Verified Bytecode Subroutines

Gerwin Klein and Martin Wildmoser

Technische Universität München
`{kleing|wildmosm}@in.tum.de`

Abstract. Subroutines are a major complication for Java bytecode verification: they are difficult to fit into the data flow analysis that the JVM specification suggests. We examine the problems that occur with subroutines and give an overview of the most prominent solutions in the literature. Using the theorem prover Isabelle/HOL, we have extended our substantial formalization of the JVM and the bytecode verifier with its proof of correctness by the most general solution for bytecode subroutines.

1 Introduction

Bytecode verification is a static check for bytecode safety. Its purpose is to ensure that the JVM only executes safe code: no operand stack over- or underflows, no ill-formed instructions, no type errors. Sun's JVM specification [12] informally describes an algorithm for it: an iterative data flow analysis that statically predicts the types of values on the operand stack and in the register set. Abstractly, the bytecode verifier (BV) is a type inference algorithm.

The relatively simple concept of procedures in the bytecode language does not seem to fit nicely into this data flow analysis. Bytecode subroutines are the center of numerous publications, the cause of bugs in the bytecode verifier, they even have been banished completely from the bytecode language by Sun in the KVM, a JVM for embedded devices.

The contributions of this paper are the following: we advance the field of Java bytecode verifcation with a mechanically verified and executable BV that supports bytecode subroutines; we report on one of the largest applications of the interactive theorem prover Isabelle/HOL [6]; and we make explicit some important assumptions about the type system that remain implicit or are missing completely in pen and paper formalizations.

The formalization we present is the continuation of our work on μJava [8,9], a downsized version of the real Java and JVM. The formalization includes the source language, with operational semantics and a proof of type safety, as well as the bytecode language, with operational semantics, proof of type safety, and executable bytecode verification algorithms. We can only show selected parts of this substantial development here, focusing on the subroutine aspect.

After introducing the JVM (§1.1), bytecode subroutines (§1.2), and the BV with the problems brought forward by subroutines (§1.3), we present the Isabelle/HOL formalization of subroutines in the μJVM (§2), and the bytecode verifier (§3).

1.1 Java Bytecode and the JVM

Sun specifies the JVM in [12] as a stack based interpreter of bytecode methods. It comprises a heap, which stores objects, and a frame stack, which captures local data of currently active methods.

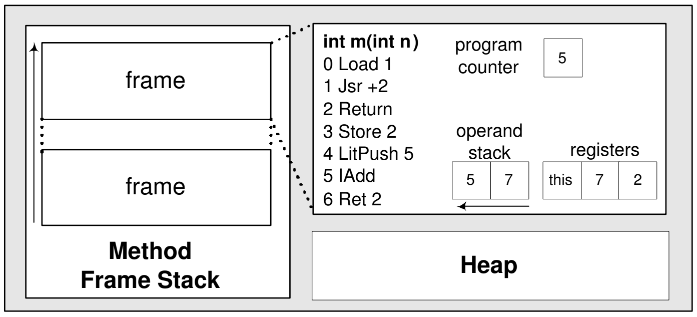

Fig. 1. The JVM.

When the JVM invokes a method it pushes a new *frame* onto the method frame stack. As Fig. 1 indicates, this frame contains the method's program counter, operand stack, and register set. These and the heap are manipulated by the method's bytecode instructions. For example, the *IAdd* instruction removes the topmost two values from the operand stack, adds them, and pushes the result back onto the operand stack. Register *0* is usually reserved for the *this* pointer of the method. The next p registers store the p parameters, and the rest is dedicated to local variables declared inside the method. The heap stores dynamically created objects while the operand stack and registers only contain references into the heap. Each method has an exception handler table, which is a list of tuples (s,e,pc',C). When an exception E occurs, the JVM searches this table for the first entry (s,e,pc',C) where E is a subclass of C and where the program counter is in the protected area $[s,e)$. Finally it enters the handler by setting the program counter to pc'.

1.2 Bytecode Subroutines

Subroutines can be seen as procedures on the bytecode level. If the same sequence of instructions occurs multiple times within a bytecode program, the compiler can place this common code into a subroutine and call it at the desired positions. This is mainly used for the `try`/`finally` construct of Java: the `finally` code must be executed on every possible way out of the block protected by `try`. In contrast to method calls, subroutines share the frame with their caller and manipulate the same register set and stack. Two bytecode instructions, namely *Jsr b* and *Ret x*, handle subroutine calls and returns. The *Jsr b* instruction pushes the return address (the program counter incremented by *1*) onto the stack and branches control to address *pc*+*b*. To return from a subroutine the bytecode language provides the *Ret x* instruction. It jumps to the return address stored in the register with index *x* (*x* is a number). For example, Fig. 2 shows a subroutine with its **entry point** at address *11*, its **call points** at *5* and *8*, and its **return points** at *6* and *9*.

1.3 Bytecode Verification

The purpose of the bytecode verifier is to filter out erroneous and malicious bytecode programs prior to execution. It guarantees that all instructions receive their arguments in correct number, order and type. It also guarantees that the operand stack cannot overflow or underflow and that the *pc* never falls off the code range.

instruction	stack	registers
0 Load 0	([],	[*Int*, *Err*, *Err*])
1 LitPush 0	([*Int*],	[*Int*, *Err*, *Err*])
2 Ifcmpeq +6	([*Int*, *Int*],	[*Int*, *Err*, *Err*])
3 LitPush 25	([],	[*Int*, *Err*, *Err*])
4 Store 1	([*Int*],	[*Int*, *Err*, *Err*])
5 Jsr +6	([],	[*Int*, *Int*, *Err*])
6 Load 1	([],	[*Int*, *Err*, *RA*])
7 Return	([*Err*],	[*Int*, *Err*, *RA*])
8 Jsr +3	([],	[*Int*, *Err*, *Err*])
9 Load 0	([],	[*Int*, *Err*, *RA*])
10 Return	([*Int*],	[*Int*, *Err*, *RA*])
11 Store 2	([*RA*],	[*Int*, $Int \sqcup Err$, *Err*])
12 Ret 2	([],	[*Int*, *Err*, *RA*])

source

```
int m(int n) {
  try {
    if (n==0) {
      return n;
    }
    int j=25;
    return j;
  }
  finally { }
}
```

Fig. 2. Bytecode program with a subroutine.

