# Verified Bytecode Verifiers

Tobias Nipkow

Fakultät für Informatik, Technische Universität München
http://www.in.tum.de/~nipkow/

**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 a model of the JVM.

## 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, in particular 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.
- 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 author's 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 is based firmly on existing formalizations of Java and the JVM in Isabelle/HOL [11–13]. From them it also inherits the absence of subroutines, object initialization, and exception handling. Although our only application is the JVM, our modular framework is in principle applicable to other "typed assembly languages" 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 [16, 4, 5, 14, 7]. 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 [13] for a subset of the system by Qian [14]). Only Qian [15] 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 [6] who rephrases and generalizes the overly concrete description of the BCV given in [9] as an instance of a generic data flow framework. Work towards a verified implementation in the SPECWARE system is sketched by Coglio *et al.* [3]. 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 [2] 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 [4] Bertot [1] has recently used the Coq system to prove the correctness of a bytecode verifier that handles object initialization (but again without classes).

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. In §4, Kildall's algorithm, an iterative data flow analysis, is shown to be a bytecode verifier. Finally, in §5, the results of the previous sections are used to derive a bytecode verifier for a subset of the JVM.

## 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 $\alpha$ *set* and $\alpha$ *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 $\alpha\ ord = \alpha \to \alpha \to bool$ and the notations $x \leq_r y\ =\ r\ x\ y$ and $x <_r y\ =\ (x \leq_r y \wedge x \neq y)$ we say that $r :: \alpha\ ord$ is a **partial order** iff the predicate $\mathsf{order} :: \alpha\ ord \to bool$ holds for $r$:

$$\begin{aligned}\mathsf{order}\ r = {} & (\forall x.\ x \leq_r x) \wedge (\forall xy.\ x \leq_r y \wedge y \leq_r x \longrightarrow x = y) \wedge \\ & (\forall xyz.\ x \leq_r y \wedge y \leq_r z \longrightarrow x \leq_r z)\end{aligned}$$

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

### 2.3 Semilattices

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

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

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

Usually data flow analysis is 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. 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.

### 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:

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

Orderings $r$ on $\alpha$ can be lifted to $\alpha\ err$ by making $\mathsf{Err}$ the top element:

$$\begin{array}{lll} \mathsf{le}\ r\ (\mathsf{OK}\ x)\ (\mathsf{OK}\ y) & = & x \leq_r y \\ \mathsf{le}\ r\ \_\ \mathsf{Err} & = & \mathsf{True} \\ \mathsf{le}\ r\ \mathsf{Err}\ (\mathsf{OK}\ y) & = & \mathsf{False} \end{array}$$

We proved that $\mathsf{le}$ preserves the ascending chain condition.

The following lifting functional is frequently useful:

$$\begin{array}{lll} \mathsf{lift2}\ f\ (\mathsf{OK}\ x)\ (\mathsf{OK}\ y) & = & f\ x\ y \\ \mathsf{lift2}\ f\ \_\ \_ & = & \mathsf{Err} \end{array}$$

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 $\mathsf{Err}$ compactly by a triple of type *esl*:

$$\alpha\ ebinop = \alpha \to \alpha \to \alpha\ err \qquad \alpha\ esl = \alpha\ set \times \alpha\ ord \times \alpha\ ebinop$$

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

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

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

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

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 type *option* and set opt as duals to type *err* and set err,

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

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

$$\begin{array}{llll} \mathsf{le}\ r\ (\mathsf{Some}\ x)\ (\mathsf{Some}\ y) & = & x \leq_r y & \mathsf{sup}\ f\ (\mathsf{Some}\ x)\ (\mathsf{Some}\ y) = \mathsf{Some}(f\ x\ y) \\ \mathsf{le}\ r\ \mathsf{None}\ \_ & = & \mathsf{True} & \mathsf{sup}\ f\ \mathsf{None}\ z = z \\ \mathsf{le}\ r\ (\mathsf{Some}\ x)\ \mathsf{None} & = & \mathsf{False} & \mathsf{sup}\ f\ z\ \mathsf{None} = z \end{array}$$

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

### 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

$$\begin{aligned}&\mathsf{esl} :: \alpha\ esl \to \beta\ esl \to (\alpha \times \beta)\ esl\\&\mathsf{esl}\ (A, r_A, f_A)\ (B, r_B, f_B) = (A \times B, \mathsf{le}\ r_A\ r_B, \mathsf{sup}\ f_A\ f_B)\end{aligned}$$

$$\begin{aligned}&\mathsf{le} :: \alpha\ ord \to \beta\ ord \to (\alpha \times \beta)\ ord\\&\mathsf{le}\ r_A\ r_B = \lambda(a_1, b_1)(a_2, b_2).\ a_1 \leq_{r_A} a_2 \wedge b_1 \leq_{r_B} b_2\end{aligned}$$

$$\begin{aligned}&\mathsf{sup} :: \alpha\ ebinop \to \beta\ ebinop \to (\alpha \times \beta)\, ebinop\\&\mathsf{sup}\ f\ g = \lambda(a_1, b_1)(a_2, b_2).\ \mathsf{Err.sup}\ (\lambda xy.(x, y))\ (a_1 +_f a_2)\ (b_1 +_g b_2)\end{aligned}$$

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 $\mathsf{esl}\ L_1\ L_2$, and that if both $r_A$ and $r_B$ satisfy the ascending chain condition, so does $\mathsf{le}\ r_A\ r_B$.

