it returns or begins with the rest of the initialization process.

Why care at all about object initialization? The question is not as rhetoric as it may seem—object initialization is not necessary for the type safety of the language, the default values of fields do nicely. After all, we have just claimed in section 4 that μJava is type safe and there was no mention of object initialization anywhere. The feature is interesting because large parts of Java's security mechanisms depend on constructors being called before objects are used and also on superclass constructors being called before the own constructor is executed. In this way and together with access modifiers secure, consistent object states can be guaranteed.

The purpose of this section is twofold. On the one hand it presents the formalization of an additional important and nontrivial feature of the Java BCV together with its proof of correctness. On the other hand it demonstrates the flexibility of our BCV framework by showing how it deals with significant changes and extensions to the specification of the existing μJava BCV. Not only do we now want to prove stronger properties of the bytecode verifier, but also the type system is now a new one, and the instruction set along with its operational semantics is slightly different.

We extend our model in four steps: First we explain how object initialization works and define a type system for it in §5.1. Then we proceed in §5.2 with the changes to the VM and the operational semantics. In §5.3 we define the BCV for object initialization. Finally, type safety relates the latter two in §5.4.

5.1 The new type system

The JVM specification [13, pages 131–133] gives a short description how to check for proper object initialization. Figure 5 is a typical piece of bytecode for the object creation/constructor call cycle as it is produced by common Java compilers (the example from [13] translated to μJava).

...	
New A	Allocate new space for class A
Dup	Duplicate reference on the stack
Load 0	Push constructor parameter
Invoke−special A (*init*, [*Class B*])	Invoke constructor on class A
...	

Fig. 5. A typical object creation/constructor call code piece

The instruction sequence first allocates space for the still uninitialized object, and then duplicates the reference to that object for the constructor call. After the constructor call the reference to the newly created and initialized object is on top of the operand stack.

To deal with that kind of situations, the JVM specification proposes to introduce two new artificial types that mark not yet initialized values; we will call them *UnInit* and *PartInit*.

As the name suggests, *UnInit* stands for uninitialized, freshly created objects. The reference on top of the stack after the *New A* instruction would get the type *UnInit A*. After the constructor has been called on that reference the object is initialized and we can replace *UnInit A* by our usual type *Class A*. As the JVM specification [13, pages 132–133] points out, that is not enough, though:

> The instruction number needs to be stored as part of the special type, as there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. For example, the Java virtual machine instruction sequence that implements
> `new InputStream(new Foo(), new InputStream("foo"))`
> may have two uninitialized instances of InputStream on the operand stack at

once. When an instance initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the local variable array that are the same object as the class instance are replaced.

By storing the program counter of the instruction that created a reference, we in fact implement a simple alias analysis: the type entry *UnInit C pc* keeps track of one specific value (the reference created by the instruction *New C* at position *pc*).

The *PartInit* type is easier: it marks the type of local variable *0* (the *this* pointer) in constructors. It can be replaced by the normal type of the class as soon as the superclass constructor has been invoked.

The following datatype definition captures the above formally; the constructor *Init ty* stands for the normal, initialized Java types.

$$\mathit{datatype\ init\text{-}ty} = \mathit{Init\ ty} \mid \mathit{UnInit\ cname\ nat} \mid \mathit{PartInit\ cname}$$

Our new type system in the BCV has *init-ty* wherever there was *ty* before, i.e. on the stack, or beneath the *OK*/*Err* structure in the local variables.

The BCV is also required to check that the superclass constructor has been called on all paths out of the constructor. For this we extend the state type by a component of HOL type *bool*. It is set to *True* when the superclass constructor is called, and checked for in the *Return* instruction rule when we are verifying a constructor. Since most of our typing rules remain the same as without the additional *bool* level, we introduce a new *state-bool*, and *state-type* is now:

$$\begin{array}{rcl} \mathit{state\text{-}type} & = & \mathit{init\text{-}ty\ list} \times \mathit{init\text{-}ty\ err\ list} \\ \mathit{state\text{-}bool} & = & \mathit{state\text{-}type} \times \mathit{bool} \end{array}$$

To deal with unreachable code, the BCV is again lifted to *state-bool option*.

Figure 6 shows a welltyping for the instructions of figure 5 with the new state type. To fit it on the page we have left out the *option* and *bool* component.

pc	instruction	*state-type*
4	. . .	
5	*New A*	([], [*Init* (*Class B*)])
6	*Dup*	([*UnInit A 5*], [*Init* (*Class B*)])
7	*Load 0*	([*UnInit A 5*, *UnInit A 5*], [*Init* (*Class B*)])
8	*Inv−spcl A* (*init*, [*Class B*])	([*Init* (*Class B*), *UnInit A 5*, *UnInit A 5*], [*Init* (*Class B*)])
9	. . .	([*Init* (*Class A*)], [*Init* (*Class B*)])

Fig. 6. Example of a welltyping with object initialization

This section covers the changes to the operational semantics we need to model constructor calls and their correct handling in the BCV.

For one we obviously need constructors. In the real JVM, constructors are methods with a special name `<init>` and no return value. Similarly, in μJava constructors are ordinary methods with a special name *init*, but they may have a return value as any other method (which we have ignored in figure 6).

We also need a bytecode instruction to call constructors: *Invoke-spcl*. Similar to the *Invoke* instruction, it has a class and a method signature as parameters. In contrast to *Invoke* it does not do a virtual method call, but a static one. μJava does not contain static methods apart from constructors; therefore *Invoke-spcl* can only be used for constructor calls.

Since we want to prove something about how far objects are initialized, we need to observe the initialization status of individual objects. The current μJava VM state does not allow this: from heap, stack and local variables we do not get any information how far an object is initialized. Thus we introduce a new, artificial component into the state: a second heap, in structure mirroring the real one, that stores the initialization status using values of HOL type *init-ty*. Formally, the new component *iheap* is simply a function from locations to *init-ty*. At this point the class name parameter of *PartInit* comes into play: if an object gets the tag *PartInit* C it means that the object is initialized up to class C in the class hierarchy, or, more precisely, that the constructor of C's superclass must be called before the fields of the object can be accessed.

This new component of the μJVM state only plays the role of an auxiliary variable for the type safety proof and does not alter the observable behaviour of the former μJava VM.

Unfortunately recording the initialization status of objects induces another problem: in our type safety proof we rely on a large invariant which requires that objects once allocated do not change their type on the heap. Values may change of course, and new objects may be created, but the type information of existing objects (that the invariant relates to what the BCV has predicted) may not change. This also applies to the new *iheap* component. Observing the changes during the lifetime of objects is the very purpose of *iheap*, so we need to accommodate it somehow. We use the solution Freund proposes in [23]: the invariant may not allow type changes, but it does allow to create new objects. So for each constructor call we create a new blank object which is a copy of the object before, only the initialization status tag gets updated. When a constructor is finished, it replaces the uninitialized reference in the calling method by the now initialized own object.

This sounds like a large modification, but on closer inspection there is again no observational difference in executions between creating new objects (and then discarding the superfluous ones) and the standard semantics.

The initialization process is basically a chain of constructor calls along the class hierarchy. It is finished when the constructor of class *Object* has been called. Before that, in between constructor calls, the object is inaccessible. During the whole process all fields retain their default value, and no method other than the constructor can be invoked on it. Once the end of the chain is reached with the constructor of class *Object*, we just work on the last allocated object. Because each constructor replaces the object of the calling constructor, all intermediate objects are discarded.

