Verified Bytecode Subroutines

G. Klein and M. Wildmoser (`(kleing|wildmosm)@in.tum.de`)
Technische Universität München

Abstract. Bytecode subroutines are a major complication for Java bytecode verification: they are difficult to fit into the data flow analysis that the JVM specification suggests. Because of that, subroutines are left out or are restricted in most formalizations of the bytecode verifier. 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.

Keywords: Java, Bytecode Verification, Subroutines, Theorem Proving, Data Flow Analysis, Isabelle

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 in operands of bytecode instructions. Sun's JVM specification [26] 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 (BCV) 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 [42], a JVM for embedded devices.

Publications about subroutines range from describing them as a pain that is best gotten rid of [12] to complex proposed solutions of the problem [13]. Many formalizations of the JVM ignore the *Jsr*/*Ret* instructions completely [14, 5, 34], offer only restricted versions of it [13, 40, 24, 2], or do not take exception handling and object initialization into account [41, 9]. We give a survey of the most promising solutions in §3. A much more thorough version of it, backed by proofs and examples, can be found in [44].

The formalization presented here is the continuation of our work on μJava in the interactive theorem prover Isabelle/HOL [19]. The μJava formalization [22, 23, 29, 30, 34] is a down-sized version of the real Java

language and JVM. It does not contain threads, interfaces, packages, or visibility modifiers like `public` or `private`. It does contain a small but representative instruction set, classes, inheritance, and object oriented method invocation. The formalization includes the source language, its operational semantics and type system together with a proof of type safety, and also the bytecode language together with its operational semantics, type system, proof of type safety, an executable, abstract, and verified bytecode verifier, and an executable verified lightweight bytecode verifier.

We can only present selected parts of this substantial development here: the μJVM, an instantiation of the abstract bytecode verifier with a type system that allows for subroutines, object initialization, and exception handling, and the proof of the BCV's correctness. We will focus on the subroutine aspect of the type system and the μJVM. Object initialization and exception handling are part of the presentation, because these features have proven to interact badly with subroutines in other formalizations [13, 40]. Object initialization is in itself a complex feature of the BCV. We will give the necessary definitions here, but we will not discuss the details and reasoning behind the formalization; [23] describes them in depth.

The remainder of this article is structured as follows: we will first give an intuitive introduction to the notions of the JVM (§1.1), bytecode subroutines (§1.2), and the bytecode verifier (§1.3), before we describe the problems that occur with subroutines in more detail in §2. In §3 we survey the most promising solutions to these problems in the literature. The remaining sections concern the Isabelle/HOL formalization of subroutines in the μJVM (§4), the bytecode verifier (§5), and its proof of correctness (§6).

1.1. Java Bytecode and the JVM

Sun specifies the JVM in [26] as a stack based interpreter of bytecode methods. It comprises a heap which stores objects and a method call stack, which captures information about currently active methods in the form of *frames*.

When the JVM invokes a method it pushes a new frame onto the frame stack to store the method's local data and its execution state. As Figure 1 indicates, each frame contains a program counter, an operand stack, and a register set.

Bytecode instructions manipulate either the heap, the registers, or the operand stack. For example, the *IAdd* instruction removes the topmost two values from the operand stack, adds them, and pushes the result back onto the stack. In the example in Figure 1 the two integers

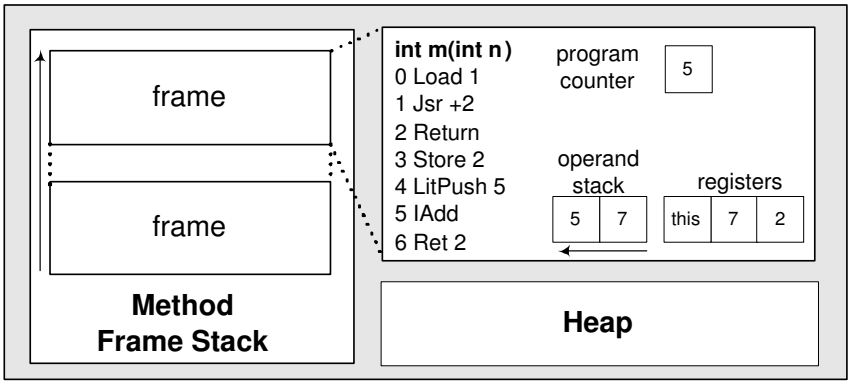

Figure 1. The JVM.

5 and *7* would be added. Apart from the operand stack, the JVM uses registers to store working data and arguments of a method. The first one (number *0*) is reserved for the *this* pointer of the method. The next p registers are reserved for the p parameters of a method, 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 to objects. Instructions can be executed normally or exceptionally. In the latter case the JVM sets the program counter to an appropriate exception handler whose code is embedded in the bytecode program. Each bytecode method has an exception handler table which is a list of tuples (s,e,pc',C). The intervall $[s,e)$ is the area that is protected by the handler, *pc'* is the start address of the exception handling code, and C is the exception class that is handled. When an exception E occurs, the JVM searches the table for the first entry where E is a subclass of C and where the program counter is in the protected area. Then it sets the program counter to *pc'*.