### 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:

$$\mathsf{list}\ n\ A = \{xs \mid \mathsf{size}\ xs = n \wedge \mathsf{set}\ xs \subseteq A\}$$

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

$$\begin{aligned}&\mathsf{sl} :: nat \to \alpha\ sl \to \alpha\ list\ sl\\&\mathsf{sl}\ n\ (A, r, f) = (\mathsf{list}\ n\ A, \mathsf{le}\ r, \mathsf{map2}\ f)\end{aligned}\qquad\begin{aligned}&\mathsf{le} :: \alpha\ ord \to \alpha\ list\ ord\\&\mathsf{le}\ r = \mathsf{list\_all2}\ (\lambda xy.\ x \leq_r y)\end{aligned}$$

where $\mathsf{map2} :: (\alpha \to \beta \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list$ and $\mathsf{list\_all2} :: (\alpha \to \beta \to bool) \to \alpha\ list \to \beta\ list \to bool$ are the obvious functions. We introduce the notation $xs \leq_{[r]} ys = xs \leq_{(\mathsf{le}\ r)} ys$. We have shown (by induction on $n$) that if $L$ is a semilattice, so is $\mathsf{sl}\ n\ L$, and that if $r$ is a partial order and satisfies the ascending chain condition, so does $\mathsf{le}\ r$.

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:

$$\begin{aligned}&\mathsf{sup} :: (\alpha \to \beta \to \gamma\ err) \to \alpha\ list \to \beta\ list \to \gamma\ list\ err\\&\mathsf{sup}\ f\ xs\ ys = \texttt{if}\ \mathsf{size}\ xs = \mathsf{size}\ ys\ \texttt{then}\ \mathsf{coalesce}(\mathsf{map2}\ f\ xs\ ys)\ \texttt{else}\ \mathsf{Err}\\&\mathsf{coalesce}\ [] = \mathsf{OK}\ []\\&\mathsf{coalesce}\ (e\#es) = \mathsf{Err.sup}\ (\lambda x\ xs.\ x\#xs)\ e\ (\mathsf{coalesce}\ es)\end{aligned}$$

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

$$\begin{aligned}&\mathsf{upto\_esl} :: nat \to \alpha\ esl \to \alpha\ list\ esl\\&\mathsf{upto\_esl}\ n\ (A, r, f) = (\textstyle\bigcup_{i \leq n} \mathsf{list}\ i\ A, \mathsf{le}\ r, \mathsf{sup}\ f)\end{aligned}$$

We have shown that if $L$ is an *err*-semilattice, so is $\mathsf{upto\_esl}\ n\ L$.

## 3 Relating type checking and data flow analysis

The purpose of this section is to set up an abstract framework for relating 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 $\sigma$. Note that $\sigma$ is intended not to represent values but their abstraction, types. For example, in a register machine $\sigma$ would be a list of types, one for each register (roughly speaking). We can now define a **program type**, i.e. the type of a program, simply as a list of state types: each element in the list characterizes the state of the machine before execution of the corresponding instruction.

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, $\sigma$ is a HOL type that represents part of the programming language type system.

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 pretend they are global constants, thus increasing readability.

### 3.1 Welltyped instructions

The type system is characterized by a function and an order with top element:

**wti** :: $\sigma\ list \to nat \to bool$ stands for *w*ell *t*yped *i*nstruction: wti $ss\ p$ is true iff instruction number $p$ is welltyped w.r.t. the program type $ss$. Thus wti characterizes welltypedness of the instructions of a particular program, which remains implicit. When instantiating the framework, wti will be instantiated by the partial application of some type system to a program.

**r** :: $(\sigma \times \sigma)set$, an ordering relation on $\sigma$ representing the subtype relation lifted to program states. Thus our framework can deal with programming languages with subtypes.

$\top$ :: $\sigma$ is the top element w.r.t. r and should be thought of as the inconsistent state, indicating a type error.

The program embodied by wti is welltyped w.r.t. some program type $ss$ iff all instructions are welltyped with non-$\top$ states:

$$\mathsf{welltyping}\ ss = \forall p < \mathsf{size}(ss).\ \mathsf{wti}\ ss\ p \wedge ss!p \neq \top$$

### 3.2 Abstract semantics

The semantics of a program is characterized by the functions

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

**succs** :: $nat \to nat\ list$ computes the possible successor instructions: $\mathsf{succs}\ p = [q_1, \ldots, q_k]$ means that 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** $n$ if for all $p < n$ the elements of $\mathsf{succs}\ p$ are less than $n$, i.e. control never leaves the list of instructions below $n$.

### 3.3 A specification of data flow analysis and bytecode verification

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 $\sigma$ 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 program type $ss$ is **stable at** $p$ and **stable**:

$$\mathsf{stable}\ ss\ p = \forall q \in \mathsf{set}(\mathsf{succs}\ p).\ \mathsf{step}\ p\ (ss!p) \leq_{\mathsf{r}} ss!q$$
$$\mathsf{stables}\ ss = \forall p < \mathsf{size}(ss).\ \mathsf{stable}\ ss\ p$$

We call a function $dfa :: \sigma\ list \to \sigma\ list$ a **data flow analyzer** (w.r.t. $n :: nat$ and $A :: \sigma\ set$) iff for all $ss \in \mathsf{list}\ n\ A$

1. $dfa$ preserves $A$: $dfa\ ss \in \mathsf{list}\ n\ A$,
2. $dfa$ produces stable program types: $\mathsf{stables}\ (dfa\ ss)$
3. $dfa$ is increasing: $ss \leq_{[\mathsf{r}]} dfa\ ss$
4. $dfa$ is bounded by stable program types:

$$\forall ts \in \mathsf{list}\ n\ A.\ ss \leq_{[\mathsf{r}]} ts \wedge \mathsf{stables}\ ts \longrightarrow dfa\ ss \leq_{[\mathsf{r}]} ts$$

In case you are not initimately familiar with data flow analysis: the correctness of this specification is not an issue because it is merely a stepping stone that does not occur in our main result.