In order to replace the correct reference when a constructor is finished, we extend our definition of call frames from §4.1 by a pair of references. In that pair we store the reference the current constructor has been called to initialize and the reference to the fully initialized object. Since in the beginning there will be no fully initialized object, we will store *Null* in that case. To avoid annoying type conversions, we work with values *val* instead of references (although we will only be using addresses and *Null*). So *frame* is now:

$$\mathit{frame} = \mathit{opstack} \times \mathit{locvars} \times \mathit{cname} \times \mathit{sig} \times \mathit{nat} \times (\mathit{val} \times \mathit{val})$$

The definition of *frame* applies to all methods, but only constructors will use the additional component, all other methods will simply ignore it. Adding the *iheap* extension we finally get the new state space of the μJava VM:

$$\begin{aligned} \mathit{jvm\text{-}state} &= \mathit{xcpt\ option} \times \mathit{aheap} \times \mathit{iheap} \times \mathit{frame\ list} \\ \mathit{iheap} &= \mathit{loc} \Rightarrow \mathit{init\text{-}ty} \end{aligned}$$

With this in mind we can take a look at how the formal definition of the operational semantics changes. The only interesting instructions are *New*, *Invoke-spcl*, and *Return*. The rules for the other instructions get two new parameters, *ihp* for initialization status and *z* for the reference update in constructors, but they simply pass them on. The rule for *Load* for instance looks like this:

$$\begin{aligned} &\mathit{exec\text{-}instr}\ (\mathit{Load}\ \mathit{idx})\ \mathit{hp}\ \mathit{ihp}\ \mathit{stk}\ \mathit{vars}\ \mathit{Cl}\ \mathit{sig}\ \mathit{pc}\ z\ \mathit{frs} = \\ &\quad (\mathit{None}, \mathit{hp}, \mathit{ihp}, ((\mathit{vars}\ !\ \mathit{idx})\ \#\ \mathit{stk}, \mathit{vars}, \mathit{Cl}, \mathit{sig}, \mathit{pc}{+}1, z)\#\mathit{frs}) \end{aligned}$$

For the *New* instruction we have to record that freshly created objects are completely uninitialized. Apart from that everything remains the same (see also §4.2, p. 17):

```
exec-instr (New C) hp ihp stk vars Cl sig pc z frs =
let (ref,xp') = new-Addr hp;
        hp' = if xp'=None then hp(ref := Some (blank C)) else hp;
       ihp' = if xp'=None then ihp(ref := UnInit C pc) else ihp';
       stk' = if xp'=None then (Addr ref)#stk else stk
in (xp', hp', ihp', (stk', vars, Cl, sig, pc+1, z)#frs)
```

The definition of *Invoke-spcl* is new:

```
exec-instr (Invoke-spcl C mn ps) hp ihp stk vars Cl sig pc z frs =
let          n = size ps;
          args = take n stk;
           ref = stk!n;
            x1 = raise-xcpt (ref=Null) NullPointer;
             D = fst(the(hp the-Addr ref));
  (dc, _, _, mxl, _) = the (method (Γ,C) (mn,ps));
       (a',x2) = new-Addr hp;
           xp' = if x1 = None then x2 else x1;
           hp' = hp(a' := Some (blank D));
             T = if C = Object then Init (Class D) else PartInit C;
            z' = if C = Object then (Addr a', Addr a') else (Addr a', Null);
          frs' = if xp'≠None then [] else
                 [([],(Addr a')#(rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,z')]
in  (xp', hp', ihp(a':= T), frs'@(stk, vars, Cl, sig, pc, z)#frs)
```

The beginning is the same as *Invoke*: in *args* we store the actual parameters of the constructor call. The reference on which to invoke the constructor is the next element on the stack after the parameters. If it is *Null*, a *NullPointer* exception is thrown. Still as in *Invoke* we retrieve the dynamic type D of the object the reference *ref* points to, but this time we do not use it for a dynamic method lookup; we do a static method lookup instead, with the parameters C (the class), *mn* (the method name), and *ps* (the list of parameter types) of the *Invoke-spcl* instruction. Then we create a new object: as in the *New* instruction, we request a free location, handle the *OutOfMemory* exception should one occur, and assign a blank object to the new address with the same dynamic type as the one at *ref* (in effect, we copy the object at *ref* to the new address *Addr a'*).

Now we come to the init status *ihp'*: the new object gets type *PartInit C* since the constructor for class C has just been invoked, and the next constructor to be invoked must be one in the superclass of C. If C has no superclass, i.e. if C = *Object* then we have reached the end of the constructor chain and can view the new object as fully initialized. After this its fields and methods are accessible, first from the constructor body of class *Object*, then in turn from each constructor in the call chain. At this point we also have to store in the frame of the new constructor the pair z' that tells which reference the constructor is supposed to initialize and with which reference to replace *ref* when it is finished. The former is the new object at *Addr a′*. The latter we do not know yet, so we set it to *Null*. Only if we have reached *Object*, we know that our newly created object is also the final one that will replace all intermediate objects in the call chain. In this setting we construct the call frame of the constructor to be invoked just as the normal *Invoke* does.

The *Return* instruction now additionally handles the reference update at constructor returns:

```
$exec\text{-}instr\ Return\ hp\ ihp\ stk_0\ vars\ Cl\ sig_0\ pc\ z_0\ frs$ =
$if\ frs=[]\ then\ (None,\ hp,\ ihp,\ [])\ else$
$let$   $(stk,loc,C,sig,pc,z) = hd\ frs;$
            $val = hd\ stk_0;$
       $(mn,ps) = sig_0;$
          $(a,b) = z_0;$
              $n = size\ ps;$
           $args = take\ n\ stk;$
           $addr = stk!n;$
         $drpstk = drop\ (n+1)\ stk;$
           $stk' = if\ mn=init\ then\ val\#(replace\ addr\ b\ drpstk)\ else\ val\#drpstk;$
           $loc' = if\ mn=init\ then\ replace\ addr\ b\ loc\ else\ loc;$
             $z' = if\ mn=init \wedge z = (addr,Null)\ then\ (addr,b)\ else\ z$
$in\ (None,\ hp,\ ihp,\ (stk',loc',C,sig,pc+1,z')\#tl\ frs))$
```

We will first take another look at the parameters: stk_0, sig_0 and z_0 are the stack, the signature and the reference update pair of the current call frame, i.e. the own method. As in §4.2, we extract the return value *val*, our own method name *mn* and our own list of formal parameter types *ps*. Then we drop the actual parameters from the stack in the caller frame. If we are not in a constructor, i.e. if our own method name *mn* is not *init*, we just put the return value on top of the stack in the caller frame and are done. If we are returning from a constructor, however, we use *replace* to substitute the now initialized object *b* (the second component of our reference update pair) for the address of the original object *addr*. This replacement must occur everywhere on the stack and in the local variables of the caller frame to delete all references to the original object. If the caller frame belongs to a constructor that is initializing the same object as we are, we also have to modify the second component of its reference update pair *z* to the initialized object *b*.

5.3 Bytecode verification

Since we already defined types for object initialization in §5.1 we now only have to construct a semilattice for them and change the transfer function accordingly to instantiate the BCV.

The ordering of the semilattice is canonical: *PartInit* and *UnInit* are only related to themselves, for *Init t* we use the old $\preceq$. Formally:

$$\begin{array}{ccccc} \mathit{Init}\ t_1 & \preceq_i & \mathit{Init}\ t_2 & = & t_1 \preceq t_2 \\ a & \preceq_i & b & = & (a = b) \end{array}$$

The carrier set is constructed easily and the supremum operation again follows the ordering canonically:

$$\begin{array}{lcl} \mathit{init\text{-}tys} & \equiv & \{\mathit{Init}\ x |\ x \in (\mathit{types}\ G)\} \cup \{x \mid \exists\, C\ n.\ x = \mathit{UnInit}\ C\ n\} \cup \\ & & \{x \mid \exists\, C.\ x = \mathit{PartInit}\ C\} \end{array}$$

$$\begin{array}{lll} \mathit{sup}\ (\mathit{Init}\ t_1)\ (\mathit{Init}\ t_2) = & \mathit{case}\ \mathit{JType.sup}\ t_1\ t_2\ \mathit{of} \\ & \mathit{Err} \Rightarrow \mathit{Err} \mid \mathit{OK}\ x \Rightarrow \mathit{OK}\ (\mathit{Init}\ x) \\ \mathit{sup}\ a\ b \qquad\qquad\qquad = & \mathit{if}\ a = b\ \mathit{then}\ \mathit{OK}\ a\ \mathit{else}\ \mathit{Err} \end{array}$$

With this we define $\mathit{Init.esl} \equiv (\mathit{init\text{-}tys},\ \preceq_i,\ \mathit{sup})$ as the err-semilattice for *init-ty* and arrive at:

Lemma 11 *If Γ is wellformed then Init.esl is an err-semilattice.*

If we repeat the construction in §4.3, we get:

stk-esl :: *nat* $\Rightarrow$ *init-ty list esl*
stk-esl mxs $\equiv$ *upto-esl mxs Init.esl*

reg-sl :: *nat* $\Rightarrow$ *init-ty err list sl*
reg-sl mxr $\equiv$ *Listn.sl mxr* (*Err.sl Init.esl*)

sl :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *state-bool option err sl*
sl mxs mxr $\equiv$ *Err.sl*(*Opt.esl*(*Product.esl* (*Product.esl*
(*stk-esl mxs*) (*Err.esl*(*reg-sl mxr*))) (*TrivLat.esl*::*bool esl*)))

where *TrivLat.esl*::*bool* is the trivial err-semilattice (with = as ordering) applied to type *bool*.