These properties are checked by an abstract interpretation, which simulates code execution by manipulating types instead of values. This abstraction views a program as a finite state machine working on so called *state types*. A state type characterizes a set of runtime states by giving type constraints for the operand stack and registers. The state type ([*Int*], [*Int*, *Err*, *Err*]) at address *1* in Fig. 2, for example, characterizes all runtime states immediately before execution of the *LitPush 0* instruction: every time execution reaches address *0*, the stack must contain a single integer, register *0* must contain an integer, and the types of the values in registers *1* and *2* are unknown. We call an instruction **applicable** in a state type *s* if it can be executetd safely in all runtime states characterized by *s*. We call a typing of a method a **welltyping** if all instructions are applicable and if the typing is consistent with execution. A state type *s* is consistent in *pc* if the state type at each successor instruction correctly characterizes the runtime state after executing the instruction at *pc* (started in a runtime state characterized by *s*). Bytecode verification is the process of computing welltypings. It is successfull if there is a welltyping for each method in the program. In the example in Fig. 2, bytecode verification was not successfull, because *Return* at *pc=7* is not applicable (execution might be unsafe; the method would return a value of unkown type).

Computing consistent typings is nontrivial because instructions may have multiple predecessors. The state type at *11*, for instance, must be consistent with the execution of both *Jsr +6* at *5* and *Jsr +3* at *8*. The usual solution is to take the least common supertype (componentwise) of the state types after executing the instructions at *5* and *8*. This is also called *merging*. Here, it results in the state type ([*RA*], [*Int*, *Int*$\sqcup$*Err*, *Err*]) at *pc=11*: both instructions put a return address *RA* onto the stack, and both agree on register *0* and *2*. For register *1* the instruction at *5* yields *Int*, while the instruction at *8* yields *Err*. The supremum *Int*$\sqcup$*Err* is *Err*. The type system we use for subroutines proposes a different solution. It recognizes the program in Fig. 2 as safe.

A BV checking code with subroutines faces the following difficulties:

Successor of Ret A BV has to compute the successors of instructions in order to propagate the resulting state types. The successors of *Ret x* instructions are hard to determine statically, because return addresses are values and not accessible on the type level. For example, in Fig. 2 at address 12, the bytecode verifier has to find out that the values of type *RA* stored in register 2 refer to the addresses 6 or 9.

Polymorphism on registers Subroutines may have multiple call points with different types for some registers. We expect that registers not used inside a subroutine have the same type before and after this subroutine's execution. In the example in Fig. 2 at address 11, the BV reacts to the type clash by merging the types *Int* and *Err* to their least common super type. If we merge register types at subroutine entry points, we loose information about their original types. If we propagate the merged type back to the return points, some programs are rejected, because they expect the original, more specific type. For example, bytecode verification fails at *pc*=7 in Fig. 2, because the instruction there expects the original *Int* from address 5 in register 1. This problem mainly occurs with registers that are not used inside the subroutine. We call these subroutines *polymorphic over unused registers*.

Subroutine boundaries Subroutines are not syntactically delimited from the surrounding code. Ordinary jump instructions may be used to terminate a subroutine. Hence it is difficult to determine which instructions belong to a subroutine and which do not.

Subroutine nesting Subroutines can also be nested; a subroutine may call a further subroutine, and so on. This contributes to the difficulty of determining return addresses statically. When we encounter a *Ret x* instruction, we have to find out which of the currently active subroutines is returning. It may be the case that we have a *multilevel return*, which means a subroutine does not return to its caller, but to its caller's caller or further up in the subroutine call history.

The literature offers various solutions to these problems:

Freund [5] labels programs prior to bytecode verification in order to simulate subroutine call stacks statically. Using these labels he specifies typing rules similar to, but more general than those of Stata and Abadi [18].

Leroy [10] proposes a polyvariant analysis which maintains multiple state types per address, ideally one for each control flow route that reaches this address. This avoids type clashes at subroutine entry points.

In Coglio's solution [4], state types are not just single types, but rather whole sets of what in the other approaches was the state type. If a program address i is reachable under two different type configurations (st, lt) and (st', lt'), he assigns the state type $\{(st, lt), (st', lt')\}$, a set, to i rather than the single, merged $(st \sqcup st', lt \sqcup lt')$. Uniting type sets instead of merging types is more precise: as the original type information is not lost, polymorphism on registers is not a problem. This is the approach we also use in the Isabelle formalization below.

In [19], Wildmoser eliminates subroutines by expanding their bodies and proves that this transformation preserves the semantics of the underlying code.

In [15], Posegga and Vogt look at bytecode verification from a model checking perspective. Basin, Friedrich, and Gawkowski [2] use Isabelle/HOL, μJava, and the abstract BV framework [13] to prove this approach correct.

Stärk et al. [17] use Java and the JVM for a case study on abstract state machines. They formalize the process from compilation of Java programs down to bytecode verification.