We need to introduce the subset $A$ of $\sigma$ to make distinctions beyond HOL's type system: for example, when representing a list of registers, $\sigma$ 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.

A bytecode verifier is defined as a function between program types, where the absence of $\top$ in the result indicates welltypedness. Formally, a function $bcv :: \sigma\ list \to \sigma\ list$ is a **bytecode verifier** (w.r.t. $n :: nat$ and $A :: \sigma\ set$) iff

$$\forall ss \in \mathsf{list}\ n\ A.\ \top \notin \mathsf{set}(bcv\ ss)\ =\ (\exists ts.\ ss \leq_{[\mathsf{r}]} ts \wedge \mathsf{welltyping}\ ts)$$

### 3.4 Relating data flow analysis and bytecode verification

Since the data flow analyzer is specified in terms of stability and the bytecode verifier in terms of welltyping, the two notions need to be related. Naively one may assume that the stable program types are exactly the welltypings. This is almost true, but for lists containing $\top$: they may be stable but will not be welltypings. We say that **wti and stable agree for $\top$-free program types** (w.r.t. $n :: nat$ and $A :: \sigma\ set$) iff for all $ss \in \mathsf{list}\ n\ A$ and $p < n$

$$\top \notin \mathsf{set}(ss) \longrightarrow (\mathsf{wti}\ ss\ p = \mathsf{stable}\ ss\ p)$$

Note that because stable is defined in terms of step, r and succs, this property relates wti to those three constants.

**Theorem 1.** *Let r be a partial order with top element $\top$. If wti and stable agree for $\top$-free program types and $dfa$ is a data flow analyzer then $dfa$ is a bytecode verifier (all w.r.t. some $n$ and $A$).*

This theorem follows easily from the definitions. The hard part is to discharge the premise that $dfa$ is a data flow analyzer, i.e. to provide a correct implementation. This is the topic of §4.

Note that instead of having both wti and step as parameters of the framework and requiring that they agree, one could define wti in terms of stable. We have not done so because much of the work on Java's bytecode verifier is based on type systems rather than flow functions. In particular, in §5.5 we build on the verification by Pusch which is phrased in terms of a type system.

## 4 Kildall's algorithm

This section defines and verifies a functional version of Kildall's algorithm [8, 10], a standard data flow analysis tool. In fact, the description of bytecode verification in the official JVM specification [9, pages 129–130] is essentially Kildall's algorithm. We define the algorithm in the context of semilattice $(\mathsf{A}, \mathsf{f}, \mathsf{r})$ and the functions step and succs. Its core is the iteration on a state type $ss$ and a **worklist** $w :: nat\ set$ that holds the set of all indices of $ss$ whose contents has changed and still needs to be propagated. Once $w$ becomes empty, $ss$ is returned:

$$\begin{array}{rl} \mathsf{iter}(ss, w) = & \texttt{if}\ \ w = \{\}\ \texttt{then}\ ss\ \texttt{else} \\ & \texttt{let}\ \ p = \varepsilon p.\ p \in w;\ \ t = \mathsf{step}\ p\ (ss!p) \\ & \texttt{in}\ \ \mathsf{iter}(\mathsf{propa}\ (\mathsf{succs}\ p)\ t\ ss\ (w - \{p\})) \end{array}$$

The choice of which position to consider next is made by Hilbert's $\varepsilon$-operator: $\varepsilon x.P(x)$ is some arbitrary but fixed $x$ such that $P(x)$ holds; if there is no such $x$, then the value of $\varepsilon x.P(x)$ 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 chose whichever element it wants.

The body of the iteration is hidden in function propa which propagates the new value at $ss!p$ to all its successors, thus reflecting the execution of a single instruction $p$. This means, the result of step $p$ $(ss!p)$ has to be "merged" [9, page 130] with the state $ss!q$ of all successor instructions $q$ of $p$. "Merging" means the computation of the supremum w.r.t. $(\mathsf{A}, \mathsf{f}, \mathsf{r})$. The worklist is updated in case this merging results in a change.

$$
\begin{array}{lll}
\mathsf{propa}\ [] & t\ ss\ w = (ss, w) & \\
\mathsf{propa}\ (q\#qs) & t\ ss\ w = \texttt{let}\ \ u = t +_{\mathsf{f}} ss!q & \\
 & \qquad\qquad w' = \texttt{if}\ \ u = ss!q\ \texttt{then}\ w\ \texttt{else}\ \{q\} \cup w & \\
 & \qquad \texttt{in}\ \ \mathsf{propa}\ qs\ t\ (ss[q := u])\ w' &
\end{array}
$$

Kildall's algorithm is simply a call to iter where the worklist is initialized with the set of indices that need propagating:

$$\mathsf{kildall}\ ss = \mathsf{iter}(ss, \{p.\ p < \mathsf{size}\ ss \wedge \exists q \in \mathsf{set}(\mathsf{succs}\ p).\mathsf{step}\ p\ (ss!p) +_{\mathsf{f}} ss!q \neq ss!q\})$$

Function iter is partial in general. Thus its above definition is illegal in HOL, a logic of total functions. The actual definition of iter in HOL requires that the ordering r of our semilattice satisfies the ascending chain condition. For reasons discussed already towards the end of §3.3 we also require that the functions step and succs do not lead outside the semilattice carrier A:

1. succs is bounded by size $ss$
2. **step preserves A**: $\forall s \in \mathsf{A}, p < n.\ \mathsf{step}\ p\ s \in \mathsf{A}$

Finally, $ss$ and $w$ must also be initialized appropriately: set $ss \subseteq \mathsf{A}$ and $w$ **is bounded by size** $ss$: $\forall p \in w.\ p < \mathsf{size}\ ss$.