Lemma 12 *If Γ is wellformed then sl is also a semilattice.*

The second ingredient to the BCV is the transfer function. In figures 7 and 8 we define *eff'* and *app'* for *init-ty*. They both work on *state-type* (where *ty* is replaced by *init-ty*), and do not involve the *bool* and *option* component yet.

Compared to the original version in figures 4 (p. 24) and 3 (p. 23), both definitions have become a bit larger, but remained the same in structure. With *pc*, the program counter of the current instruction, they recieve one

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{eff}' :: \mathit{instr} \times \mathit{state\text{-}type} \Rightarrow \mathit{state\text{-}type}} \\
\mathit{eff}'\ (\mathit{Load}\ \mathit{idx},\ (\mathit{st},\ \mathit{lt})) & = (\mathit{ok\text{-}val}\ (\mathit{lt}!\mathit{idx})\#\mathit{st},\ \mathit{lt}) \\
\mathit{eff}'\ (\mathit{Store}\ \mathit{idx},\ (t\#\mathit{st},\ \mathit{lt})) & = (\mathit{st},\ \mathit{lt}[\mathit{idx}{:=}\ \mathit{OK}\ t]) \\
\mathit{eff}'\ (\mathit{LitPush}\ v,\ (\mathit{st},\ \mathit{lt})) & = (\mathit{Init}\ (\mathit{the}\ (\mathit{typeof}\ v))\#\mathit{st},\ \mathit{lt}) \\
\mathit{eff}'\ (\mathit{Getfield}\ F\ C,(t\#\mathit{st},\ \mathit{lt})) & = (\mathit{Init}\ (\mathit{snd}\ (\mathit{the}\ (\mathit{field}\ (\Gamma,C)\ F)))\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Putfield}\ F\ C,\ (t_1\#t_2\#\mathit{st},\ \mathit{lt})) & = (\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{New}\ C,\ (\mathit{st},\mathit{lt})) & = (\mathit{UnInit}\ C\ \mathit{pc}\#\mathit{st}, \\
 & \qquad \mathit{replace}\ (\mathit{OK}\ (\mathit{UnInit}\ C\ \mathit{pc}))\ \mathit{Err}\ \mathit{lt}) \\
\mathit{eff}'\ (\mathit{Checkcast}\ C,\ (t\#\mathit{st},\mathit{lt})) & = (\mathit{Init}\ (\mathit{Class}\ C)\ \#\ \mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Dup},\ (t\#\mathit{st},\mathit{lt})) & = (t\#t\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{IAdd},\ t_1\#t_2\#\mathit{st},\mathit{lt})) & = (\mathit{Init}\ (\mathit{PrimT}\ \mathit{Integer})\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Ifcmpeq}\ b,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Goto}\ b,\ s) & = s \\
\mathit{eff}'\ (\mathit{Return},\ s) & = s \\
\mathit{eff}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{let}\quad \mathit{st}' = \mathit{drop}\ (1{+}\mathit{size}\ \mathit{ps})\ \mathit{st}; \\
 & \quad (\_,\mathit{rt},\_,\_,\_) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps})) \\
 & \quad \mathit{in}\ (\mathit{Init}\ \mathit{rt}\#\mathit{st}',\ \mathit{lt}) \\
\mathit{eff}'\ (\mathit{Invoke\text{-}spcl}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{let}\ t \quad = \mathit{st}!\mathit{size}\ \mathit{ps}; \\
 & \qquad i \quad = \mathit{Init}\ (\mathit{theClass}\ t); \\
 & \qquad \mathit{st}'' = \mathit{drop}\ (1{+}\mathit{size}\ \mathit{ps})\ \mathit{st}; \\
 & \qquad \mathit{st}' \ = \mathit{replace}\ t\ i\ \mathit{st}''; \\
 & \qquad \mathit{lt}' \ = \mathit{replace}\ (\mathit{OK}\ t)\ (\mathit{OK}\ i)\ \mathit{lt}; \\
 & \quad (\_,\mathit{rt},\_,\_,\_) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps})) \\
 & \quad \mathit{in}\ (\mathit{Init}\ \mathit{rt}\#\mathit{st}',\ \mathit{lt}')
\end{array}$$

Fig. 7. Effect of instructions on the state type with object initialization

more implicit parameter.

In *eff'* the instructions *Load*, *Store*, *Putfield*, *Ifcmpeq*, *Goto*, *Return*, and *Dup* remain unchanged; the instructions *LitPush*, *Getfield*, *CheckCast*, *IAdd*, and *Invoke* now explicitly yield initialized values. The instructions *Invoke-spcl* and *New* are more interesting.

Invoke-spcl is similar to *Invoke*, but it can only be used on uninitialized references (which is checked in *app*). After the constructor returns, the reference will be fully initialized, so the *Invoke-spcl* rule replaces the uninitialized type t with an initialized one of the same class ($\mathit{theClass} :: \mathit{ini\text{-}ty} \Rightarrow \mathit{ty}$ satisfies $\mathit{theClass}\ (\mathit{PartInit}\ C) = \mathit{Class}\ C$ and $\mathit{theClass}\ (\mathit{UnInit}\ C\ \mathit{pc}) = \mathit{Class}\ C$). The replacement again happens everywhere in the stack and local variables.

The *New* instruction seems easy at first sight: if a *New C* is at position *pc*, it produces the type *UnInit C pc*. The rule in figure 7 does a bit more, though. For the alias analysis to be correct, there must not be any former instances of the type *UnInit C pc* with the same *pc* on the stack or in the local variables. We take care of this with the *replace* for local variables in *eff'* and with

$$
\begin{array}{ll}
\mathit{app}' :: \mathit{instr} \times \mathit{state\text{-}type} \Rightarrow \mathit{bool} & \\
\mathit{app}'\ (\mathit{Load}\ \mathit{idx},\ (\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{lt} \wedge \mathit{lt}!\mathit{idx} \neq \mathit{Err} \wedge \\
 & \quad \mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Store}\ \mathit{idx},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size}\ \mathit{lt} \\
\mathit{app}'\ (\mathit{LitPush}\ v,\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{st} < \mathit{mxs} \wedge \mathit{typeof}\ v \neq \mathit{None} \\
\mathit{app}'\ (\mathit{Getfield}\ F\ C,\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge\ t \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,\ t')) \\
\mathit{app}'\ (\mathit{Putfield}\ F\ C,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,t') \wedge \\
 & \quad t_2 \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge t_1 \preceq_i \mathit{Init}\ t') \\
\mathit{app}'\ (\mathit{New}\ C,\ (\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{size}\ \mathit{st} < \mathit{mxs} \wedge \\
 & \quad \mathit{UnInit}\ C\ \mathit{pc} \notin \mathit{set}\ \mathit{st} \\
\mathit{app}'\ (\mathit{Checkcast}\ C,\ t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge (\exists\, r.\ t = \mathit{Init}\ (\mathit{RefT}\ r)) \\
\mathit{app}'\ (\mathit{Dup},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{1}{+}\mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{IAdd},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \wedge t_1 = \mathit{Init}\ (\mathit{PrimT}\ \mathit{Integer}) \\
\mathit{app}'\ (\mathit{Ifcmpeq}\ b,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \vee (\exists\, r\ r'.\ t_1 = \mathit{Init}\ (\mathit{RefT}\ r) \wedge \\
 & \quad t_2 = \mathit{Init}\ (\mathit{RefT}\ r')) \\
\mathit{app}'\ (\mathit{Goto}\ b,\ s) & = \mathit{True} \\
\mathit{app}'\ (\mathit{Return},\ (t\#\mathit{st},\mathit{lt})) & = t \preceq_i \mathit{Init}\ \mathit{rt} \\
\mathit{app}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st} \wedge \mathit{mn} \neq \mathit{init} \wedge \\
 & \quad \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \wedge \\
 & \quad \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st}); \\
 & \qquad t\ \ = \mathit{st}!\mathit{size}\ \mathit{ps} \\
 & \quad \mathit{in}\ t \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \qquad (\forall\, (a,f) {\in} \mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq_i \mathit{Init}\, f) \\
\mathit{app}'\ (\mathit{Invoke\text{-}spcl}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st} \wedge \mathit{mn} = \mathit{init} \wedge \\
 & (\exists\, r.\ \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) = \mathit{Some}\ (C,r)) \wedge \\
 & \quad \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st}); \\
 & \qquad t\ \ = \mathit{st}!\mathit{size}\ \mathit{ps} \\
 & \quad \mathit{in}\ \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \qquad ((\exists\, \mathit{pc}.\ t = \mathit{UnInit}\ C\ \mathit{pc}) \vee \\
 & \qquad t = \mathit{PartInit}\ C' \wedge (C',C) \in \mathit{subcl}\ \Gamma) \wedge \\
 & \qquad (\forall\, (a,f) {\in} \mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq_i (\mathit{Init}\, f))
\end{array}
$$

Fig. 8. Applicability of instructions with object initialization

checking for *UnInit C pc* on the stack in *app′*. Sun's JVM specification solves the problem by disallowing all backwards jumps as long as there is any *UnInit* type anywhere on the stack or in the local variables. This is correct, but somewhat drastic: although the restriction is not severe for programming in Java, it rejects an unnecessary large number of type safe programs. At the same time it is hard to reason about, because it makes assumptions about the dataflow analysis itself and not only about properties of the resulting welltyping. Freund leaves the local variables untouched, but checks for *UnInit* in *app* (if we translate his notation into our framework). This has the drawback of not being monotone (more specifically the local variable part of *app* would not be monotone).