Barthe et al. [1] employ the Coq system for proofs about the JVM and bytecode verification. They formalize the full JavaCard language, but have only a simplified treatment of subroutines.

2 The μJava VM

This and the following section present our formalization of subroutines for bytecode verification in Isabelle/HOL. We begin with an overview of the structure and the operational semantics of the μJVM. In §3 we then develop the bytecode verifier.

As it is one major point of this article to demonstrate not only how a bytecode verifier with subroutines can be formalized, but how it can be formalized in a theorem prover, we will directly use Isabelle/HOL [14] notation. This coincides mostly with the notation used in mathematics and functional programming. We will show some of the basics now and then introduce new notation as we go along.

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. 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.

List operations may be unfamiliar: the "cons" operator is the infix #, concatenation the infix @; head and tail are *hd* and *tl*. 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 $xs[i := x]$.

We shall now briefly sketch the operational semantics of the JVM. See [7,9] for a more in-depth discussion.

Fig. 3 shows the instruction set. Method bodies are lists of such instructions together with the exception handler table and two integers specifying the maximum operand stack size and the number of local variables.

```
datatype instr =    Load nat          | Store nat
    | LitPush val   | New cname       | Getfield vname cname
    | Ifcmpeq int   | Checkcast cname | Putfield vname cname
    | Return        | Dup             | Invoke cname mname (ty list)
    | IAdd          | Goto int        | Throw
    | Ret nat       | Jsr int
```

Fig. 3. The μJava instruction set.

The state transition relation $s \xrightarrow{\text{jvm}} t$ is the transitive reflexive closure of one-step execution. Execution halts if the frame stack is empty or an unhandled exception has occurred. In all other cases one-step execution is defined by the function *exec-instr*.

The parameters of *exec-instr* are the instruction, the heap, the stack, registers, class, signature, and program counter of the top frame, and the rest of the frame stack; the result is the new state (*None* indicates that no exception occurred). For *Jsr* and *Ret* the definition is:

```
exec-instr (Jsr b) hp stk regs Cl sig pc frs =
   (None, hp, (RetAddr (pc+1)#stk, regs, Cl, sig, nat ((int pc)+b))#frs)

exec-instr (Ret x) hp stk regs Cl sig pc frs =
   (None, hp, (stk, regs, Cl, sig, the-RetAddr (regs ! x)) # frs)
```

The *Jsr* instruction puts the return address *pc+1* on the operand stack and performs a relative jump to the subroutine (*nat* and *int* are Isabelle type conversion functions that convert the HOL type *int* to *nat* and vice versa). The *Ret x* instruction affects only the program counter. It fetches the return address from register x and converts it to *nat* (*the-RetAddr* is defined by *the-RetAddr* (*RetAddr p*) = *p*).

This style of VM is called *aggressive*, because it does not perform any runtime type or safety checks. It just assumes that everything is as expected, e.g. for *Ret x* that in register x there is indeed a return address. It is the task of the bytecode verifier to ensure that these assumptions are met at any time.

For proving type safety it is useful to additionally define a defensive VM that performs safety checks for each instruction. This way it becomes obvious what exactly the bytecode verifier guarantees.

To indicate type errors in the defensive VM, we introduce the datatype

$$\alpha\ \textit{type-error} = \textit{TypeError} \mid \textit{Normal}\ \alpha$$

Similar to the aggressive machine, we build on a function *check-instr* that performs the safety checks for a single execution step. The definitions for *Jsr* and *Ret* do not contain any surprises. In fact, for *Jsr* we only need the branch target to be inside the method:

$\textit{check-instr}\ (\textit{Jsr}\ b)\ \textit{hp stk regs Cl sig pc maxpc frs} =$
$0 \leq \textit{int pc}{+}b \wedge \textit{nat}(\textit{int pc}{+}b) < \textit{maxpc}$

The *Ret x* instruction requires that the index x is inside the register set, that the value of the register is indeed a return address, and that this address is inside the method:

$\textit{check-instr}\ (\textit{Ret}\ x)\ \textit{hp stk regs Cl sig pc maxpc frs} =$
$x < \textit{length regs} \wedge \textit{isRetAddr}\ (\textit{regs}!x) \wedge \textit{the-RetAddr}\ (\textit{regs}!x) < \textit{maxpc}$

One-step execution in the defensive machine directly uses the aggressive machine. The function *check* merely unpacks the state s into the parameter form used by *check-instr*, *exec* does the same for *exec-instr*.

$\textit{exec-d TypeError} = \textit{TypeError}$
$\textit{exec-d}\ (\textit{Normal}\ s) = \textit{if check}\ s\ \textit{then Normal}\ (\textit{exec}\ s)\ \textit{else TypeError}$

The transition relation $\xrightarrow{\text{djvm}}$ is again the reflexive transitive closure of single step execution.

3 The Bytecode Verifier

Our formalization of the BV consists of an abstract framework for dataflow analysis and of a concrete type system that instantiates the framework. In the following, we will concentrate on the type system for subroutines and we will make explicit the requirements the dataflow analysis places on the type system to be admissible for bytecode verification. Abstractly, the parameters of the framework are a *semilattice* describing types and subtyping and a *flow function* describing the effect of instructions on the type level. The result of the framework is a fully executable BV together with a proof of termination and adherence to the type system. We discuss the semilattice with its restrictions in §3.1 and the flow function in §3.2. Details on the framework itself can be found in [9,7].