### 4.1 Kildall's algorithm is a bytecode verifier

We will now sketch the proof that, under suitable assumptions, Kildall's algorithm is indeed a bytecode verifier. For that purpose we introduce some more terminology. We call step **monotone** (w.r.t. A and $n$) iff for all $s \in \mathsf{A}$, $p < n$, $s \leq_{\mathsf{r}} t$ implies step $p\ s \leq_{\mathsf{r}}$ step $p\ t$. The main theorem states that iter satisfies the properties of a data flow analyzer as defined in §3.3:

**Theorem 2.** *If r satisfies the ascending chain condition, step preserves A and is monotone w.r.t. A and n, succs and w are bounded by n, $ss \in$ list $n$ A, and ss is stable at $p < n$ for all $p \notin w$, then*

$$
\begin{array}{l}
\mathit{iter}(ss, w) \in \mathit{list}\ n\ A\ \wedge\ \mathit{stables}\ (\mathit{iter}(ss, w))\ \wedge\ ss \leq_{[r]} \mathit{iter}(ss, w)\ \wedge \\
\forall ts \in \mathit{list}\ n\ A.\ ss \leq_{[r]} ts \wedge \mathit{stables}\ ts \longrightarrow \mathit{iter}(ss, w) \leq_{[r]} ts
\end{array}
$$

The following two corollaries follow easily from the definition of a data flow analyzer and kildall and from Theorem 1:

**Corollary 3.** *If r satisfies the ascending chain condition, step preserves A and is monotone w.r.t. A and n, and succs is bounded by n, then kildall is a data flow analyzer.*

**Corollary 4.** *If $r$ satisfies the ascending chain condition, $\top$ is its top element, step preserves $A$ and is monotone w.r.t. $A$ and $n$, succs is bounded by $n$, and wti and stable agree for $\top$-free program types, then kildall is a bytecode verifier.*

The proof of Theorem 2 is by induction along the recursion of iter. In an imperative language, iter would be a while-loop and the proof would proceed by the following invariant:

$$ss \in \mathsf{list}\ n\ \mathsf{A}\ \wedge\ (\forall p < n.\ p \notin w \longrightarrow \mathsf{stable}\ ss\ p)\ \wedge\ cs \leq_{[\mathsf{r}]} ss\ \wedge$$
$$\forall ts \in \mathsf{list}\ n\ \mathsf{A}.\ cs \leq_{[\mathsf{r}]} ts \wedge \mathsf{stables}\ ts \longrightarrow ss \leq_{[\mathsf{r}]} ts$$

where $cs$ is a logical variable that refers to the initial value of $ss$. Of course, the proof of Theorem 2 follows exactly the same line. As a first step we show that propa can be split into two independent computations: if $w$ is bounded by size $ss$ then

$$\mathsf{propa}\ qs\ t\ ss\ w = (\mathsf{merges}\ t\ qs\ ss, \{q.\ q \in \mathsf{set}\ qs \wedge t +_{\mathsf{f}} ss!q \neq ss!q\} \cup w)$$

where merges is defined recursively:

$$\begin{array}{ll}\mathsf{merges}\ t\ []\ & ss = ss\\ \mathsf{merges}\ t\ (p\#ps)\ & ss = \mathsf{merges}\ t\ ps\ (ss[p := t +_{\mathsf{f}} ss!p])\end{array}$$

Thus the verification can be phrased in terms of the simpler merges instead of propa. Preservation of the invariant can be shown with a few suitable lemmas about merges. When $w = \{\}$ it is easy to see and prove that the invariant implies the corresponding conditions in Theorem 2.

## 5 Application to JVM

In this section we apply the generic framework to a subset of the JVM. We do not consider the details of how information, e.g. the subclass relationship, is encoded in the compiled class files. Instead, we represent this information abstractly, e.g. as a binary relation between class names. To minimize explicit parameterization, such class file derived information is dealt with via implicit parameters, just as explained in the beginning of §3.

### 5.1 Types

Theory JType describes the types of our JVM. The machine only supports the void type, integers, null references and class types (based on a type *cname* of class names):

$$\textbf{datatype}\ ty = \mathsf{Void} \mid \mathsf{Integer} \mid \mathsf{NullT} \mid \mathsf{Class}\ cname$$

Each class file records the direct superclass. In our formalization this gives rise to an implicit parameter S of HOL type $(cname \times cname)set$, where $(D, C) \in \mathsf{S}$

means that $D$ is a direct subclass of $C$. Subclasses induce a subtype relation:

$$\begin{aligned}
&\mathsf{subtype}\ \tau_1\ \tau_2 &&= (\tau_1 = \tau_2\ \vee\ \tau_1 = \mathsf{NullT} \wedge \mathsf{is_Class}\ \tau_2\ \vee \\
&&&\quad \exists C\ D.\ \tau_1 = \mathsf{Class}\ C \wedge \tau_2 = \mathsf{Class}\ D \wedge (C, D) \in \mathsf{S}^*) \\
&\mathsf{is_Class}\ (\mathsf{Class}\ C) &&= \mathsf{True} \\
&\mathsf{is_Class}\ _ &&= \mathsf{False} \\
&\mathsf{is_ref}\ \tau &&= (\tau = \mathsf{NullT} \vee \mathsf{is_Class}\ \tau)
\end{aligned}$$