We use the solution of [10]; it yields a monotone transfer function, admits as many programs and is still safe. In fact, here *replace* is the identity function, because the first path to *New* in the dataflow analysis cannot contain *UnInit C pc* (because only the instruction at *pc* creates the type *UnInit C pc*). A merge with subsequent paths that might contain *UnInit C pc* would already give *Err* without *replace*. This fact is difficult to prove formally, because it again involves reasoning about the dataflow analysis (*paths*); just looking at the welltyping is not enough.

Since *eff′* does not yet include the new *bool* component of *state-bool*, we need one more lifting step. The purpose of the *bool* part in the state type is to mark wether a constructor has already called its superclass constructor. We set it to *True* when we call *Invoke-spcl* for a partly initialized type (*app* checks that it is the superclass constructor), otherwise we leave it untouched:

$\mathit{eff\text{-}bool} :: \mathit{instr} \Rightarrow \mathit{state\text{-}bool} \Rightarrow \mathit{state\text{-}bool}$
$\mathit{eff\text{-}bool}\ i\ ((st,lt),z) \equiv$
$(\mathit{eff}'\ (i,(st,lt)),$
$\mathit{if}\ \exists\, C\ m\ p\ D.\ i = \mathit{Invoke\text{-}spcl}\ C\ m\ p \wedge st!\mathit{size}\ p = \mathit{PartInit}\ D\ \mathit{then}\ \mathit{True}\ \mathit{else}\ z)$

$\mathit{eff} :: \mathit{instr} \Rightarrow \mathit{state\text{-}bool}\ \mathit{option} \Rightarrow \mathit{state\text{-}bool}\ \mathit{option}$
$\mathit{eff}\ i\ pc \equiv \mathit{option\text{-}map}\ (\mathit{eff\text{-}bool}\ i\ pc)$

The definition of *app′* in figure 8 looks intimidating, but compared to figure 3 the changes are again small: *Load*, *Store*, *LitPush*, *Dup* and *Goto* are the same. *Getfield*, *Putfield*, *CheckCast*, *IAdd*, *Ifcmpeq*, and *Return* only restrict their arguments to initialized types. *New* we have already mentioned in the discussion of its *eff′* rule. Normal method invocation is also restricted to initialized objects (for the parameters as well as for the object on which to invoke the method). Additionally, the method must not be a constructor. Finally, there is the new rule for *Invoke-spcl*. It is similar to *Invoke*, but here we make sure that we actually call a constructor ($mn = init$). Since it is supposed to be a static method invocation, the method dictionary *method* must yield an entry telling us that the method is defined in class C (and not in a superclass of it). The rule ensures that the type t of the object on which to invoke the constructor is either completely uninitialized, or partly initialized. If it is uninitialized it must be of type *UnInit C pc* (for some *pc*)—only then are we allowed to invoke the C constructor. If it is partly initialized it must be of type *PartInit* C' (where the implicit parameter C' is the class we are currently verifying). This is because if it is partly initialized, then we are verifying a constructor (only there may partly initialized objects occur). It must be initialized up to exactly C', because for any class D the type *PartInit D* may only occur in D constructors. The only thing a constructor may do with partly initialized objects is invoke the superclass constructor on them—so C must be the direct superclass of C'. The JVM specification also allows to call another constructor of the own class, not only one of the superclass. In practice this is convenient, in our formalization it would just add one more uninteresting case (where $C{=}C'$) to all proofs about *Invoke-spcl*.

Again, we have to lift *app'* to the new *bool* component and then the *option* type. This time we do it in one step: for *None* we again get *True*, for *Some* first *app'* must be satisfied, then we have two additional conditions on the superclass-constructor-has-been-called marker z:

```
app :: instr ⇒ state-bool option ⇒ bool
app i s ≡ case s of None ⇒ True | Some t ⇒
let ((st,lt),z) = t in
 app' (i,(st,lt)) ∧
 mn = init ⟶
  ((i = Return ⟶ z) ∧
   (∀ C m p. i = Invoke-spcl C m p ∧ st!size p = PartInit C' ⟶ ¬z))
```

If we are verifying a constructor then at each *Return* instruction the marker must be *True*, and at each *Invoke-spcl* for partly initialized objects the marker must be *False* (we may call the superclass constructor only once; C' is again the current class).

With case distinction over the instruction set, we get:

Lemma 13 *eff and app are monotone.*

The instantiation of Kildall's algorithm remains the same, it just works with other abstract execution functions *eff* and *app* and a slightly different type system. The only thing we need to adjust is the start value of the iteration.

```
wt-kil ≡
bounded (λn. succs (ins!n) n) (size ins) ∧ 0 < size ins ∧
let t  = OK (if mn = init ∧ C ≠ Object then PartInit C else Init (Class C));
    S₀ = Some (([],t#(map (OK ∘ Init) ps))@(replicate mxl Err)),C=Object);
    φ₀ = (OK S₀)#(replicate (size ins−1) (OK None))
in ∀ n < size ins. (kiljvm φ₀)!n ≠ Err
```

The *this* pointer t (local variable 0) is a bit more complicated than before: if we verify a constructor, and if it is not the constructor of class *Object* then the *this* pointer is only partly initialized yet—the superclass constructor has to be called before it can be used. Otherwise, if it is a normal method or if it is class *Object*, we may assume that the *this* pointer is an initialized object. The only new thing in the rest of the start value is that we may also assume that the parameters only contain initialized values.

Since our type system is a semilattice, the ordering meets the ascending chain condition when Γ is wellformed, *step* built of *eff* and *app* is monotone and preserves the carrier set, we get from Theorem 1:

Lemma 14 *If* Γ *is wellformed and succs is bounded by the number of instructions then kiljvm is a bytecode verifier.*

When we also replace *OK* (*Class C*) in *wt-start* by *t* as in *wt-kil*, we obtain the desired connection between BCV and welltypings:

Theorem 15 *If a program* Γ *is wellformed and wt-kil holds for a method, then the method is type correct w.r.t. wt-method.*

Let us revisit what we needed to do in this section to instantiate the framework for the new type system. The largest piece of work were the new definitions of *eff* and *app*. They are necessary anyway, they not only define the executable BCV the framework provides, but they also describe the welltypings we will need in the type safety proof. The additional work was the semilattice construction, the proof that *eff* and *app* are monotone, and the proof that *eff* and *app* preserve the carrier set. The latter two proofs are case distinctions on the instruction set and are handled very well by Isabelle's automatic tactics (with a few relatively easy lemmas about the new types). The semilattice constructions were straightforward since *Init.esl* is easy and for the rest we could use the theories of §2.

We may summarize that while it is considerable work to extend a large formalization by a new feature, the BCV framework presented here handles these changes very well. Due to its modularity and abstractness the amount of additional work needed for the framework is very small in comparison to a monolithic development.

5.4 Type Safety

In this section we investigate if the BCV with object initialization really does what we expect it to do. The type safety proof is still an invariant proof as in §4.3.4 and the main theorem is still that in welltyped programs execution preserves the invariant. The proof itself, a case distinction over the instruction set, is large, but it is not the hard part. The challenge is finding the right invariant.

The invariant should ensure that runtime types are approximated correctly by the static welltyping. With object initialization we also want the new *iheap* component of the μJava VM state to agree with the initialization status the BCV predicts. This is the property that tells us that objects are initialized correctly.