1.2. Bytecode Subroutines

The literature agrees on bytecode subroutines as one of the major sources of complexity for bytecode verification. 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 put 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` (see Figure 3 below for an example). In contrast to method calls, subroutines share the frame with their caller. Hence, a subroutine manipulates the same register set and stack as its caller. Two bytecode instructions, namely *Jsr b* and *Ret x*, handle subroutine

calls and returns. If the JVM encounters a *Jsr b* instruction, it pushes the return address (the program counter incremented by *1*) onto the stack and branches control to address *pc+b*. For example, the program in Figure 1 contains a subroutine which starts at address *3* and is called from address *1*. We call *3* the **entry point**, *1* the **call point**, and *2* the **return point** of the subroutine. 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). This means the return address pushed onto the stack by *Jsr* needs to be transferred to a register first. Therefore, the instruction at the subroutine entry point is usually a *Store x* which takes the topmost value of the stack and moves it to register x.

1.3. Bytecode Verification

The purpose of the bytecode verifier is to filter out erroneous and malicious bytecode programs prior to execution. It guarantees the following safety properties.

Argument safety Bytecode instructions receive their arguments in correct number, order and type.

Stack safety The operand stack cannot overflow or underflow.

Program counter safety The *pc* never falls off the code range.

Initialization safety Objects are properly initialized before they are used.

The usual technique for checking these properties is abstract interpretation of the underlying bytecode program. This means instead of values we only consider their types. This abstraction allows us to view 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 information for the operand stack and registers. For example the state type ([],[*Class B*, *Int*]) in Figure 2 characterizes all states whose stack is empty, whose register *0* contains a reference to an object of class B (or to a subclass of B), and whose register *1* contains an integer. We say a method is *welltyped* if we can assign stable state types to each instruction. A state type (*st*,*lt*) is stable for an instruction if it can be executed safely on a state whose stack is typed according to *st* and whose registers are typed according to *lt*. In other words, the arguments of the instruction are provided in correct number, order, and type.

Let us look at an example. In Figure 2 on the left the instructions are shown and on the right the type of the stack elements and the registers. The method type is the right-hand side of the table, a state type is one line of it. The type information attached to an instruction characterizes the state *before* execution of that instruction. We assume that class B is a subclass of A and that A has a field F of type A.

Execution starts with an empty stack and the two registers hold a reference to an object of class B and an integer. The first instruction loads register *0*,

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

Figure 2. Example of a welltyping.

a reference to a B object, on the stack. The type information associated with the following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that usage of register *1* may produce an error. This means the type information has become less precise but is still correct: a B object is also an A object and an integer is now classified as unusable (*Err*). The reason for these more general types is that the predecessor of the *Store* instruction may have either been *Load 0* or *Goto −3*. Since there exist different execution paths to reach *Store*, the type information of the two paths has to be "merged". The type of the second register is either *Int* or *Class A*, which are incompatible, i.e. the only common supertype is *Err*.

Bytecode verification is the process of inferring the types on the right from the instruction sequence on the left and some initial condition, and of ensuring that each instruction receives arguments of the correct type. Type inference is the computation of a method type from an instruction sequence, type checking means checking that a given method type fits an instruction sequence.

Figure 2 was an example for a welltyped method (we were able to find stable state types for all instructions). If we changed the third instruction from *Load 0* to *Store 0*, the method would not be welltyped. The *Store* instruction would try to take an element from the empty stack and could therefore not be executed. We would also not be able to find any other method type that is stable in all instructions.

2. Problems

A bytecode verifier checking code with subroutines faces the following difficulties.

Successor of Ret After the BV has analyzed an instruction, it has to compute its successors in order to propagate the resulting state type. The successors of *Ret x* instructions are hard to determine because return addresses are values and not accessible on the type

	instruction		stack	registers		source
0	*LitPush 5*	(	[],	[*Err*, *Err*, *Err*]	)	
1	*Store 0*	(	[*Int*],	[*Err*, *Err*, *Err*]	)	
2	*Load 0*	(	[],	[*Int*, *Err*, *Err*]	)	
3	*LitPush 0*	(	[*Int*],	[*Int*, *Err*, *Err*]	)	`int ml() {`
4	*Ifcmpeq +4*	(	[*Int*, *Int*],	[*Int*, *Err*, *Err*]	)	`  int i=5;`
5	*Jsr +8*	(	[],	[*Int*, *Err*, *Err*]	)	`  try {`
6	*Load 0*	(	[],	[*Int*, *Err*, *RA*]	)	`   if (i!=0) {`
7	*Return*	(	[*Int*],	[*Int*, *Err*, *RA*]	)	`     return i;`
8	*LitPush 7*	(	[],	[*Int*, *Err*, *Err*]	)	`   }`
9	*Store 1*	(	[*Int*],	[*Int*, *Err*, *Err*]	)	`   int j=7;`
10	*Jsr +3*	(	[],	[*Int*, *Int*, *Err*]	)	`   return j;`
11	*Load 1*	(	[],	[*Int*, *Err*, *RA*]	)	`  }`
12	*Return*	(	[*Err*],	[*Int*, *Err*, *RA*]	)	`  finally {`
13	*Store 2*	(	[*RA*],	[*Int*, *Int*⊔*Err*, *Err*]	)	`   if (i!=1) {`
14	*Load 0*	(	[],	[*Int*, *Err*, *RA*]	)	`     i=3;`
15	*LitPush 1*	(	[*Int*],	[*Int*, *Err*, *RA*]	)	`   }`
16	*Ifcmpeq +3*	(	[*Int*, *Int*],	[*Int*, *Err*, *RA*]	)	`  }`
17	*Litpush 3*	(	[],	[*Int*, *Err*, *RA*]	)	`}`
18	*Store 0*	(	[*Int*],	[*Int*, *Err*, *RA*]	)	
19	*Ret 2*	(	[],	[*Int*, *Err*, *RA*]	)	

Figure 3. Bytecode program with a subroutine.

level. For example, in Fig. 3 at address 19, the bytecode verifier has to find out that the return address *RA* stored in register 3 refers to the addresses 6 or 11.