Corresponding to it we have a supremum operation on types:

$$\begin{aligned}
&\mathsf{sup}\ \mathsf{NullT}\ &&(\mathsf{Class} D) &&= \mathsf{OK}(\mathsf{Class}\ D) \\
&\mathsf{sup}\ (\mathsf{Class}\ C)\ &&\mathsf{NullT}\ &&= \mathsf{OK}(\mathsf{Class}\ C) \\
&\mathsf{sup}\ (\mathsf{Class}\ C)\ &&(\mathsf{Class}\ D) &&= \mathsf{OK}(\mathsf{Class}(\mathsf{some_lub}\ C\ D)) \\
&\mathsf{sup}\ \tau_1\ &&\tau_2\ &&= \mathtt{if}\ \tau_1 = \tau_2\ \mathtt{then}\ \mathsf{OK}\ \tau_1\ \mathtt{else}\ \mathsf{Err}
\end{aligned}$$

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 $\varepsilon$-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.

The type *cname* is assumed to have a distinguished element Object. Predicate $\mathsf{is\_type} :: \mathit{ty} \to \mathit{bool}$ is true for all basic types and for all class types below Class Object (w.r.t. the given subclass hierarchy S). As abbreviations we introduce $\mathsf{types} = \{\tau \mid \mathsf{is\_type}\ \tau\}$ and $\tau_1 \sqsubseteq \tau_2 = \tau_1 \leq_{\mathsf{subtype}} \tau_2$.

**Theorem 5.** *The triple* $\mathit{esl} = (\mathit{types}, \mathit{subtype}, \mathit{sup})$ *is an err-semilattice provided* S *is* ***univalent*** *(each subclass has at most one direct superclass, i.e.* S *represents a single inheritance hierarchy) and* S *is acyclic.*

Univalence and acyclicity together imply that S is a set of trees, and is_type focusses on the subtree below Object.

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

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

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 [9, p. 127] does permit eager loading.

### 5.2 Instructions

For our subset of the JVM we have selected the following representative instruction set

$$\begin{aligned}
\mathbf{datatype}\ \mathit{instr} = {} & \mathsf{Load}\ \mathit{nat} \mid \mathsf{Store}\ \mathit{nat} \mid \mathsf{AConst_Null} \mid \mathsf{IConst}\ \mathit{int} \mid \mathsf{IAdd} \\
\mid {} & \mathsf{Getfield}\ \mathit{ty}\ \mathit{cname} \mid \mathsf{Putfield}\ \mathit{ty}\ \mathit{cname} \mid \mathsf{New}\ \mathit{cname} \\
\mid {} & \mathsf{Invoke}\ \mathit{cname}\ \mathit{mname}\ \mathit{ty}\ \mathit{ty} \mid \mathsf{CmpEq}\ \mathit{nat} \mid \mathsf{Return}
\end{aligned}$$

where *mname* is the type of method names. The main difference to Sun's JVM is that Load, Store, CmpEq, and Return are polymorphic (the real JVM has one such instruction for each base type), Invoke only supports methods with a single parameter (the first *ty* argument), Getfield and Putfield carry the field type but not its name, and CmpEq uses absolute instead of relative addressing.

The JVM is a stack machine where each activation record consists of a stack for expression evaluation and a list of local variables (which we call **registers**). The abstract semantics, which operates on the type level, records the type of each stack element and each register. At a specific program point, a register may hold either of two incompatible types, e.g. an integer or a reference, depending on the computation path that lead to that point. This facilitates reuse of registers and is modeled by the HOL type *ty err*, where OK $\tau$ represents type $\tau$ itself and Err represents the unusable/inconsistent type. In contrast, the stack should only hold values that can actually be used. Thus the configurations of our abstract JVM are pairs of an expression stack and a list of registers:

$$config = ty\ list \times ty\ err\ list$$

The execution of a single instruction is modeled by function exec :: *instr* $\to$ *config* $\to$ *config err* where the result Err indicates a run time error, either because of stack over or underflow or because of type mismatch. In Sun's JVM the maximal stack size is recorded in the class file as the code attribute `max_stack` for each method. To simplify the presentation, we concentrate on the verification of a single method and introduce two further implicit parameters maxs :: *nat* and rT :: *ty*, the method's maximal stack size and its return type. The definition of exec below is quite straightforward.

Note that access to a register beyond the size of *reg* is not checked during execution because it is prevented easily by a single pass over the instruction sequence (see wfi below). A dynamic check would merely add clutter.