As it is common in invariant proofs, we need to strengthen these goals for the proof to succeed. In the previous sections we extended our model by three things: the new types *UnInit* and *PartInit*, the *iheap*, and the reference updates at constructor returns (because we create a new object at each constructor call). It is not surprising that the invariant needs to formally describe the purpose of these extensions. Apart from the relation of the *iheap* to the statically predicted initialization status, this is captured by the following two properties:

- The alias analysis that the BCV does on uninitialized values must be correct. We created the type *UnInit C pc* to keep track of a single value: the reference to the object that was freshly created by the instruction *New C* at address *pc*. Also the type *PartInit C* is intended to keep track of a single value: the *this* pointer in constructors. The predicate *consistent-init stk loc s ihp* (see §A.2) states that each type *UnInit C pc* and *PartInit C* in a state type *s* refers to at most one value in stack and local variables.
- The BCV and the operational semantics must agree on the new objects that are created for constructor calls and on the reference update that we do at constructor returns. The former is part of *correct-frame*, the latter is described by *constructor-ok*. In §A.2, we show both predicates in more detail.

The formal definitions of these additional properties are hard to understand, but we do not need to go into further detail at this point: it is not necessary to understand them in order to trust the proof. It is enough, and more important, to understand the definition of type approximation, since this is the original goal we wanted to prove. The invariant implies that it holds. In the following we show how to extend the notion of type approximation such that it covers object initialization.

The basic building block is single value conformance. A value v conforms to a type T, if it has the dynamic type T' and $T' \preceq T$. The function *typeof* :: *aheap* ⇒ *val* ⇒ *ty option* looks up the dynamic type of objects in the heap (for other values, dynamic and static type are always equal):

$$(\text{-} \vdash \text{-} ::\preceq \text{-}) :: \textit{aheap} \Rightarrow \textit{val} \Rightarrow \textit{ty} \Rightarrow \textit{bool}$$
$$\textit{hp} \vdash v ::\preceq T \equiv \exists\, T'.\ \textit{typeof hp } v = \textit{Some } T' \wedge T' \preceq T$$

We extend it to take the new *iheap* and *init-ty* into account by declaring a new judgment $\textit{hp},\textit{ih} \vdash v ::\preceq_i T$:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textit{hp},\textit{ih} \vdash v \mathrel{::\preceq_i} T \equiv} \\
\textit{case } T \textit{ of} & \textit{Init } t & \Rightarrow & \textit{hp} \vdash v{::}\preceq t \wedge \textit{is-init hp ih } v \\
 & \mid \textit{UnInit C pc} & \Rightarrow & \textit{hp} \vdash v{::}\preceq \textit{Class } C \wedge \\
 & & & \textit{typeof hp } v = \textit{Some } C \wedge \textit{tag ih } v = \textit{Some } T \\
 & \mid \textit{PartInit } C & \Rightarrow & \textit{hp} \vdash v{::}\preceq \textit{Class } C \wedge \textit{tag ih } v = \textit{Some } T
\end{array}
$$

For initialized types we just use the existing $::\preceq$ and require with *is-init* that the value is initialized. For *UnInit C pc* and *PartInit C* we require that v approximates the type *Class C*, and that the *iheap* exactly agrees with the predicted type (*tag ih v* returns *Some T* if the value v is an address tagged with type T in *ih*). In the *UnInit* case we can be more precise: since *UnInit C pc* is only used for freshly created objects, we know that the static type is exact. We call a value v initialized (*is-init hp ih v*) if it is not an address, or for addresses that really point to an object, if the *iheap ih* contains a tag *Init t* for it.

$$
\begin{array}{l}
\textit{tag} :: \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{init-ty option} \\
\textit{tag ih } v \equiv \textit{if } \exists\, l.\ v = \textit{Addr } l \textit{ then Some } (\textit{ih } (\textit{the-Addr } v)) \textit{ else None}
\end{array}
$$

$$
\begin{array}{l}
\textit{is-init} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{bool} \\
\textit{is-init hp ih } v \equiv \forall\, \textit{loc}.\ v = \textit{Addr loc} \longrightarrow \textit{hp loc} \neq \textit{None} \longrightarrow (\exists\, t.\ \textit{ih loc} = \textit{Init } t)
\end{array}
$$

Using $::\preceq_i$ we now define what it means for a welltyping of stack and local variables to approximate a concrete stack and set of local variables:

$$
\begin{array}{l}
\textit{approx-val} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{init-ty err} \Rightarrow \textit{bool} \\
\textit{approx-val hp ih } v \textit{ any} \equiv \textit{case any of Err} \Rightarrow \textit{True} \mid \textit{OK } T \Rightarrow \textit{hp},\textit{ih} \vdash v \mathrel{::\preceq_i} T
\end{array}
$$

$$
\begin{array}{l}
\textit{approx-loc} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{locvars} \Rightarrow \textit{ty err list} \Rightarrow \textit{bool} \\
\textit{approx-loc hp ih loc lt} \equiv \textit{list-all2 } (\textit{approx-val hp ih}) \textit{ loc lt}
\end{array}
$$

$$
\begin{array}{l}
\textit{approx-stk} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{opstack} \Rightarrow \textit{ty list} \Rightarrow \textit{bool} \\
\textit{approx-stk hp ih stk st} \equiv \textit{approx-loc hp ih stk } (\textit{map OK st})
\end{array}
$$

In §A.1 the invariant *correct-state* implies that the heap is consistent, i.e. that all objects on the heap only have fields according to their declared type. Now, we also need an *h-init hp ihp* to say that the fields of all objects contain fully initialized values. Both have the same structure: in *h-init* we say that for each object that is defined in the heap *hp* and each field f of these objects, *is-init hp ihp f* must hold. See §A.2 for the formal definition.

These are the properties the invariant must express. The complete invariant is the predicate *correct-state* shown in §A.2. Since it is quite large, complex, and therefore not obvious in its semantics, we need to ask ourselves if that really is what we want prove about the BCV, if it ensures proper object initialization and type safety of bytecode. Fortunately, as already mentioned above, we do not have to understand the whole invariant to trust it. We proved the following lemma.

Lemma 16 *If the invariant holds, dynamic types during execution and the initialization status of all objects are correctly approximated by the welltyping:*

$$\begin{array}{l} \textit{correct-state}\ (\textit{None},\textit{hp},\textit{ihp},(\textit{stk},\textit{loc},C,\textit{sig},\textit{pc},r)\#\textit{frs})\ \Phi \longrightarrow \\ \quad \exists\, st\ lt\ z.\ \Phi\ C\ \textit{sig} = \textit{Some}\ ((st,lt),z)\ \wedge \\ \qquad\qquad \textit{approx-stk}\ \textit{hp}\ \textit{ihp}\ \textit{stk}\ st\ \wedge\ \textit{approx-loc}\ \textit{hp}\ \textit{ihp}\ \textit{loc}\ lt \end{array}$$

The actual type saftey theorem for μJava with object initialization is again:

Theorem 17 *In a welltyped and wellformed program, execution preserves the invariant:*

$$\textit{wt-jvm-prog}\ \Phi\ \wedge\ \textit{correct-state}\ s\ \Phi\ \wedge\ s \xrightarrow{\text{jvm}} t \longrightarrow \textit{correct-state}\ t\ \Phi$$

The Isabelle proof of that theorem is about 3100 lines long (not counting lemmas about the type system) and consists mainly of a large case distinction over the instruction set together with a wealth of lemmas about the invariant. The same proof without object initialization only took about 1000 lines. Of these 1000 lines about 600 could be replayed after slight adjustments. The reusable part consisted of lemmas about the invariant (more specifically about its individual parts) and of the general structure of the type safety proof. The detailed reasoning for individual instructions had to be changed due to the stronger properties to prove. The additional work is mostly due to the new parts of the invariant adding not only in size but also in complexity. Especially the alias analysis predicate *consistent-init* causes an increase in proof size: since it is designed to keep track of single values, each kind of instruction required its own set of lemmas (which was not necessary for the rest of the invariant).

Freund presents a similar proof of type safety in his PhD thesis [23]. While Freund's model as a whole is sound and at least the theorems we have looked at more closely all hold, it still contains some subtle problems (as the non monotonicity of the typing rule for *New*) as well as small errors in the proofs (for instance an incomplete case distinction in lemma D.16.16). We are not trying to find fault in Freund's thesis, however: for a formal development of this size done by hand it is of remarkably high quality. Rather, the point is that exactly in a large and complex formal development there are bound to be human errors, and these are addressed by theorem provers like Isabelle. Isabelle not only does not allow you to make errors in proofs, its automatic tactics also have become powerful enough to help with the technical detail. Another effect of a strict, mechanical formal development is that specifications get validated more thoroughly: a single non monotone rule in the specification is easy to overlook and skip over and thus might not turn up as a problem in hand proofs about the specification. In a theorem prover these kinds of inconsistencies turn up quickly. They also can be fixed more easily and safely because it is easy to see which properties are still valid and which are affected by the change.