Register Polymorphism Subroutines may have multiple call points, each with different types for some registers. We expect that registers not used inside the subroutine have the same type before and after the subroutine's execution. In the example in Fig. 3 at address 13, 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 lose information about the original types. If we propagate the merged type back to the return points, some programs will not be accepted, because they expect the original, more specific type. For example, bytecode verification fails at address 12 in Fig. 3, because the instruction there expects the original *Int* from address 10 in register 2. Note that this problem mainly occurs with registers that are not used inside the subroutine. We call these subroutines *polymorphic over unused registers*. Although rare in practice, subroutines could also be polymorphic

over used registers. For example, consider a subroutine that copies an array irrespective of its element types.

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 points 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 even further up in the subroutine call stack.

3. Solutions

3.1. Subroutine Labeling

Using typing rules, Stata and Abadi [41] specify safety properties for bytecode programs with subroutines. Their main issues are the polymorphism problem with unused registers and the determination of the successors of a *Ret x* instruction. To address the first problem they capture the type information of registers in type maps; each address in the program receives its own type map, which maps each register to its current type. Registers not used inside a subroutine are kept out of the domain of type maps. When two type maps need to be merged, for example at the entry point of a subroutine, only the types of registers in the domains of both maps get merged. Hence, unused registers are not affected by type merges. Their type, however, needs to be restored when a subroutine returns. Stata and Abadi accomplish this by using the types the unused registers have at the subroutine's relevant call point. This call point is the instruction immediately before the one the subroutine returns to. To determine return addresses statically, Stata and Abadi label all addresses. Each label tells us which subroutines are active when we reach the address the label is assigned to. If we can reach an address under more than one subroutine call stack, the label is a linearization of all these subroutine call stacks. Freund and Mitchell [16, 14, 13] point out that this technique fails when subroutines do not return to their caller, but to their caller's caller or even

further up in the subroutine call stack. To support these multilevel returns, they rather label each address with a set of subroutine call stacks. In addition, Freund and Mitchell improve [41] in various other ways: they not only state welltypedness conditions and prove them sound, but also propose algorithms to check them. They introduce additional constraints to exclude improper exception handling, improper object initialization, and bad interactions these two may have with subroutines.

3.2. Polyvariant Analysis

Leroy [24] proposes a **polyvariant analysis** to tackle subroutines. In contrast to traditional (monovariant) data flow analysis, a verifier based on polyvariant analysis maintains multiple state types per address, ideally one for each control flow route that reaches this address. This avoids type clashes at the entry point t of a subroutine with multiple call points cp. Not only the types at cp, but also those at the call points of all active[1] subroutines may influence the types at this entry point t. Hence, Leroy captures the entire subroutine call history; he uses *contours*, which are stacks of subroutine return points. The type information he assigns to addresses consists of state types paired with contours. The polyvariant bytecode verifier traverses the control flow path and separately analyzes each address for each contour that reaches it. To each address it assigns a pair of the current contour and the current state type. Contours approximate control flow routes sufficiently. They enable the BCV to distinguish between state types originating from different call histories and avoid the loss of precision type merges would bring along. In Figure 3, the polyvariant approach avoids the type merge of *Int* and *Err* at address 13 by typing this address with $\{[6] \mapsto ([RA\ 6], [Int, Err, Err]), [11] \mapsto ([RA\ 11], [Int, Int, Err])\}$, which keeps the state types originating from the contours [6] and [11] separate.

The absence of type merges at subroutine entry points allows Leroy to lift return addresses to the type system. For example, $RA\ r$ denotes the return address r. When he analyzes a $Ret\ x$ instruction under the contour ξ and the state type (st, lt), Leroy obtains the return address r stored in register x from the type stored in lt at position x. Assume he finds lt at x is $RA\ r$; in this case he can split the contour ξ into two parts ξ_1 and ξ_2, such that $\xi = \xi_1@[r]@\xi_2$ (@ is Isabelle's append operator for lists), and propagate the pair $(\xi_2, (lt, st))$ to the successor address r. Note that Leroy does not avoid type merges entirely. If a pair $(\xi_2, (st', lt'))$ is already assigned to address r, Leroy's verifier replaces it

[1] A subroutine is active if it has been called but has not returned yet.

with $(\xi_2, (st \sqcup st', lt \sqcup lt'))$, where $\sqcup$ computes the least common super type of two type lists.

3.3. Type sets

In Coglio's solution [9], 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 much more precise. Since the original type information is not lost, polymorphism of unused registers is not a problem anymore. Due to the absence of type merges, Coglio, too, is able to lift return addresses into his type system. Again, *RA r* denotes the return address *r*. This makes it possible to compute the successors of *Ret x* instructions and to propagate only the relevant types to them. For example, if the instruction *Ret 1* is at address i and the type set $\{([], [Int, RA\ 3]), ([], [Class\ A, RA\ 7])\}$ is assigned to i, then Coglio's verifier propagates $\{([], [Int, RA\ 3])\}$ to address *3* and $\{([], [Class\ A, RA\ 7])\}$ to address *7*. This accurately models the machine behavior. For his simple model of the JVM (not even containing classes), Coglio can give a nice characterization of the set of accepted programs: he proves that all type safe programs are welltyped, provided the programs run in his so-called integer-insensitive semantics that interprets conditional jumps as nondeterministic branches. Contrary to the other approaches, he does not take object initialization or exception handling into account.