We now start to instantiate the parameters of our abstract framework in §3: $\sigma$ = *config err option* where None indicates a program point that has not been reached yet, Some(OK $c$) is a normal configuration $c$ and Some Err an error. The introduction of the extra *option* layer means that our data flow analyzer will also determine reachability of instructions and enforce type correctness for reachable instructions only. Function step is merely a lifted version of exec

$$\begin{array}{l}\mathsf{step}\ p = \mathsf{option\_map}\ (\mathsf{lift}\ \mathsf{exec}\ (\mathsf{bs}!p))\\ \mathsf{lift}\ f\ e = \mathtt{case}\ e\ \mathtt{of}\ \mathsf{Err} \Rightarrow \mathsf{Err} \mid \mathsf{OK}\ x \Rightarrow f\ x\\ \mathsf{option\_map}\ f\ \mathsf{None} = \mathsf{None}\\ \mathsf{option\_map}\ f\ (\mathsf{Some}\ x) = \mathsf{Some}(f\ x)\end{array}$$

where bs is yet another implicit parameter, the list of instructions of the method under examination. As dictated by the general framework in §3.2, we also need a successor function. Its definition is straightforward:

$$\mathsf{succs}\ p = \mathtt{case}\ \mathsf{bs}!p\ \mathtt{of}\ \mathsf{Return} \Rightarrow [p] \mid \mathsf{CmpEq}\ q \Rightarrow [p{+}1, q] \mid \_ \Rightarrow [p{+}1]$$

```
exec instr (st, reg) = case instr of
 Load n ⇒ if size st < maxs then case reg!n of Err ⇒ Err | OK τ ⇒ OK(τ#st, reg)
          else Err
| Store n ⇒ (case st of [] ⇒ Err | τ#st₁ ⇒ OK(st₁, reg[n := OK τ]))
| AConst_Null ⇒ if size st < maxs then OK(NullT#st, reg) else Err
| IConst i ⇒ if size st < maxs then OK(Integer#st, reg) else Err
| IAdd ⇒ (case st of Integer#Integer#st₂ ⇒ OK(Integer#st₂, reg) | _ ⇒ Err)
| Getfield τ_f C ⇒ (case st of [] ⇒ Err
                    | τ_o#st₁ ⇒ if τ_o ⊑ Class C then OK(τ_f#st₁, reg) else Err)
| Putfield τ_f C ⇒ (case st of τ_v#τ_o#st₂ ⇒ if τ_v ⊑ τ_f ∧ τ_o ⊑ Class C
                                              then OK(st₂, reg) else Err
                    | _ ⇒ Err)
| New C ⇒ if size st < maxs then OK((Class C)#st, reg) else Err
| Invoke C m τ_p τ_r ⇒ (case st of τ_a#τ_o#st₂ ⇒ if τ_o ⊑ Class C ∧ τ_a ⊑ τ_p
                                                 then OK(τ_r#st₂, reg) else Err
                        | _ ⇒ Err)
| CmpEq q ⇒ (case st of τ₁#τ₂#st₂ ⇒ if τ₁ = τ₂ ∨ is_ref τ₁ ∧ is_ref τ₂
                                    then OK(st₂, reg) else Err
             | _ ⇒ Err)
| Return ⇒ (case st of [] ⇒ Err | τ#_ ⇒ if τ ⊑ rT then OK(st, reg) else Err)
```

Modelling the Return instruction by a self loop may seem odd, but it needs a successor instruction because otherwise data flow analysis would never execute it. And a loop is simpler than introducing a fictitious successor.

The main omissions in our model of the JVM are exceptions, object initialization, and `jsr/ret` instructions because our work builds on the type safety proof by Pusch (see §5.5) which does not cover these features either. Including object initialization is straightforward, but exceptions require a modification of our abstract framework: flow functions (step) need to be associated with edges rather than nodes of the program graph because exceptional behaviour can differ from normal behaviour. The difficulty of including `jsr/ret` is unclear.

### 5.3 Type system

We start with those context conditions that can be checked separately for each instruction. This check uses a **method dictionary** of HOL type

$$mdict = cname \to (mname \times ty \to ty\ option)$$

A method dictionary is a functional abstraction of the information about the types of all methods defined in the current collection of class files. It maps a class name $C$, method name $m$, and a type $\tau$ to the result type of the method that is selected when $m$ is invoked with an object of class $C$ and a parameter of type $\tau$ (remember that Invoke only supports single argument methods). In case there is no applicable method, None is returned. The argument type is necessary

because Java allows overloading of method names, which are disambiguated via their argument types. The formal details need not concern us here and can be found elsewhere [11].

The wellformedness of an instruction is checked in the context of two further implicit parameters: a method dictionary md :: *mdict* and a limit maxr :: *nat* for the maximal register index in the current method (the `max_locals` code attribute in Sun's JVM).

$$\begin{array}{ll}
\mathsf{wfi}(\mathsf{Load}\ n) & = n < \mathsf{maxr} \\
\mathsf{wfi}(\mathsf{Store}\ n) & = n < \mathsf{maxr} \\
\mathsf{wfi}(\mathsf{Getfield}\ \tau\ C) & = \mathsf{is_type}\ \tau \wedge \mathsf{is_type}\ (\mathsf{Class}\ C) \\
\mathsf{wfi}(\mathsf{Putfield}\ \tau\ C) & = \mathsf{is_type}\ \tau \wedge \mathsf{is_type}\ (\mathsf{Class}\ C) \\
\mathsf{wfi}(\mathsf{New}\ C) & = \mathsf{is_type}\ (\mathsf{Class}\ C) \\
\mathsf{wfi}(\mathsf{Invoke}\ C\ m\ \tau_p\ \tau_r) & = \mathsf{md}\ C\ (m, \tau_p) = \mathsf{Some}(\tau_r) \\
\mathsf{wfi}(_) & = \mathsf{True}
\end{array}$$

The instruction sequence bs is **wellformed** iff all of its elements are: $\forall instr \in \mathsf{set\ bs}.\ \mathsf{wfi}\ instr$.