6 Conclusion

By verifying an executable BCV, this work closes a significant gap in the effort to provide a machine-checked formalization of the Java/JVM architecture. Despite its relative compactness (the abstract executable BCV is about 2000 lines of specifications, programs, and proofs), the amount of work to construct such a detailed model should not be underestimated. But when it comes to security, there is no substitute for complete formality and tools like Isabelle are very well suited to support it.

We have demonstrated that our abstract formalization of the executable BCV integrates well with our existing work about Java and the JVM. As promised in [21] we included a more realistic feature, object initialization, in our model. We proved that the BCV framework copes well with such significant changes in the type system (μJava was about 9000 lines of Isabelle code before and 11500 lines after adding object initialization).

Including exception handling in the formalization is not difficult; in fact we have already done so [24], but leave it out here for clarity. The complication with exceptions is that two successors of the same instruction can have different result state types in the dataflow analysis (one for normal execution, a different one when an exception occurs). The idea is to change the type of the transfer function such that it yields a list of pairs each consisting of successor instruction and state type.

The JSR instruction is the next challenge on our agenda.

Acknowledgments. We thank Zhenyu Qian, Gilad Bracha, David von Oheimb, Martin Strecker, and Norbert Schirmer for many helpful discussions and for reading drafts of this paper.

A Conformance Relation

A.1 Conformance without object initialization

This section shows the formal definition of the conformance relation between dynamic JVM states and static types in the μJava model without object initialization. It is explained in detail in [2] and [3], so we will be brief here.

A heap conforms if all objects conform. An object conforms if all fields conform to their declared type. The function *fields* takes a program and a class and yields a function that maps field names to their declared type. For the definition of single value conformance $hp \vdash v ::\preceq T$ see §5.4, p. 38.

$$
\begin{aligned}
&\textit{lconf} :: \textit{aheap} \Rightarrow (\alpha \Rightarrow \textit{val option}) \Rightarrow (\alpha \Rightarrow \textit{ty option}) \Rightarrow \textit{bool}\\
&\textit{lconf hp vs Ts} \equiv\\
&\forall n\ T.\ \textit{Ts}\ n = \textit{Some}\ T \longrightarrow (\exists v.\ \textit{vs}\ n = \textit{Some}\ v \wedge hp \vdash v ::\preceq T)
\end{aligned}
$$

$$
\begin{aligned}
&\textit{oconf} :: \textit{aheap} \Rightarrow \textit{obj} \Rightarrow \textit{bool}\\
&\textit{oconf hp}\ (C,\textit{fs}) \equiv \textit{lconf hp fs}\ (\textit{fields}\ (\Gamma,C))
\end{aligned}
$$

$$
\begin{aligned}
&(\text{-}\ \surd) :: \textit{aheap} \Rightarrow \textit{bool}\\
&hp\ \surd \equiv \forall a\ obj.\ hp\ a = \textit{Some obj} \longrightarrow \textit{oconf hp obj}
\end{aligned}
$$

Single value conformance is lifted to a register set and stack by *approx-loc* and *approx-stk*. Any value conforms to the unusable type *Err*.

$$
\begin{aligned}
&\textit{approx-val} :: \textit{aheap} \Rightarrow \textit{val} \Rightarrow \textit{init-ty err} \Rightarrow \textit{bool}\\
&\textit{approx-val hp v any} \equiv \textit{case any of Err} \Rightarrow \textit{True} \mid \textit{OK}\ T \Rightarrow hp \vdash v ::\preceq T
\end{aligned}
$$

$$
\begin{aligned}
&\textit{approx-loc} :: \textit{aheap} \Rightarrow \textit{locvars} \Rightarrow \textit{ty err list} \Rightarrow \textit{bool}\\
&\textit{approx-loc hp loc lt} \equiv \textit{list-all2}\ (\textit{approx-val hp})\ \textit{loc lt}
\end{aligned}
$$

$$
\begin{aligned}
&\textit{approx-stk} :: \textit{aheap} \Rightarrow \textit{opstack} \Rightarrow \textit{ty list} \Rightarrow \textit{bool}\\
&\textit{approx-stk hp stk st} \equiv \textit{approx-loc hp stk}\ (\textit{map OK st})
\end{aligned}
$$

A call frame conforms, if its stack and register set conform, the program counter lies inside the instruction list, and if the register set has space for the *this* pointer, the method parameters, and the local variables.

$$
\begin{aligned}
&\textit{correct-frame} :: \textit{aheap} \Rightarrow \textit{state-type} \Rightarrow \textit{nat} \Rightarrow \textit{instr list} \Rightarrow \textit{frame} \Rightarrow \textit{bool}\\
&\textit{correct-frame hp}\ (st,lt)\ \textit{mxl ins}\ (stk,loc,C,sig,pc) \equiv\\
&\quad \textit{approx-stk hp stk st} \wedge \textit{approx-loc hp loc lt} \wedge\\
&\quad pc < \textit{size ins} \wedge \textit{size loc} = \textit{size}\ (\textit{snd sig}) + mxl + 1
\end{aligned}
$$

The following definition uses *prog-type* $=$ *cname* $\Rightarrow$ *sig* $\Rightarrow$ *method-type* which lifts *method-type* to whole programs. The predicate describes the structure of the call frame stack beneath the topmost frame. The parameters rt_0 and sig_0 are the return type and signature of the topmost frame. A list of call frames conforms if it is empty. If it is not empty, the head frame is investigated more closely: the current state type $\Phi\ C\ sig\ !\ pc$ must denote a reachable instruction; the call frame must belong to a defined method; it must be halted at an *Invoke* instruction which created the call frame above (this is not easily expressed, but we can demand that *mn* and *ps* stem form sig_0, that the return type of a static lookup on C' conforms to the one from the frame above (rt_0), and that the current stack is large enough to hold the actual parameters plus the object on which the method was invoked); finally the current frame and the rest of the call frame stack must conform.

```
correct-frames :: aheap ⇒ prog-type ⇒ ty ⇒ sig ⇒ frame list ⇒ bool
correct-frames hp Φ rt₀ sig₀ [] = True
correct-frames hp Φ rt₀ sig₀ (f#frs) =
let (stk,loc,C,sig,pc) = f; (mn,ps) = sig₀ in
  ∃ st lt rt mxs mxl ins C'.
    Φ C sig ! pc = Some (st,lt) ∧ is-class Γ C ∧
    method (Γ,C) sig = Some(C,rt,mxs,mxl,ins) ∧
    ins!pc = Invoke C' mn ps ∧
    (∃ D' rt' b'. method (Γ,C') sig₀ = Some(D',rt',b') ∧ rt₀ ⪯ rt') ∧
    (∃ as t st'. st = (rev as)@t#st' ∧ size as = size ps ∧
                 correct-frame hp (st,lt) mxl ins f ∧
                 correct-frames hp Φ rt sig frs)
```

The following is the toplevel conformance relation between a state and a program type. The first two cases are trivial, the third case requires a conformant heap ($hp\ \surd$), contains special handling for the topmost call frame and delegates the rest to *correct-frames*. The topmost frame is special because it does not need to be halted at an *Invoke* instruction. The topmost frame must conform and the current state type must denote a reachable instruction. The method lookup provides *correct-frame* and *correct-frames* with the required parameters.

```
correct-state :: jvm-state ⇒ prog-type ⇒ bool
correct-state (Some xp,hp,frs) Φ = True
correct-state (None,hp,[]) Φ = True
correct-state (None,hp,f#fs) Φ =
  let (stk,loc,C,sig,pc) = f in
  ∃ rt mxs mxl ins s.
      method (Γ,C) sig = Some(C,rt,mxs,mxl,ins) ∧ Φ C sig ! pc = Some s ∧
      correct-frame hp s mxl ins f ∧ correct-frames hp Φ rt sig fs ∧
      hp √ ∧ is-class Γ C
```

This section shows the full definition of the conformance relation as far as it not already appeared in §5.4.