3.4. Subroutine Expansion

Most difficulties with subroutines arise because different calling contexts have to be respected during the analysis of subroutine code. If every subroutine only had one call point, bytecode verifiers could handle *Jsr* and *Ret* similarly to ordinary jump instructions. This prompts the idea of transforming incoming bytecode prior to verification so that only subroutines of the form above appear. Since subroutines must not be recursive, we can expand their code and copy it directly to the call points. In [44], we discuss this approach in detail by specifying and implementing a program that performs subroutine expansion. We prove that this program transformation preserves the semantics of the underlying code.

Although this approach may expand the code size of the analyzed program exponentially to its original size, it is feasible, because subroutines are rarely nested in practice.

The main difficulty a subroutine expansion program faces are the unclear boundaries of subroutines in a bytecode program. In particular, jump instructions and exception throws impose problems. Do the jump targets or the corresponding exception handler still belong to the current subroutine and must therefore be expanded or do they not? In our expansion algorithm, we solve this problem with a simple control flow analysis. Whenever we reach an instruction, we mark it with its *complex address* which is the call point sequence of the current subroutine call history. When we jump to an instruction marked with a prefix of the current complex address, we assume that we leave the subroutine and stop expanding its code. Otherwise we continue the expansion assuming the jump target still belongs to the current subroutine. Even if this assumption is wrong, the expanded program still behaves the same way as the original one, as our bisimularity proof shows.

3.5. Evaluation

Freund's and Mitchell's bytecode language seems to be the closest approximation of the official Java Bytecode Language [26] the current literature about bytecode verification has to offer. However, type checking and type inference are rather complicated in this approach, because labels and unused registers need to be determined first. The type checking rules in [24] and [9] are simpler and even accept more typesafe programs. Figure 4 visualizes the relative completeness of the discussed approaches; we write JVMLcom to denote the intersection of the bytecode languages these approaches use. The hierarchy of type systems this diagram visualizes is backed up by proofs and (counter-) examples in [44]. Subroutine expansion is not part of Figure 4, as it is not directly a type system. Complex addresses and Leroy's contours are closely related, though. One is a sequence of call points, whereas the other is a sequence of return points. In fact, polyvariant analysis is more or less equivalent to conventional data flow analysis on an expanded program.

3.6. Related work

Apart from [41], [13], [24], and [9], which we summarized and compared above, the literature about subroutines and bytecode verification offers various other approaches.

In [33], Posegga and Vogt look at bytecode verification from a model checking perspective. They transform a given bytecode program into a finite state machine and check type safety, which they phrase in terms of temporal logic, by using an off-the-shelf model checker. Basin, Friedrich,

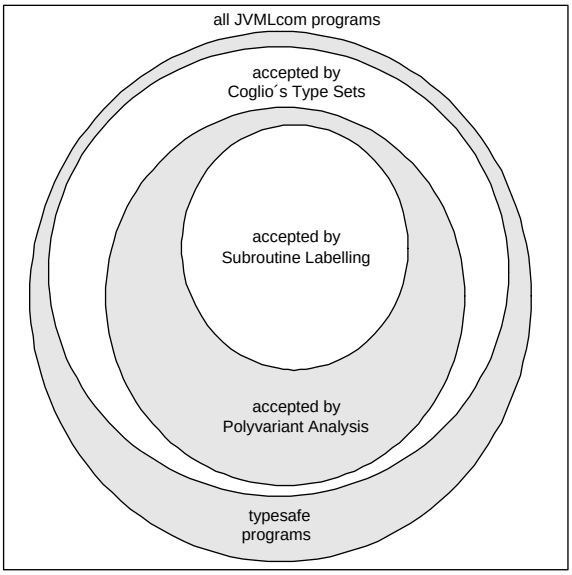

Figure 4. Typing rules hierarchy.

and Gawkowski [3] use Isabelle/HOL, μJava, and the abstract BCV framework [28] to prove the model checking approach correct.

O'Callahan [32] uses type variables and continuations to handle subroutines. Although his approach accepts a large portion of typesafe programs—even recursive subroutines could be supported—it remains unclear whether it can be realized efficiently.

Hagiya and Tozawa [18] propose welltypedness conditions in the form of typing rules similar to [41]. They avoid type merges by using indirect types.

Stärk *et al.* use Java and the JVM as a case study for abstract state machines (ASM). In [40], they formalize the process from compilation of Java programs down to bytecode verification. Their main theorem says that a the bytecode verifier accepts all byte code programs the compiler generates from valid[2] Java sources. They use tool support for the specification. Proofs, however, are by pen and paper.

The Kimera project [38] treats bytecode verification in an empirical manner. Its aim is to check bytecode verifiers by automated testing.

Using the B method, Casset [6] specifies a bytecode verifier for the JavaCard VM, which he then refines into an executable program

[2] Stärk *et al.* introduce a stronger constraint for definite assignments than the JVM specification.

that provably satisfies the specification. He does not show type safety, however.

Using SPECWARE, Qian, Goldberg and Coglio have specified and analyzed large portions of a Java bytecode verifier [11]. Apart from this work, Goldberg [17] and Qian [35, 36] have specified the JVM and its bytecode verifier mathematically. Coglio analyzes the traditional data flow analysis approach for bytecode verification [26] and makes several contributions to it in [10, 7].