Since the BV verifies one method at a time, we can view the context of a method and a program as fixed. The context consists of the following values:

Γ :: *program*	the program,
ins :: *instr list*	the instructions of the method,
mxs :: *nat*	maximum stack size of the method,
mxr :: *nat*	size of the register set,
mpc :: *nat*	maximum program counter,
rt :: *ty*	return type of the method,
et :: *ex-table*	exception handler table of the method,
pc :: *nat*	program counter of the current instruction.

The context variables are proper parameters of all functions that use them in the Isabelle formalization. We treat them as global here to spare the reader endless parameter lists in each definition.

3.1 Semilattice

The first parameter of the framework is a semilattice: a tuple $(A, \le_r, \sqcup_f)$ of a carrier set $A :: \alpha\ set$ (the set of types), a partial order $\le_r :: \alpha \Rightarrow \alpha \Rightarrow bool$ (the subtyping relation), and a supremum function $\sqcup_f :: \alpha \Rightarrow \alpha \Rightarrow \alpha$ (calculating least common supertypes). The termination proof of the dataflow analysis requires $\le_r$ to satisfy the *ascending chain condition.* A partial order $\le_r$ satisfies the ascending chain condition on A iff there is no infinitely ascending chain $x_0 <_r x_1 <_r \ldots$ in A ($a <_r b$ is short for $a \le_r b \wedge a \neq b$). In this section we shall build up a semilattice suitable for subroutines.

Following the idea of Coglio [4], we use sets as the elements of the semilattice. The order is the usual subset relation $\subseteq$, and the supremum is union $\cup$. The HOL datatype of basic types in μJava is the following:

$$\begin{array}{lll} \textbf{datatype}\ \textit{prim-ty} & = & \textit{Void} \mid \textit{Boolean} \mid \textit{Integer} \mid \textit{RetA}\ \textit{nat} \\ \textit{ref-ty} & = & \textit{NullT} \mid \textit{ClassT}\ \textit{cname} \\ \textit{ty} & = & \textit{PrimT}\ \textit{prim-ty} \mid \textit{RefT}\ \ \textit{ref-ty} \end{array}$$

The above means that, in μJava, a type *ty* is either a primitive type or a reference type. Primitive types can be the usual *Void*, *Boolean*, and *Integer*, but also a return address *RetA pc*. As we do not need to merge types, we can lift the value *pc* of return addresses into the type system and use it to determine the successors of *Ret* instructions. Reference types are the null type (for the *Null* reference), and class types. For readability, we use the following abbreviations, implemented as *syntax translations* in Isabelle:

$$\begin{array}{lrcllcl} \textbf{translations} & \textit{NT} & \rightleftharpoons & \textit{RefT}\ \textit{NullT} & \textit{Int} & \rightleftharpoons & \textit{PrimT}\ \textit{Integer} \\ & \textit{Class}\ C & \rightleftharpoons & \textit{RefT}\ (\textit{ClassT}\ C) & \textit{Bool} & \rightleftharpoons & \textit{PrimT}\ \textit{Boolean} \\ & \textit{RA}\ pc & \rightleftharpoons & \textit{PrimT}\ (\textit{RetA}\ pc) & & & \end{array}$$

To satisfy the ascending chain condition with $\subseteq$, we need to restrict the datatype *ty* to a finite subset:

$$\textit{types} \equiv \{T \mid \textit{is-type}\ \Gamma\ \ T \wedge (\textit{isRA}\ \ T \longrightarrow \textit{theRA}\ \ T < \textit{mpc})\}$$

The predicate *is-type* T holds if T is declared in Γ, *isRA* does the obvious, and *theRA* (*RA pc*) is *pc*.

It remains to lift this set to the operand stack and register set structure of the BV. The register set is a list of a fixed length *mxr*. Apart from basic types, it may contain unusable values that we denote by an artificial top element *Err*. We write $\textit{types}_\top$ for the extended set of basic types. The operand stack is a list not of fixed, but of maximum, length *mxs*. Using *list n A* for the set of lists over A with length n we arrive at:

$$\textit{state-types} \equiv (\bigcup \{\textit{list n types} |\ n \leq \textit{mxs}\}) \times \textit{list mxr types}_\top$$

The carrier set *states* of the semilattice in the BV is the power set of *state-types* extended by another artificial error element:

$$\textit{states} \equiv (\textit{Pow state-types})_\top$$

The framework [9,7] provides us with $\subseteq_\top$ and $\cup_\top$ that extend $\subseteq$ and $\cup$ canonically by treating *Err* as top element. Using these, we have shown the following lemma.

Lemma 1. ($states$, $\subseteq_\top$, $\cup_\top$) *is a semilattice and* $\subseteq_\top$ *satisfies the ascending chain condition on states.*

It was easy (automatic) to convince Isabelle of the semilattice property. The ascending chain condition for $\subseteq$ follows directly from the fact *states* is finite. Since we know that the state types are finite sets, we can replace them by a list implementation in a real BV. In the ML code generated from the Isabelle specification (using [3]) we have done so; in the formalization we continue with sets.

3.2 Flow Function

The flow function of the framework describes the effect instructions have on the type level. It is intuitive to split the definition of this flow function into two parts: a function $\textit{app} :: \textit{nat} \Rightarrow \textit{state-type} \Rightarrow \textit{bool}$ which checks the applicability of the instruction, and $\textit{eff} :: \textit{nat} \Rightarrow \textit{state-type} \Rightarrow (\textit{nat} \times \textit{state-type})\ \textit{list}$ which carries out the instruction assuming it is applicable. Going back to the example of §1.3 (Fig. 2), *app* would check for position *1* (*LitPush 0*) that *0* is not an address and that there is enough space on the stack to push the result; *eff* would return [(*2*,{([*Int*,*Int*], [*Int*,*Err*,*Err*])})], which says that the effect of *LitPush 0* is pushing one integer onto the stack, and that this result must be propagated to position *2* in the instruction list. Note that there can be a different effect for each successor instruction and that the successors can also depend on the input state type, i.e., the shape of the control flow graph may change during the analysis. Both degrees of freedom are necessary to model the *Ret x* instruction.