We begin with the definition of *h-init*: all objects only have fields with initialized values. The structure is analogous to heap conformance (see §A.1):

$\mathit{l\text{-}init} :: \mathit{aheap} \Rightarrow \mathit{iheap} \Rightarrow (\alpha \Rightarrow \mathit{val\ option}) \Rightarrow (\alpha \Rightarrow \mathit{ty\ option}) \Rightarrow \mathit{bool}$
$\mathit{l\text{-}init\ hp\ ih\ vs\ Ts} \equiv$
$\forall\, n\ T.\ \mathit{Ts}\ n = \mathit{Some}\ T \longrightarrow (\exists\, v.\ \mathit{vs}\ n = \mathit{Some}\ v \wedge \mathit{is\text{-}init\ hp\ ih}\ v)$

$\mathit{o\text{-}init} :: \mathit{aheap} \Rightarrow \mathit{iheap} \Rightarrow \mathit{obj} \Rightarrow \mathit{bool}$
$\mathit{o\text{-}init\ hp\ ih}\ (C,\mathit{fs}) \equiv \mathit{l\text{-}init\ hp\ ih\ fs}\ (\mathit{fields}\ (\Gamma,C))$

$\mathit{h\text{-}init} :: \mathit{aheap} \Rightarrow \mathit{iheap} \Rightarrow \mathit{bool}$
$\mathit{h\text{-}init\ hp\ ih} \equiv \forall\, a\ \mathit{obj}.\ \mathit{hp}\ a = \mathit{Some\ obj} \longrightarrow \mathit{o\text{-}init\ hp\ ih\ obj}$

The next component of the conformance relation regards the alias analysis on uninitialized objects. This part follows closely the description in [23].

The basic idea of the judgment *consistent-init* is: if the static local variables or the stack (on the type level) contain two entries *UnInit C pc* then the corresponding entries in the dynamic local variables and stack (on the value level) must contain the same value. Otherwise the BCV might mistakenly mark uninitialized values as initialized at *Invoke-spcl*. The other direction does not hold necessarily: if two values are equal, they do not need to have the same type (one could be *UnInit C pc* the other *Err*). We also require that all uninitialized values are tagged correctly in the *ihp*.

$\mathit{corr\text{-}loc} :: \mathit{locvars} \Rightarrow \mathit{ty\ err\ list} \Rightarrow \mathit{iheap} \Rightarrow \mathit{val} \Rightarrow \mathit{init\text{-}ty} \Rightarrow \mathit{bool}$
$\mathit{corr\text{-}loc\ loc\ lt\ ihp}\ v\ T \equiv$
$\quad \mathit{list\text{-}all2}\ (\lambda l\ t.\ t = \mathit{OK}\ T \longrightarrow l = v \wedge \mathit{tag\ ihp}\ v = \mathit{Some}\ T)\ \mathit{loc\ lt}$

$\mathit{corr\text{-}stk} :: \mathit{opstack} \Rightarrow \mathit{ty\ list} \Rightarrow \mathit{iheap} \Rightarrow \mathit{val} \Rightarrow \mathit{init\text{-}ty} \Rightarrow \mathit{bool}$
$\mathit{corr\text{-}stk\ stk\ st\ ihp}\ v\ T \equiv \mathit{corr\text{-}loc\ stk}\ (\mathit{map\ OK\ st})\ \mathit{ihp}\ v\ T$

$\mathit{corresponds} :: \mathit{opstack} \Rightarrow \mathit{locvars} \Rightarrow \mathit{state\text{-}type} \Rightarrow \mathit{iheap} \Rightarrow \mathit{val} \Rightarrow \mathit{init\text{-}ty} \Rightarrow \mathit{bool}$
$\mathit{corresponds\ stk\ loc\ s\ ihp}\ v\ T \equiv$
$\quad \mathit{corr\text{-}stk\ stk}\ (\mathit{fst}\ s)\ \mathit{ihp}\ v\ T \wedge \mathit{corr\text{-}loc\ loc}\ (\mathit{snd}\ s)\ \mathit{ihp}\ v\ T$

$\mathit{consistent\text{-}init} :: \mathit{opstack} \Rightarrow \mathit{locvars} \Rightarrow \mathit{state\text{-}type} \Rightarrow \mathit{iheap} \Rightarrow \mathit{bool}$
$\mathit{consistent\text{-}init\ stk\ loc\ s\ ihp} \equiv$
$\quad (\forall\, C\ \mathit{pc}.\ \exists\, v.\ \mathit{corresponds\ stk\ loc\ s\ ihp}\ v\ (\mathit{UnInit}\ C\ \mathit{pc})\ ) \wedge$
$\quad (\forall\, C.\ \exists\, v.\ \mathit{corresponds\ stk\ loc\ s\ ihp}\ v\ (\mathit{PartInit}\ C)\ )$

With these definitions we can define conformance of call frames as follows: stack and register set conform, the alias analysis is correct, the type *PartInit* is only used for the *this* pointer in constructors, the *this* pointer in constructors is tagged correctly, the *pc* is inside the method, and the size of the register set is correct:

$$
\begin{array}{l}
\textit{correct-frame} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{state-type} \Rightarrow \textit{nat} \Rightarrow \textit{instr list} \Rightarrow \textit{frame} \Rightarrow \textit{bool}\\
\textit{correct-frame hp ih } (\textit{st},\textit{lt})\ \textit{mxl ins } (\textit{stk},\textit{loc},C,\textit{sig},\textit{pc},(\textit{this},c)) \equiv\\
\quad \textit{approx-stk hp ih stk st} \ \wedge\ \textit{approx-loc hp ih loc lt} \ \wedge\\
\quad \textit{consistent-init stk loc } (\textit{st},\textit{lt})\ \textit{ih} \ \wedge\\
\quad (\textit{fst sig} = \textit{init} \longrightarrow\\
\qquad \textit{corresponds stk loc } (\textit{st},\textit{lt})\ \textit{ih this } (\textit{PartInit } C) \ \wedge\\
\qquad (\exists\, C'.\ \textit{typeof hp this} = \textit{Some } C' \ \wedge\\
\qquad\qquad \textit{tag ih this} \in \{\textit{Some } (\textit{PartInit } C),\ \textit{Some } (\textit{Init } (\textit{Class } C'))\})) \ \wedge\\
\quad \textit{pc} < \textit{size ins} \ \wedge\ \textit{size loc}{=}\textit{size}(\textit{snd sig}){+}\textit{mxl}{+}1
\end{array}
$$

The predicate *constructor-ok* describes the stored references in constructor call chains. It does not describe the situation completely, though. Only if we take into account that the program is welltyped, i.e. that *app* holds for every instruction, do we get the whole picture. To achieve this, *constructor-ok* relates the following three references:

- The *this* pointer a of the calling constructor (of frame n). That is the reference originally to be initialized. Since at that point the initialization process cannot be complete, it must have the *iheap* tag *UnInit* or *PartInit.*
- The *this* pointer b of the current constructor (of frame $n + 1$). This is the reference to the object that was artificially created for this constructor call. It is one step further in the initialization chain and must therefore be tagged with *PartInit* or *Init* (*Init*, only if we have reached the end of the chain and arrived at *Object*). We require that if the tag is *PartInit* then it must be *PartInit* C where C is the current class, and if it is some *Init* (*Class* D) then D from the *iheap* must be equal to the dynamic type C' from the heap.

  That b is exactly one step further and only initialized if the current class is *Object* can be inferred from *app* and the start value of the BCV.
- The reference c to the fully initialized object. It will be passed up along the initialization chain by the *Return* instructions in constructors (see also the semantics of the *Return* instruction in §5.2). The reference can be *Null* when the superclass constructor has not been called yet. More specifically it will exactly be *Null* as long as the superclass-constructor-has-been-called marker z of the BCV is false. If it is not *Null* then it must point to an initialized object of type C'.

All three references have to agree on the dynamic type C' of the object (since it is the same object copied around). Formally this lengthy text reads as:

$$\textit{constructor-ok} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{cname} \Rightarrow \textit{bool} \Rightarrow \textit{val} \times \textit{val} \Rightarrow \textit{bool}$$

$$\begin{array}{l}
\textit{constructor-ok hp ih a C z } (b,\ c) \equiv \\
\quad \exists\, C'\ D\ pc.\ z = (c{\neq}\textit{Null}) \land \\
\qquad\qquad \textit{typeof hp a} = \textit{Some } C' \land \textit{typeof hp b} = \textit{Some } C' \land \\
\qquad\qquad (c{\neq}\textit{Null} \longrightarrow \textit{typeof hp c} = \textit{Some } C') \land \\
\qquad\qquad \textit{tag ih a} \in \{\textit{Some } (\textit{UnInit } C'\ pc),\ \textit{Some } (\textit{PartInit } D)\} \land \\
\qquad\qquad \textit{tag ih b} \in \{\textit{Some } (\textit{PartInit } C),\ \textit{Some } (\textit{Init } (\textit{Class } C'))\} \land \\
\qquad\qquad (c{\neq}\textit{Null} \longrightarrow \textit{tag ih c} = \textit{Some } (\textit{Init } (\textit{Class } C')))
\end{array}$$