Bertot [5] uses the Coq system to prove the correctness of a bytecode verifier based on [15]. He focuses on object initialization only.

Barthe *et al.* [1, 2] also 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.

4. The μJava VM

This and the following sections 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; this will later be the basis for the type safety proof. In §5 we develop the bytecode verifier itself, starting from an abstract framework in §5.1 that we then instantiate step by step in §5.2 to §5.4. The type safety theorem in §6 concludes the technical part.

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 for the most part directly use Isabelle/HOL notation as it appears in the Isabelle theories. Isabelle/HOL notation coincides mostly with what is used in mathematics and functional programming. We will show some of the basics now and then introduce new notation as we go along. For those curious for more on Isabelle/HOL, we recommend [31].

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. In order to lessen confusion between types in the programming language under consideration and types in our modeling language, the latter are sometimes referred to as **HOL types**.

List operations may be unfamiliar: the "cons" operator 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 following (polymorphic) declaration extends an existing type α with a new element *None*.

$$datatype\ \alpha\ option = Some\ \alpha \mid None$$

Functions in HOL are total, but they may be underspecified. The destructor *the*, for example, is defined by $the\ (Some\ x) = x$. The value for *None* exists, but is unknown.

We shall now take a look at the structure of the μJava VM (§4.1) and its operational semantics, first without (§4.2) and then with (§4.3) runtime type checks.

4.1. Structure

To keep things abstract and manageable, the source and bytecode language in μJava share as many notions as possible, i.e. program structure, wellformedness of programs, large parts of the type system, etc. Because of that, the μJVM does not have a notion of class files and constant pool like Sun's JVM has. A program is a list of classes and their declaration information, a class a list of fields and methods. For bytecode verification we will only be concerned with the method level, so we only describe the types at this level in more detail:

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

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

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

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

$$jvm\text{-}state \quad = \quad val\ option \times aheap \times frame\ list$$

The heap is simple: a partial function from locations to objects.

$$aheap = loc \Rightarrow obj\ option.$$

Here, *loc* is the type of addresses and *obj* is short for $cname \times (vname \times cname \Rightarrow val\ option)$, i.e. each object is a pair consisting of a class name (the class the object belongs to) and a mapping for the fields of the object (taking the name and defining class of a field, and yielding its value if such a field exists, *None* otherwise).

As in the real JVM, each method execution gets its own call frame, containing its own operand stack (a list of values), its own set of registers (also a list of values), and its own program counter. We also store the class and signature (i.e. name and parameter types) of the method and arrive at:

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

The alert reader of [23] may be missing an additional component *iheap* that has to do with object initialization. Although object initialization and subroutines are not orthogonal features, and the verification of either might fail because of the other, the *iheap* component is only needed deep within the proof of type safety. Adding it here to the description of the formalization would give no further insights. Our full formalization (available at [43]) still contains it, but we have left it out here for clarity.

4.2. Operational semantics

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

Figure 5 shows the instruction set. Method bodies are lists of such instructions together with the exception handler table and two integers *mxs* and *mxl* containing the maximum operand stack size and the number of local variables (not counting the *this* pointer and parameters of the method which get stored in the first *0* to n registers). So the type parameter γ for method bodies gets instantiated with $\mathit{nat} \times \mathit{nat} \times \mathit{instr\ list} \times \mathit{ex\text{-}table}$, i.e. *mdecl* becomes the following.

$$\mathit{mdecl} = \mathit{sig} \times \mathit{ty} \times \mathit{nat} \times \mathit{nat} \times \mathit{instr\ list} \times \mathit{ex\text{-}table}$$

The exception table is a list of tuples of *start-pc*, *end-pc*, *handler-pc* and exception class.

$$\mathit{ex\text{-}table} = (\mathit{nat} \times \mathit{nat} \times \mathit{nat} \times \mathit{cname})\ \mathit{list}$$

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

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

It says that execution halts if the call frame stack is empty or an unhandled exception has occurred. In all other cases execution is defined, *exec* decomposes

datatype instr =		
	Load nat	load from register
|	*Store nat*	store into register
|	*LitPush val*	push a literal (constant)
|	*New cname*	create object on heap
|	*Getfield vname cname*	fetch field from object
|	*Putfield vname cname*	set field in object
|	*Checkcast cname*	check if object is of class *cname*
|	*Invoke cname mname* (*ty list*)	invoke instance method
|	*Invoke-spcl cname mname* (*ty list*)	invoke constructor
|	*Return*	return from method
|	*Dup*	duplicate top element
|	*IAdd*	integer addition
|	*Goto int*	go to relative address
|	*Ifcmpeq int*	branch if equal
|	*Throw*	throw exception
|	*Ret nat*	return from subroutine
|	*Jsr int*	jump to subroutine

Figure 5. The μJava instruction set.

the top call frame, looks up the current method, retrieves the instruction list (the 5th element) of that method, delegates actual execution for single instructions to *exec-instr*, and finally sets the *pc* to the appropriate exception handler (with *find-handler*) if an exception occurred. As throughout the rest of this article, the program Γ is treated as a global parameter.

Exception handling in *find-handler* is as in the JVM specification: it looks up the exception table in the current method and sets the program counter to the first handler that protects *pc* and that matches the exception class. If there is no such handler, the topmost call frame is popped, and the search continues recursively in the invoking frame. If this procedure does find an exception handler, it clears the operand stack and puts a reference to the exception on top. If it does not find an exception handler, the exception flag remains set and the machine halts.