For the dataflow analysis to be correct, the framework places the following restrictions on *app* and *eff*:

The semilattice carrier *A* must be closed under *eff* and the effect *eff* must be bounded by *mpc*:

$$\forall s \in \textit{states}.\ \forall p < \textit{mpc}.\ \forall (q,t) \in \textit{set}\ (\textit{eff}\ p\ s).\ q < \textit{mpc} \wedge t \in \textit{states}$$

Applicability must be monotone:

$$\forall s \in \textit{states}.\ \forall t \in \textit{states}.\ \forall p < \textit{mpc}.\ s \subseteq t \longrightarrow (\textit{app}\ p\ t \longrightarrow \textit{app}\ p\ s)$$

The effect must be monotone:

$$\forall s \in \textit{states}.\ \forall t \in \textit{states}.\ \forall p < \textit{mpc}.\ s \subseteq t \wedge \textit{app}\ p\ t \longrightarrow \textit{eff}\ p\ s) \leq_{\{\subseteq\}} \textit{eff}\ p\ t$$

where $A \leq_{\{r\}} B \equiv \forall (p,s) \in \mathit{set}\ A.\ \exists t.\ (p,t) \in \mathit{set}\ B \wedge s \leq_r t$

Note that the monotonicity restriction on *eff* allows the shape of the control flow graph to change in the analysis: if we increase the state type s at a position p, the data flow graph may get more edges (but not less), and the result at each edge may increase (but not decrease).

In the following we instantiate *app* and *eff* for the instruction set of the μJVM. Both definitions are again subdivided into one part for normal and one part for exceptional execution.

We begin with the exception handling part of *app*. It builds on a function *xcpt-names* that determines which of the exceptions that could occur for instruction i have a handler in the method. It returns a list of the exception class names mentioned in those handlers. For *Getfield* for instance it either returns the one element list [*NullPointer*], or the empty list if there is no handler for a *NullPointer* exception. For the *Invoke* instructions all handlers that protect instruction i have to be reported, because an uncaught exception could be propagated up from the invoked method. Applicability for the exception case then only requires that these class names are declared in the program:

$$\mathit{xcpt\text{-}app}\ i \equiv \forall\, C \in \mathit{set}\ (\mathit{xcpt\text{-}names}\ i\ \mathit{pc}\ \mathit{et}).\ \mathit{is\text{-}class}\ \Gamma\ C$$

The definition of the effect in the exception case uses *match-ex-table C pc et* returning *Some handler-pc* if there is an exception handler in the table *et* for an exception of class C thrown at position *pc*, and *None* otherwise. The actual effect is the same for all instructions: the registers *lt* remain the same; the stack is cleared, and a reference to the exception object is pushed. The Isabelle notation f ‘ A is the image of a set A under a function *f*. This effect occurs for every exception class C the instruction may possibly throw.

$$\begin{array}{l}
\mathit{xcpt\text{-}eff} :: \mathit{instr} \Rightarrow \mathit{state\text{-}type} \Rightarrow (\mathit{nat} \times \mathit{state\text{-}type})\ \mathit{list} \\
\mathit{xcpt\text{-}eff}\ i\ s \equiv \mathit{let} \quad t = \lambda C.\ (\lambda(\mathit{st},\mathit{lt}).\ ([\mathit{Class}\ C],\ \mathit{lt}))\ \text{‘}\ s; \\
\qquad\qquad\qquad\qquad \mathit{pc}' = \lambda C.\ \mathit{the}\ (\mathit{match\text{-}ex\text{-}table}\ C\ \mathit{pc}\ \mathit{et}) \\
\qquad\qquad\quad \mathit{in}\ \mathit{map}\ (\lambda C.\ (\mathit{pc}'\ C,\ t\ C))\ (\mathit{xcpt\text{-}names}\ i\ \mathit{pc}\ \mathit{et})
\end{array}$$

This concludes the exception case and we proceed to the applicability of instructions in the normal case. Here, it suffices to look at the elements of the state type separately: *app*$'$, defined in Fig. 4, works on one single stack and register set; *app* then lifts this to sets, i.e. complete state types.

In *app*$'$, a few new functions occur: *typeof* :: *val* $\Rightarrow$ *ty option* returns *None* for addresses, and the type of the value otherwise; *field* looks up declaration information of object fields (defining class and type), while *method* looks up declaration information for methods (here only used to determine if and in which class the method is defined); *take*, and *rev* are the usual functions on lists known from functional programming. The subtype ordering $\preceq$ builds on the direct subclass relation *subcls* Γ induced by the program Γ. It satisfies:

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

$$
\begin{array}{ll}
\mathit{app}' :: \mathit{instr} \times (\mathit{ty\ list} \times \mathit{ty\ err\ list}) \Rightarrow \mathit{bool} & \\
\mathit{app}'\ (\mathit{Load\ idx},\ (\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{lt} \wedge \mathit{lt}!\mathit{idx} \neq \mathit{Err} \wedge \mathit{size\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Store\ idx},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size\ lt} \\
\mathit{app}'\ (\mathit{LitPush}\ v,\ (\mathit{st},\mathit{lt})) & = \mathit{size\ st} < \mathit{mxs} \wedge \mathit{typeof}\ v \in \mathit{Some}`\{\mathit{NT},\ \mathit{Bool},\ \mathit{Int}\} \\
\mathit{app}'\ (\mathit{Getfield}\ F\ C,\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge\ \ t \preceq \mathit{Class}\ C \wedge \\
 & \ (\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 \\
 & \ (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,t') \wedge \\
 & \ t_2 \preceq \mathit{Class}\ C \wedge t_1 \preceq t') \\
\mathit{app}'\ (\mathit{New}\ C,\ (\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{size\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Checkcast}\ C,\ t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{isRefT}\ t \\
\mathit{app}'\ (\mathit{Dup},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{1{+}size\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{IAdd},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \wedge t_1 = \mathit{PrimT\ Integer} \\
\mathit{app}'\ (\mathit{Ifcmpeq}\ b,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (\mathit{isRefT}\ t_1 \wedge \mathit{isRefT}\ t_2) \vee t_1 = t_2 \\
\mathit{app}'\ (\mathit{Goto}\ b,\ s) & = \mathit{True} \\
\mathit{app}'\ (\mathit{Return},\ (t\#\mathit{st},\mathit{lt})) & = t \preceq \mathit{rt} \\
\mathit{app}'\ (\mathit{Throw},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{isRefT}\ t \\
\mathit{app}'\ (\mathit{Jsr}\ b,\ (\mathit{st},\mathit{lt})) & = \mathit{length\ st} < \mathit{maxs} \\
\mathit{app}'\ (\mathit{Ret}\ x,\ (\mathit{st},\mathit{lt})) & = x < \mathit{length\ lt} \wedge (\exists\, r.\ \mathit{lt}!x{=}\mathit{OK}\ (\mathit{RA}\ r)) \\
\mathit{app}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{size\ ps} < \mathit{size\ st} \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \ \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \wedge \\
 & \ \mathit{let\ as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size\ ps})\ \mathit{st});\ t = \mathit{st}!\mathit{size\ ps} \\
 & \ \mathit{in}\ t \preceq \mathit{Class}\ C \wedge \mathit{as}\ [\preceq]\ \mathit{ps} \\
\mathit{app}'\ (i,s) & = \mathit{False}
\end{array}
$$

Fig. 4. Applicability of instructions.

where $(C,D) \in (\mathit{subcls}\ \Gamma)^*$ means that C is a subclass of D. Note that although the subtype relation is no longer used as the semilattice order in the BV, it is still needed to check applicability of instructions.

The definition of app' itself is large, but for most instructions straightforward. Since they are the focus of this paper, we will look at *Jsr* and *Ret* in more detail. The *Jsr b* instruction is easy: it puts the return address on the stack, so we have to make sure that there is enough space for it. The test whether pc' is within the code boundaries is done once for all instructions in *app* below. *Ret x* is equally simple: the index x must be inside the register set, and the value in register x must be a return address.

With app', we can now build the full applicability function *app*: an instruction is applicable when it is applicable in the normal and in the exception case for every element in the state type. To ensure that *eff* is bounded, we also require that all successor program counters are in the method:

$$
\begin{array}{l}
\mathit{app} :: \mathit{nat} \Rightarrow \mathit{state\text{-}type} \Rightarrow \mathit{bool} \\
\mathit{app}\ p\ s \equiv \forall\, t{\in}s.\ \mathit{xcpt\text{-}app}\ (\mathit{ins}!p) \wedge \mathit{app}'\ (\mathit{ins}!p,t) \wedge (\forall\, (q,t){\in}\mathit{set}\ (\mathit{eff}\ p\ s).\ q{<}\mathit{mpc})
\end{array}
$$

This concludes applicability. It remains to build the normal case for *eff* and to combine the two cases into the final effect function. In *eff* we must calculate the successor program counters together with new state types. For the non-exception case, we can define them separately. Fig. 5 shows the successors. Again, most

```
succs :: instr ⇒ nat ⇒ state-type ⇒ nat list
succs (Ifcmpeq b) pc s = [pc+1, nat (int pc + b)]
succs (Goto b) pc s    = [nat (int pc + b)]
succs Return pc s      = []
succs Throw pc s       = []
succs (Jsr b) pc s     = [nat (int pc + b)]
succs (Ret x) pc s     = (SOME l. set l = (the-RA x) ‘ s)
succs i pc s           = [pc+1]
```

Fig. 5. Successor program counters for the non-exception case.

instructions are as expected. *Jsr* is a simple, relative jump, the same as *Goto*. *Ret x* is more interesting. It is the only instruction whose successors depend on the current state type *s*. The function *the-RA x* (*st*,*lt*) extracts the return address from register *x* in *lt*. Since *succs* returns lists and not sets, we use Hilbert's epsilon operator *SOME* to pick any list that converts to this set. The result of *succs* (*Ret 1*) {([],[*Int*,*RA 5*]), ([],(*Int*,*RA 7*))}, for example, is [*5*,*7*]. Remember that in the implementation we plan to use lists for state types instead of sets, so this *SOME* will be just the identity function.