The method dictionary also comes with a wellformedness condition reflecting the type soundness requirement that a method redefined in a subclass must have a more specialized result type: md is **wellformed** iff

$$\begin{array}{l}
\mathsf{md}\ C\ mT = \mathsf{Some}\ \tau \longrightarrow \\
\mathsf{is_type}\ \tau \wedge (\forall D.\ (D, C) \in \mathsf{S}^* \longrightarrow (\exists \tau'.\ \mathsf{md}\ D\ mT = \mathsf{Some}\ \tau' \wedge \tau' \sqsubseteq \tau))
\end{array}$$

Note that the official JVM specification does not mention this wellformedness property but imposes a stronger one implicitly: one method can override another only if their result types are identical, a restriction we have relaxed.

Now we come to the core of the type system, namely the semilattice structure on $\sigma$ and the predicate wti. Turning $\sigma$ 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 of registers is fixed:

$$\begin{array}{ll}
\mathsf{stk_esl} :: ty\ list\ esl & \mathsf{reg_sl} :: ty\ err\ list\ sl \\
\mathsf{stk_esl} = \mathsf{upto_esl}\ \mathsf{maxs}\ \mathsf{JType.esl} & \mathsf{reg_sl} = \mathsf{Listn.sl}\ \mathsf{maxr}\ (\mathsf{Err.sl}\ \mathsf{JType.esl})
\end{array}$$

See Theorem 5 for JType.esl. Since any error on the stack must be propagated, the stack and registers are combined in a coalesced product via Product.esl and then embedded into *err* and *option* to form the final semilattice sl :: $\sigma$ *sl*:

$$\mathsf{sl} = \mathsf{Opt.sl}(\mathsf{Err.sl}(\mathsf{Product.esl}\ \mathsf{stk\_esl}\ (\mathsf{Err.esl}\ \mathsf{reg\_sl}))) \tag{1}$$

As a shorthand, the three components of sl are named states (the carrier set), le (the ordering), and sup (the supremum). Furthermore we introduce an infix abbreviation for the ordering le: $s \ll t\ =\ s \leq_{\mathsf{le}} t$

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 $\mathsf{S}$ is univalent and acyclic then $\mathsf{sl}$ is a semilattice.*

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

Below you find the definition of wti, the only remaining parameter of our abstract framework. If you expect type systems to come as a set of inference rules, note that each case in the definition of wti could be turned into an equivalent inference rule for a judgment $ss, p \vdash instr$.

$$
\begin{array}{l}
\mathsf{wti}\ ss\ p = \mathtt{case}\ ss!p\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{True} \mid \mathsf{Some}\ e \Rightarrow \mathtt{case}\ e\ \mathtt{of}\ \mathsf{Err} \Rightarrow \mathsf{False} \\
\mid \mathsf{OK}(st, reg) \Rightarrow \mathtt{case}\ \mathsf{bs}!p\ \mathtt{of} \\
\quad \mathsf{Load}\ n \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \exists \tau.\ reg!n = \mathsf{OK}\ \tau \wedge \mathsf{Some}(\mathsf{OK}(\tau \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Store}\ n \Rightarrow \exists \tau\ st_1.\ st = \tau \# st_1 \wedge \mathsf{Some}(\mathsf{OK}(st_1, reg[n := \mathsf{OK}\ \tau])) \ll ss!(p{+}1) \\
\mid \mathsf{AConst\_Null} \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \mathsf{Some}(\mathsf{OK}(\mathsf{NullT} \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{IConst}\ i \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \mathsf{Some}(\mathsf{OK}(\mathsf{Integer} \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{IAdd} \Rightarrow \exists st_2.\ st = \mathsf{Integer} \# \mathsf{Integer} \# st_2 \wedge \mathsf{Some}(\mathsf{OK}(\mathsf{Integer} \# st_2, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Getfield}\ \tau_f\ C \Rightarrow \exists \tau\ st_1.\ st = \tau \# st_1 \wedge \tau \sqsubseteq \mathsf{Class}\ C \wedge \\
\qquad\qquad \mathsf{Some}(\mathsf{OK}(\tau_f \# st_1, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Putfield}\ \tau_f\ C \Rightarrow \exists \tau_v\ \tau_o\ st_2.\ st = \tau_v \# \tau_o \# st_2 \wedge \tau_v \sqsubseteq \tau_f \wedge \tau_o \sqsubseteq \mathsf{Class}\ C \wedge \\
\qquad\qquad \mathsf{Some}(\mathsf{OK}(st_2, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{New}C \Rightarrow \mathsf{size}\ st < \mathsf{maxs} \wedge \mathsf{Some}(\mathsf{OK}((\mathsf{Class}\ C) \# st, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{Invoke}\ C\ m\ \tau_p\ \tau_r \Rightarrow \exists \tau_a\ \tau_o\ st_2.\ st = \tau_a \# \tau_o \# st_2 \wedge \tau_o \sqsubseteq \mathsf{Class}\ C \wedge \tau_a \sqsubseteq \tau_p \wedge \\
\qquad\qquad \mathsf{Some}(\mathsf{OK}(\tau_r \# st_2, reg)) \ll ss!(p{+}1) \\
\mid \mathsf{CmpEq}\ q \Rightarrow \exists \tau_1\ \tau_2\ st_2.\ st = \tau_1 \# \tau_2 \# st_2 \wedge (\tau_1 = \tau_2 \vee \mathsf{is\_ref}\ \tau_1 \wedge \mathsf{is\_ref}\ \tau_2) \wedge \\
\qquad\qquad \mathsf{Some}(\mathsf{OK}(st_2, reg)) \ll ss!(p{+}1) \wedge \mathsf{Some}(\mathsf{OK}(st_2, reg)) \ll ss!q \\
\mid \mathsf{Return} \Rightarrow \exists \tau\ st_1.\ st = \tau \# st_1 \wedge \tau \sqsubseteq \mathsf{rT}
\end{array}
$$