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

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

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

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

The definitions for *Jsr* and *Ret* are equally short:

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 also called *aggressive*, because it does not perform any runtime type or sanity checks. It just assumes that everything is as expected, e.g. for *Ret x* that in register *x* there is indeed a return address and that this return address points to a valid instruction inside the method. If the situation is not as expected, the operational semantics is unspecified at this point. In Isabelle, this means that there is a result (because HOL is a logic of total functions), but nothing is known about that result. It is the task of the bytecode verifier to ensure that this does not occur.

4.3. A Defensive VM

Although it is possible to prove type safety by using the aggressive VM alone, it is crisper to write and a lot more obvious to see just what the bytecode verifier guarantees when we additionally look at a defensive VM. Our defensive VM builds on the aggressive one by performing extra type and sanity checks. We can then state the type safety theorem by saying that these checks will never fail if the bytecode is welltyped.

To indicate type errors, we introduce another datatype.

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

Similar to §4.2, we build on a function *check-instr* that is lifted over several steps. At the deepest level, we take apart the state to feed *check-instr* with parameters (which are the same as for *exec-instr* plus the size of the method):

check :: *jvm-state* ⇒ *bool*
check (*xp*, *hp*, []) = *True*
check (*xp*, *hp*, *f*#*frs*) = *let* (*stk*,*reg*,*C*,*sig*,*pc*) = *f*;
ins = *5th* (*the* (*method* (Γ,*C*) *sig*));
i = *ins*!*pc*
in check-instr i hp stk reg C sig pc (*size ins*) *frs*

The next level is the one step execution of the defensive VM which stops in case of a type error and calls the aggressive VM after a successful check:

exec-d :: *jvm-state type-error* ⇒ *jvm-state option type-error*
exec-d TypeError = *TypeError*
exec-d (*Normal s*) = *if check s then Normal* (*exec s*) *else TypeError*

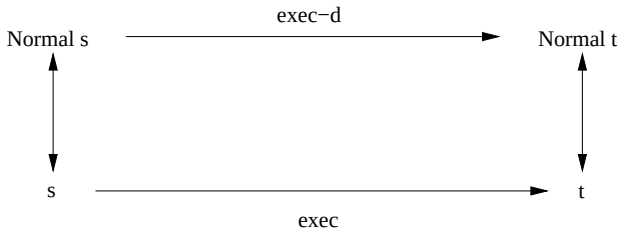

Figure 6. Aggressive and defensive μJVM commute if there are no type errors.

Again we take the reflexive transitive closure after getting rid of the *Some* and *None* constructors:

$$
\begin{array}{l}
s \xrightarrow{\text{djvm}} t \equiv \\
(s,t) \in (\{(s,t)|\ \textit{exec-d}\ s = \textit{TypeError} \wedge t = \textit{TypeError}\} \cup \\
\qquad \{(s,t)|\ \exists\, t'.\ \textit{exec-d}\ s = \textit{Normal}\ (\textit{Some}\ t') \wedge t = \textit{Normal}\ t'\})^*
\end{array}
$$

It remains to define *check-instr*, the heart of the defensive μJava VM. Again, this is relatively straightforward. A typical example is the *IAdd* instruction which requires two elements of type *Integer* on the stack.

$$
\begin{array}{l}
\textit{check-instr IAdd hp stk regs Cl sig pc maxpc frs} = \\
1 < \textit{size stk} \wedge \textit{isIntg}\ (\textit{hd stk}) \wedge \textit{isIntg}\ (\textit{hd}\ (\textit{tl stk})) \wedge \textit{pc+1} < \textit{maxpc}
\end{array}
$$

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:

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

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:

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

We have shown that defensive and aggressive VM have the same operational one step semantics if there are no type errors.

THEOREM 4.1.

$$\textit{exec-d}\ (\textit{Normal}\ s) \neq \textit{TypeError} \longrightarrow \textit{exec-d}\ (\textit{Normal}\ s) = \textit{Normal}\ (\textit{exec}\ s)$$

Figure 6 depicts this result as a commuting diagram.

For executing programs we will later also need a canonical start state. In the real JVM, a program is started by invoking its static *main* method. In the μJVM this is similar. We call a method *main method* of class C if there is a method body b such that $\textit{method}\ (\Gamma, C)\ (\textit{main}, []) = \textit{Some}\ (C,\ b)$ holds.

For *main* methods we can define the canonical start state *start* Γ C as the state with exception flag *None*, an otherwise empty heap *start-hp* Γ that has preallocated system exceptions, and a method invocation frame stack with one element: empty operand stack, *this* pointer set to *Null*, the rest of the register set filled up with a dummy value *arbitrary*, the class entry set to C, signature to (*main*,[]), and program counter *0*.

$$\begin{aligned}
&\mathit{start} :: \mathit{jvm\text{-}prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{jvm\text{-}state}\\
&\mathit{start}\ \Gamma\ C \equiv \mathit{let}\ (\text{-},\text{-},\text{-},\mathit{mxl},\text{-},\text{-}) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{main},[]));\\
&\qquad\qquad \mathit{regs} = \mathit{Null}\ \#\ \mathit{replicate}\ \mathit{mxl}\ \mathit{arbitrary}\\
&\quad \mathit{in}\ \mathit{Normal}\ (\mathit{None},\ \mathit{start\text{-}hp}\ \Gamma,\ [([],\ \mathit{regs},\ C,\ (\mathit{main},[]),\ \mathit{0})])
\end{aligned}$$

5. The Bytecode Verifier

5.1. An Abstract Framework

The abstract framework for data flow analysis is independent of the JVM, its typing rules and instruction set. Since it is a slightly extended version of the framework already presented in [28] and (with more detail) in [23], we concentrate on the general setting, the result of the data flow analysis, and the conditions under which such a result is guaranteed. We leave out the data flow analysis itself, i.e. Kildall's algorithm.