As with *app* we first define an *eff'* on single stack and registers sets (Fig. 6). The *method* expression for *Invoke* determines the return type of the method in question.

```
eff' :: instr ⇒ ty list × ty err list ⇒ ty list × ty err list
eff' (Load idx) (st, lt)         = (ok-val (lt!idx)#st, lt)
eff' (Store idx) (t#st, lt)      = (st, lt[idx:= OK t])
eff' (LitPush v) (st, lt)        = (the (typeof v)#st, lt)
eff' (Getfield F C) (t#st, lt)   = (snd (the (field (Γ,C) F))#st,lt)
eff' (Putfield F C) (st, lt)     = (tl (tl st),lt)
eff' (New C) (st,lt)             = (Class C # st,lt)
eff' (Checkcast C) (t#st,lt)     = (Class C # st,lt)
eff' Dup (t#st,lt)               = (t#t#st,lt)
eff' IAdd (t₁#t₂#st,lt)          = (PrimT Integer#st,lt)
eff' (Ifcmpeq b) (st,lt)         = (tl (tl st),lt)
eff' (Goto b) s                  = s
eff' (Jsr t) (st,lt)             = (RA (pc+1)#st,lt)
eff' (Ret x) s                   = s
eff' (Invoke C mn ps) (st,lt)    = let st' = drop (1+size ps) st;
                                         (_,rt,_,_,_) = the (method (Γ,C) (mn,ps))
                                     in (rt#st', lt)
```

Fig. 6. Effect of instructions on the state type.

While *eff'* saves the *Ret* instruction for later (by just returning *s*), the effect of *Jsr b* is defined there: we put *pc+1* as the return address on top of the stack. Remember that *eff'* is defined in the context we have set up in the beginning of this section, so *pc* is the program counter of the current instruction. If it was not for *Ret*, we could apply *eff'* to every element of the state type and be done. For all other instructions we do just that, for *Ret x* there is special treatment: if we return from a subroutine to a return position *pc'*, only those elements of the state type may be propagated that can return to this position *pc'*—the rest originates from different calls to the subroutine. These are the elements of the state type that contain the return address *pc'* in register *x*. We use *theIdx*, satisfying *theIdx* $(Ret\ x) = x$, to extract the register index from the instruction and *isRet i* to test whether *i* is a *Ret* instruction.

$norm\text{-}eff :: instr \Rightarrow nat \Rightarrow state\text{-}type \Rightarrow state\text{-}type$
$norm\text{-}eff\ i\ pc'\ s \equiv$
$if\ isRet\ i\ then\ \{s' |\ s' \in s \wedge the\text{-}RA\ (theIdx\ i)\ s' = pc'\}\ else\ (eff'\ i)\ `\ s$

For $s = \{([],[Int,RA\ 5]),\ ([],(Int,RA\ 7))\}$, the result of $norm\text{-}eff\ (Ret\ 1)\ 5\ s$ is $\{([],[Int,RA\ 5])\}$ and for $norm\text{-}eff\ (Ret\ 1)\ 7\ s$ it is $\{([],[Int,RA\ 7])\}$.

This is the effect of instructions in the non-exception case. If we apply it to every successor instruction *pc'* returned by *succs* and append the effect for the exception case, we arrive at the final effect function *eff*.

$eff :: nat \Rightarrow state\text{-}type \Rightarrow (nat \times state\text{-}type)\ list$
$eff\ p\ s \equiv (map\ (\lambda pc'.\ (pc',norm\text{-}eff\ i\ pc'\ s))\ (succs\ i\ p\ s))\ @\ (xcpt\text{-}eff\ i\ s)$

If at *p*, *s* has for example the value used above, the result of *eff p s* is $[(5,\ \{([],[Int,RA\ 5])\}),\ (7,\ \{([],(Int,RA\ 7))\})]$.

We have shown the following lemma.

Lemma 2. *The functions app and eff are monotone, eff is bounded by mpc, and states is closed under eff.*

The proof that *eff* is bounded is easy, since *app* explicitly checks this condition. For monotonicity we do not even need to look at single instructions to see that the state type set returned by *eff* cannot decrease when we increase *eff*'s argument, and the number of successors, too, can only increase for larger state types. Preservation of the carrier set is a large case distinction over the instruction set, but Isabelle handles most cases automatically.

3.3 Instantiating the Framework

For any given semilattice and flow function, the framework yields a characterization of welltypings. In our case this is the following.

$$wt\text{-}app\text{-}eff\ \varphi \equiv \forall\, p{<}size\ \varphi.\ app\ p\ (\varphi!p) \wedge (\forall\,(q,t){\in}set(eff\ p\ (\varphi!p)).\ t \subseteq \varphi!q)$$

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.

With Lemmas 1 and 2 the framework also provides an executable function *kildall* :: *state-type list* $\Rightarrow$ *state-type list* (implementing Kildall's algorithm) that is a bytecode verifier in the following sense:

Theorem 1. *The algorithm terminates for any start value* φ_0 *in the carrier set with a result* $\varphi = \mathit{kildall}\ \varphi_0$. *Moreover, if* $\forall\, p < \mathit{mpc}.\ \varphi!p \neq \top$, *then* $\mathit{wt\text{-}app\text{-}eff}\ \ \varphi$ *holds true.*

To turn *kildall* into a type checker that accepts or rejects programs, we need to supply a start state type to the algorithm. The JVM specification tells us what the first state type (at method invocation) looks like: the stack is empty, the first register contains the *this* pointer (of type *Class C*), the next registers contain the parameters of the method, and the rest of the registers is reserved for local variables (which do not have a value yet). In the definition of S_0 we use *ps*, the list of parameter types of the method, and *mxl*, the number of local variables (related to *mxr* by $\mathit{mxr} = \mathit{1}{+}\mathit{size}\ \mathit{ps}{+}\mathit{mxl}$). The state types of the other instructions are initialized with the empty set, the bottom element of the ordering.

$$\begin{aligned} S_0 &= ([],\mathit{Class}\ C\#(\mathit{map}\ (\mathit{OK} \circ \mathit{Init})\ \mathit{ps})@(\mathit{replicate}\ \mathit{mxl}\ \mathit{Err})) \\ \varphi_0 &= (\mathit{OK}\ \{S_0\})\#(\mathit{replicate}\ (\mathit{size}\ \mathit{ins}{-}\mathit{1})\ (\mathit{OK}\ \{\})) \end{aligned}$$

With this, the function *wt-kil* defines the notion of a method being welltyped w.r.t. Kildall's algorithm.

$$\mathit{wt\text{-}kil} \equiv \mathit{0} < \mathit{size}\ \mathit{ins} \wedge (\forall\, n < \mathit{size}\ \mathit{ins}.\ (\mathit{kildall}\ \varphi_0)!n \neq \top)$$

Apart from the call to *kildall*, the function *wt-kil* contains the condition of the JVM specification that the instruction list must not be empty.