The rest of the invariant proceeds much as in §A.1, we merely have inserted *constructor-ok* and adjusted for *Invoke-special.* The parameters rt_0, sig_0, z_0, r_0 stem from the call frame above the current f:

$$\begin{array}{l}
\textit{correct-frames} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{prog-type} \Rightarrow \textit{ty} \Rightarrow \textit{sig} \Rightarrow \textit{bool} \Rightarrow \\
\qquad\qquad\qquad \textit{val} \times \textit{val} \Rightarrow \textit{frame list} \Rightarrow \textit{bool} \\
\textit{correct-frames hp ih } \Phi\ rt_0\ sig_0\ z_0\ r_0\ [] = \textit{True} \\
\textit{correct-frames hp ih } \Phi\ rt_0\ sig_0\ z_0\ r_0\ (f\#\textit{frs}) = \\
\textit{let } (\textit{stk},\textit{loc},C,\textit{sig},\textit{pc},r) = f;\ (\textit{mn},\textit{ps}) = sig_0 \textit{ in} \\
\quad \exists\, \textit{st lt z rt mxs mxl ins } C'. \\
\qquad \Phi\ C\ \textit{sig} \ !\ \textit{pc} = \textit{Some } ((\textit{st},\textit{lt}),z) \land \textit{is-class } \Gamma\ C \land \\
\qquad \textit{method } (\Gamma,C)\ \textit{sig} = \textit{Some}(C,\textit{rt},\textit{mxs},\textit{mxl},\textit{ins}) \land \\
\qquad \textit{ins}!\textit{pc} \in \{\textit{Invoke } C'\ \textit{mn ps},\ \textit{Invoke-special } C'\ \textit{mn ps}\} \land \\
\qquad (\exists\, D'\ rt'\ b'.\ \textit{method } (\Gamma,C')\ sig_0 = \textit{Some}(D',rt',b') \land rt_0 \preceq rt') \land \\
\qquad (\exists\, \textit{as t st}'.\ \textit{st} = (\textit{rev as})@t\#\textit{st}' \land \textit{size as} = \textit{size ps} \land \\
\qquad\qquad (\textit{mn} = \textit{init} \longrightarrow \textit{constructor-ok hp ih } (\textit{stk}!\textit{size as})\ C'\ z_0\ r_0) \land \\
\qquad\qquad \textit{correct-frame hp ih } (\textit{st},\textit{lt})\ \textit{mxl ins f} \land \\
\qquad\qquad \textit{correct-frames hp ih } \Phi\ \textit{rt sig z r frs})
\end{array}$$

To the toplevel conformance predicate we add the new *h-init*, the rest of the definition is the same as in §A.1:

$$\begin{array}{l}
\textit{correct-state} :: \textit{jvm-state} \Rightarrow \textit{prog-type} \Rightarrow \textit{bool} \\
\textit{correct-state } (\textit{Some xp},\textit{hp},\textit{ihp},\textit{frs})\ \Phi = \textit{True} \\
\textit{correct-state } (\textit{None},\textit{hp},\textit{ihp},[])\ \Phi = \textit{True} \\
\textit{correct-state } (\textit{None},\textit{hp},\textit{ihp},f\#\textit{fs})\ \Phi = \\
\quad \textit{let } (\textit{stk},\textit{loc},C,\textit{sig},\textit{pc},r) = f \textit{ in} \\
\quad \exists\, \textit{rt mxs mxl ins s z}. \\
\qquad \textit{method } (\Gamma,C)\ \textit{sig} = \textit{Some}(C,\textit{rt},\textit{mxs},\textit{mxl},\textit{ins}) \land \Phi\ C\ \textit{sig} \ !\ \textit{pc} = \textit{Some } (s,z) \land \\
\qquad \textit{correct-frame hp ihp s mxl ins f} \land \textit{correct-frames hp ihp } \Phi\ \textit{rt sig z r fs} \land \\
\qquad \Gamma \vdash\!h\ \textit{hp}\surd \land \textit{h-init hp ihp} \land \textit{is-class } \Gamma\ C
\end{array}$$

References

[1] T. Nipkow, D. v. Oheimb, Java$_{\ell ight}$ is type-safe — definitely, in: Proc. 25th ACM Symp. Principles of Programming Languages, 1998, pp. 161–170.

[2] T. Nipkow, D. v. Oheimb, C. Pusch, μJava: Embedding a programming language in a theorem prover, in: F. Bauer, R. Steinbrüggen (Eds.), Foundations of Secure Computation, IOS Press, 2000, pp. 117–144.

[3] C. Pusch, Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL, in: W. Cleaveland (Ed.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Vol. 1579 of Lect. Notes in Comp. Sci., Springer-Verlag, 1999, pp. 89–103.

[4] A. D. Gordon, D. Syme, Typing a multilanguage intermediate code, in: Symposium on Principles of Programming Languages, ACM Press, 2001, pp. 248–260.

[5] R. Stata, M. Abadi, A type system for Java bytecode subroutines, in: Proc. 25th ACM Symp. Principles of Programming Languages, ACM Press, 1998, pp. 149–161.

[6] S. N. Freund, J. C. Mitchell, A type system for object initialization in the Java bytecode language, in: ACM Conf. Object-Oriented Programming: Systems, Languages and Applications, 1998.

[7] S. N. Freund, J. C. Mitchell, A formal framework for the java bytecode language and verifier, in: ACM Conf. Object-Oriented Programming: Systems, Languages and Applications, 1999.

[8] Z. Qian, A formal specification of Java Virtual Machine instructions for objects, methods and subroutines, in: J. Alves-Foss (Ed.), Formal Syntax and Semantics of Java, Vol. 1523 of Lect. Notes in Comp. Sci., Springer-Verlag, 1999, pp. 271–311.

[9] M. Hagiya, A. Tozawa, On a new method for dataflow analysis of Java virtual machine subroutines, in: G. Levi (Ed.), Static Analysis (SAS'98), Vol. 1503 of Lect. Notes in Comp. Sci., Springer-Verlag, 1998, pp. 17–32.

[10] R. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer-Verlag, 2001.

[11] Z. Qian, Standard fixpoint iteration for java bytecode verification, ACM Transactions on Programming Languages and Systems (TOPLAS) 22 (4) (2000) 638–672.

[12] A. Goldberg, A specification of Java loading and bytecode verification, in: Proc. 5th ACM Conf. Computer and Communications Security, 1998.

[13] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996.

[14] A. Coglio, A. Goldberg, Z. Qian, Toward a provably-correct implementation of the JVM bytecode verifier, in: Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2, IEEE Computer Society Press, 2000, pp. 403–410.

[15] L. Casset, J. L. Lanet, A formal specification of the Java bytecode semantics using the B method, in: ECOOP'99 Workshop Formal Techniques for Java Programs, 1999.

[16] Y. Bertot, Formalizing a jvml verifier for initialization in a theorem prover, in: Computer Aided Verification (CAV'2001), Vol. 2102 of LNCS, Springer Verlag, 2001, pp. 14–24.

[17] G. Barthe, G. Dufay, L. Jakubiec, S. M. de Sousa, B. Serpette, A Formal Executable Semantics of the JavaCard Platform, in: D. Sands (Ed.), Proceedings of ESOP'01, Vol. 2028 of Lect. Notes in Comp. Sci., Springer-Verlag, 2001, pp. 302–319.

[18] G. Barthe, G. Dufay, L. Jakubiec, S. M. de Sousa, B. Serpette, A formal correspondence between offensive and defensive JavaCard virtual machines, in: A. Cortesi (Ed.), Proceedings of VMCAI'02, Lect. Notes in Comp. Sci., Springer-Verlag, 2002, to appear.

[19] G. A. Kildall, A unified approach to global program optimization, in: Proc. ACM Symp. Principles of Programming Languages, 1973, pp. 194–206.

[20] S. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[21] T. Nipkow, Verified bytecode verifiers, in: F. Honsell (Ed.), Foundations of Software Science and Computation Structures (FOSSACS 2001), Vol. 2030 of Lect. Notes in Comp. Sci., Springer-Verlag, 2001, pp. 347–363.

[22] S. Berghofer, T. Nipkow, Executing higher order logic, in: P. Callaghan, Z. Luo, J. McKinna, R. Pollack (Eds.), Types for Proofs and Programs (TYPES 2000), Vol. 2277 of Lect. Notes in Comp. Sci., Springer-Verlag, 2002, pp. 24–40.

[23] S. N. Freund, Type systems for object-oriented intermediate languages, Ph.D. thesis, Stanford University (2000).

[24] Verificard project website in Munich,
`http://isabelle.in.tum.de/verificard/`.