5.1.1. *Orders and semilattices*

This section introduces the HOL-formalization of the basic lattice-theoretic concepts required for data flow analysis and its application to the JVM.

Partial orders Partial orders are formalized as binary predicates. Based on the type synonym $\alpha\ \mathit{ord} = \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$ 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 is a **partial order** iff the predicate $\mathit{order} :: \alpha\ \mathit{ord} \Rightarrow \mathit{bool}$ holds for r:

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

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

Semilattices Based on the two type synonyms $\alpha\ \mathit{binop} = \alpha \Rightarrow \alpha \Rightarrow \alpha$ and $\alpha\ \mathit{sl} = \alpha\ \mathit{set} \times \alpha\ \mathit{ord} \times \alpha\ \mathit{binop}$ and the supremum notation $x +_f y = f\ x\ y$, we call $(A,r,f) :: \alpha\ \mathit{sl}$ a **semilattice** iff the predicate $\mathit{semilat} :: \alpha\ \mathit{sl} \Rightarrow \mathit{bool}$ holds:

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

where *closed* $A\ f = \forall\, x\ y \in A.\ x +_f y \in A$.

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

In the following sections let $(A,\ r,\ f)$ be a semilattice.

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

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:

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

An ordering r on α can be lifted to $\alpha\ \mathit{err}$ by making *Err* the top element:

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

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

The following lifting functional is useful below:

$$\begin{array}{lll} \mathit{lift2} :: (\alpha \Rightarrow \beta \Rightarrow \gamma\ \mathit{err}) \Rightarrow \alpha\ \mathit{err} \Rightarrow \beta\ \mathit{err} \Rightarrow \gamma\ \mathit{err} & & \\ \mathit{lift2}\ f\ (\mathit{OK}\ x)\ (\mathit{OK}\ y) & = & f\ x\ y \\ \mathit{lift2}\ f\ \_\ \_ & = & \mathit{Err} \end{array}$$

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

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

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

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

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

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

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

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:

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

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

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

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

5.1.2. *Stability*

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 behavior. These functions form the parameters of our model, namely the type system and the data flow analyzer. In the Isabelle formalization, these functions are parameters of everything. In this article, we often make them "implicit parameters", i.e. we pretend they are global constants, thus increasing readability.

Data flow analysis and type systems are based on an abstract view of the semantics of a program in terms of types instead of values. Since our programs are sequences of instructions the semantics can be characterized by a function $step :: nat \Rightarrow \sigma \Rightarrow (nat \times \sigma)\ list$. It is the abstract execution function: $step\ p\ s$ provides the results of executing the instruction at p starting in state s together with the positions to which these results are propagated. Contrary to the usual concept of *transfer function* or *flow function* in the literature, $step\ p$ not only provides the result, but also the structure of the data flow graph at position p. This is best explained by example. Figure 7 depicts the information we get when $step\ 3\ s_3$ returns the list $[(1,t_1),(4,t_4)]$: executing the instruction at position *3* with state type s_3 may lead to position *1* in the graph with result t_1, or to position *4* with result t_4.
Note that the length of the list and the target instructions do not only depend on the source position p in the graph, but also on the value of s. It is possible

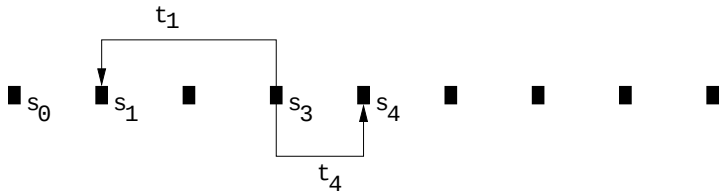

Figure 7. Data flow graph for *step* 3 $s_3 = [(1,t_1),(4,t_4)]$

(and for the *Ret* instruction even necessary) that the structure of the data flow graph dynamically changes in the iteration process of the analysis. It may not change freely, however. For the analysis to succeed, *step* must be **monotone** up to n:

$$\begin{array}{l} \mathit{mono\ step\ n} \equiv \\ \quad \forall\, s\ p\ t.\ s \in A \wedge t \in A \wedge p < n \wedge s \leq_r t \longrightarrow \mathit{step\ p\ s} \mid\leq_r\mid \mathit{step\ p\ t} \end{array}$$

where

$$xs \mid\leq_r\mid ys \equiv \forall\,(p,s) \in set\ xs.\ \exists\, s'.\ (p,s') \in set\ ys \wedge s \leq_r s'$$

This means, 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).

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

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

$$\mathit{stable\ ss\ p} \equiv \forall\,(q,s') \in \mathit{set}(\mathit{step\ p\ (ss!p)}).\ s' \leq_r \mathit{ss}!q$$

Stability induces the notion of a method type *ts* being a **welltyping w.r.t.** *step*:

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

Here, $\top$ is assumed to be a special element in the state space indicating a type error. It is the top element of the ordering.