3.4 Type Safety

If we write $\mathit{wt\text{-}prog}_k$ for *wt-kil* lifted to programs, the type safety theorem is the following: if the bytecode verifier succeeds and we start the program Γ in its canonical start state, the defensive μJVM will never return a type error.

Theorem 2. *If* C *is a class in* Γ *with a main method, then*

$$\mathit{wt\text{-}prog}_k\ \Gamma \wedge (\mathit{start}\ \Gamma\ C) \xrightarrow{\text{djvm}} \tau \Longrightarrow \tau \neq \mathit{TypeError}$$

The proof of this theorem goes by way of an invariant argument that we do not show here formally. It builds on the idea that the runtime states conform to their predicted type in the sense that each stack and register value is of a subtype of the statically predicted one. In this type system, the statically predicted type is a set of types. Conformance here means that the value conforms to one of the elements of the set. The proof of the invariant lemma is then by induction over the length of the execution, and by case distinction over the instruction set. For each instruction, we pick an element s of the static type set, and we conclude from the conformance of the dynamic state together with with the *app* part of the BV that all assumptions of the operational semantics are met (e.g. non-empty stack). Then we execute the instruction and observe that the new state conforms to $t = \mathit{eff}\ pc\ s$. This t is the element of the type set that we need to show conformance of the new state.

With the additional facts that the start state conforms to Φ if the program has a main method (otherwise the start state is not defined) and that the defensive machine does not produce a type error in conformant states, we can conclude Theorem 2: there will be no type errors in welltyped programs.

4 Conclusion

We have instantiated our previous formalization of an abstract verified data flow analysis with a type system that supports classes, subroutines, and exception handling. The bytecode verifier we have specified is fully executable (as a standalone program) and we have proved in Isabelle/HOL that it is correct.

Our formalization of μJava consists of about 17,000 lines of Isabelle code. This includes all specifications and proofs we mentioned in this article and additionally the source language, a lightweight bytecode verifier, object initialization, and arrays which we have not shown here.

The type system we use is based on [4]; our formalization is more than a version of [4] in Isabelle/HOL, though: we have shown that the idea scales up to a realistic model of the JVM ([4] did not even have classes). Moreover, using Isabelle has forced us to make explicit the conditions under which type systems are admissible for the data flow analysis of the BV: a generalized notion of monotonicity and the ascending chain condition. These concepts are missing in pen and paper formalizations, or they remain at least implicit.

In theory, the sets we use as state types in the data flow analysis could become very large (up to the full set of all possible types). In practice, this is not the case. Even for contrived examples in our tests most sets were singletons; the maximum size was 4. Leroy [11] proposes an optimization of the type system (using widening steps) that effectively reduces all sets to singletons, and the type system to standard bytecode verification, when no subroutines are present. Our formalization can serve as a basis for the correctness of this optimization. There exists an industrial implementation (by Trusted Logic S.A., France) of this type system, time and space efficient, for use on embedded devices.

The type system presented here is directly applicable to lightweight bytecode verification [8,16], eliminating the need to expand subroutines prior to verification on embedded devices; [7] contains more details on this aspect.

After about 5 years of research (starting with [18]), we can finally conclude that, for bytecode verification, subroutines are not a problem anymore.

Acknowledgments We thank Norbert Schirmer and Martin Strecker for commenting on and improving drafts of this paper.

References

1. G. Barthe, G. Dufay, L. Jakubiec, S. M. de Sousa, and B. Serpette. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proceedings of VMCAI'02*, Lect. Notes in Comp. Sci. Springer, 2002. to appear.
2. D. Basin, S. Friedrich, and M. Gawkowski. Verified bytecode model checkers. In *Theorem Proving in Higher Order Logics (TPHOLs'02)*, volume 2410 of *Lect. Notes in Comp. Sci.*, pages 47–66, Virginia, USA, August 2002. Springer.
3. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lect. Notes in Comp. Sci.*, pages 24–40. Springer, 2002.
4. A. Coglio. Simple verification technique for complex Java bytecode subroutines. In *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002.
5. S. N. Freund. *Type Systems for Object-Oriented Intermediate Languages*. PhD thesis, Stanford University, 2000.
6. Isabelle home page, 2002. `http://isabelle.in.tum.de/`.
7. G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.
8. G. Klein and T. Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue on Formal Techniques for Java.
9. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. to appear.
10. X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer, 2001.
11. X. Leroy. Java bytecode verification: algorithms and formalizations. *J. Automated Reasoning*, 2004. to appear.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
13. T. Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lect. Notes in Comp. Sci.*, pages 347–363. Springer, 2001.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer, 2002.
15. J. Posegga and H. Vogt. Java bytecode verification using model checking. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.
16. E. Rose and K. Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.
17. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer, 2001.
18. 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.
19. M. Wildmoser. Subroutines and java bytecode verification. Master's thesis, Technische Universität München, 2002.