### 5.4 A verified bytecode verifier

We can now instantiate Kildall's algorithm with its remaining (implicit) parameter, namely the semilattice. Clearly $(\mathsf{A}, \mathsf{f}, \mathsf{r})$ must be sl as defined in (1) and hence $\mathsf{A} = \mathsf{states}$, $\mathsf{f} = \mathsf{sup}$ and $\mathsf{r} = \mathsf{le}$. We call the resulting specialized algorithm $\mathsf{kiljvm} :: \sigma\ list \to \sigma\ list$. Before we can use Corollary 4 to prove that kiljvm is a bytecode verifier, we need three properties of step:

**Lemma 8.** *If bs and md are wellformed, step preserves states and is monotone. If succs is bounded by size bs then wti and stable agree for $\top$-free program types.*

Each property is proved by a case distinction over the different instructions.

**Corollary 9.** *If $\mathsf{S}$ is univalent and $\mathsf{S}^{-1}$ wellfounded (see Lemma 6), bs and md are wellformed, and succs is bounded by size bs, then kiljvm is a bytecode verifier.*

It follows directly from Corollary 4: some of its preconditions are covered by Lemma 8; the ascending chain property of le follows from Lemma 6 because the semilattice constructions involved preserve the property. This corollary clearly shows that the verification of class files can be split up into separate tasks: a first phase where static wellformedness conditions of class files are checked (the class hierarchy, the method result types, and the individual instructions), and the actual data flow analysis. The same separation is also present in the official JVM specification.

### 5.5 The global picture

So far we have only 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 JVM operating on values. To this end we connect our work with that of Pusch [13] who gave a specification of a bytecode verifier and showed that it implies type soundness of the concrete JVM. 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 type *jvmstate* of concrete machine states (which include the pc), a function $\mathsf{exec}_{val}$ :: *jvmstate* $\to$ *jvmstate option* for executing a single instruction, and a conformance relation $\mathsf{corrstate}\ s\ ss$ between a concrete machine state $s$ and a program type $ss$ (all in the context of a set of class files). And she proved

$$\mathsf{welltyping}\ ss \wedge \mathsf{corrstate}\ s\ ss \wedge \mathsf{exec}_{val}\ s = \mathsf{Some}\ t \longrightarrow \mathsf{corrstate}\ t\ ss$$

i.e. a welltyping guarantees type safe execution. By our definition of "bytecode verifier" in terms of welltyping (§3.3) and the fact that kiljvm is a bytecode verifier (Corollary 9) it follows that if $\top \notin \mathsf{kiljvm}\ ss_0$ then execution is type safe from any concrete state $s$ conforming to $ss_0$ (because corrstate is monotone: $\mathsf{corrstate}\ s\ ss_0$ and $ss_0 \leq_{[\mathsf{le}]} ss$ imply $\mathsf{corrstate}\ s\ ss$).

The program type $ss_0$ that iteration starts from is initialized as follows. When checking a method in class $C$ with parameter types $\tau_1, \ldots, \tau_n$ we set

$$\begin{aligned} ss_0 &= [\mathsf{Some}(\mathsf{OK}(init)), \mathsf{None}, \ldots, \mathsf{None}] \\ init &= ([], [\mathsf{Some}(\mathsf{Class}\ C), p_1, \ldots, p_n, \mathsf{None}, \ldots, \mathsf{None}]) \end{aligned}$$

such that $\mathsf{size}\ ss_0 = \mathsf{size}\ \mathsf{bs}$; *init* models the state upon entry into the method: the stack is empty, the first register contains the `this`-pointer, the next $n$ registers contain the parameters $p_i = \mathsf{Some}(\tau_i)$, and the remaining registers, which hold the actual local variables, are uninitialized, i.e. do not contain a usable value.

## 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 (500 lines of specifications and programs, and 2000 lines of 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 since both the theory and the tools are available, we intend to verify implementations of more realistic BCVs (incl. object initialization and exceptions) in the not too distant future.

**Acknowledgments.** I thank Zhenyu Qian, Gilad Bracha, Gerwin Klein and David von Oheimb for many helpful discussions and for reading drafts of this paper.

## References


1. Y. Bertot. A Coq formalization of a type checker for object initialization in the Java Virtula Machine. Technical Report RR-4047, INRIA, Nov. 2000.
2. L. Casset and 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.
3. A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2*, pages 403–410. IEEE Computer Society Press, 2000.
4. S. N. Freund and 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.
5. S. N. Freund and J. C. Mitchell. A formal framework for the java bytecode language and verifier. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1999.
6. A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*, 1998.
7. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In G. Levi, editor, *Static Analysis (SAS'98)*, volume 1503 of *Lect. Notes in Comp. Sci.*, pages 17–32. Springer-Verlag, 1998.
8. G. A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. Principles of Programming Languages*, pages 194–206, 1973.
9. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
10. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
11. T. Nipkow and D. v. Oheimb. $\text{Java}_{light}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170, 1998.
12. T. Nipkow, D. v. Oheimb, and C. Pusch. $\mu$Java: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 117–144. IOS Press, 2000.
13. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 89–103. Springer-Verlag, 1999.
14. Z. Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 271–311. Springer-Verlag, 1999.
15. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Programming Languages and Systems*, ?:?–?, 200? Accepted for publication.
16. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 149–161. ACM Press, 1998.