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

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

The notation $\leq[r]$ lifts $\leq_r$ to lists (see §5.1.1). In practice, *bcv ss* itself will be the welltyping, and it will also be the least welltyping. However, it is simpler not to require this.

The introduction of a subset A of the state space σ is necessary to make distinctions beyond HOL's type system: for example, when representing a list of registers, σ is likely to be a HOL list type; but the fact that in any particular program the number of registers is fixed cannot be expressed as a HOL type, because it requires dependent types to formalize lists of a fixed length. We use sets to express such fine-grained distinctions. We say *step* **preserves** A up to n iff for all $s{\in}A$ and $p{<}n$ the values *step s p* returns are in A:

$$\mathit{preserves}\ n \equiv \forall\, s{\in}A.\ \forall\, p{<}n.\ \forall\, (q,t){\in}\mathit{set}\ (\mathit{step}\ p\ s).\ t \in A$$

In [28, 23] we have defined and verified a functional version of Kildall's algorithm [20, 27], a standard data flow analysis tool that is a bytecode verifier in the sense above. In fact, the description of bytecode verification in the official JVM specification [26, pages 129–130] is essentially Kildall's algorithm. We do not show the definition here, but assume that there is a function $\mathit{kildall} :: \sigma\ \mathit{list} \Rightarrow \sigma\ \mathit{list}$ satisfying the following theorem.

THEOREM 5.1. *If* $(A,\ r,\ f)$ *is a semilattice,* r *meets the ascending chain condition, step is monotone up to* n*, preserves* A *up to* n*, and is bounded by* n*, then kildall is a bytecode verifier.*

5.2. The Semilattice

We now begin to instantiate the framework. Theorem 5.1 requires state types to form a semilattice that satisfies the ascending chain condition. In this section we shall build up a semilattice suitable for the treatment of the *Jsr* and *Ret* instructions.

Following the idea of Coglio [8, 9], we will use sets as the elements of the semilattice. The order is the usual subset relation $\subseteq$, and the supremum is union $\cup$. It is easy to see that $(\mathit{Pow}\ A,\ \subseteq,\ \cup)$ is a semilattice (where $\mathit{Pow}\ A$ is the power set of A).

Unfortunately, the subset relation allows infinitely ascending chains and therefore violates the ascending chain condition of theorem 5.1. If the carrier set A is finite, however, $\subseteq$ does satisfy the ascending chain condition on $\mathit{Pow}\ A$. The goal in this section is therefore to start out with a finite the set of basic types, and then to build up a structure which preserves this finiteness and which can describe the μJVM's operand stack and register set.

The HOL datatype of basic types in μJava is the following:

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

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*. Reference types are the null type (for the *Null* reference), and class types (*cname* is the type of class names). For readability, we use the following abbreviations, implemented as *syntax translations* in Isabelle:

$$\begin{aligned} \textit{translations} \qquad NT &== \textit{RefT NullT} \\ \textit{Class } C &== \textit{RefT } (\textit{ClassT } C) \\ \textit{RA } pc &== \textit{PrimT } (\textit{RetA } pc) \end{aligned}$$

On top of that come additional type distinctions for object initialization:

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

The distinction is between usual, initialized types *Init ty*, uninitialized types *UnInit C pc*, i.e. the type of objects of class C that have been created in line *pc*, and partly initialized classes *PartInit C*. For a more in-depth description of the object initialization part of our formalization see [23].

These are the basic types that may occur on the operand stack and in the register set of the bytecode verifier. They can easily be restricted to a finite subset: the set of class names can be restricted to the classes declared in the program, and the program counters occurring in return addresses and *UnInit* types can be restricted to the program counters that occur in the method. Formally, the carrier set *init-tys* is defined as follows:

$$\begin{aligned} \textit{init-tys max-pc} \equiv\ & \{\textit{Init } T \mid \textit{is-type } \Gamma\ T \wedge \textit{boundedRA } (\textit{max-pc}, T)\} \cup \\ & \{\textit{UnInit } C\ pc \mid \textit{is-class } \Gamma\ C \wedge pc < \textit{max-pc}\} \cup \\ & \{\textit{PartInit } C \mid \textit{is-class } \Gamma\ C\} \end{aligned}$$

The definition is in the context of a fixed program Γ; the parameter *max-pc* is the maximum program counter in the method to be verified. The predicates *is-class* and *is-type* hold if the class (or type) is declared in Γ; with *boundedRA* $(\textit{max-pc}, T)$ we check that the program counter is not greater than *max-pc* if T is a return address.

It remains to lift this set to the operand stack and register set structure of the bytecode verifier. The register set is a list of a fixed length *mxr*. Apart from basic types, it may contain unusable values that we denote by *Err*, introduced by the *err* function of §5.1.1. The operand stack is also a list, not of fixed length, but of maximum length *mxs*. Apart from operand stack and registers we also need a boolean flag for verifying constructors. It tells the BCV whether the super class constructor has already been called or not. Using *list n A* as in §5.1.1 for the set of lists over A with length n we arrive at:

$$\begin{aligned} \textit{state-types mxs mxr mpc} \equiv\ & ((\bigcup \{\textit{list } n\ (\textit{init-tys mpc}) \mid n \leq \textit{mxs}\}) \times \\ & \textit{list mxr } (\textit{err } (\textit{init-tys mpc}))) \times \\ & \{\textit{True}, \textit{False}\} \end{aligned}$$

This means in the terminology of §1.3 that the elements of *state-types* have the form $((st,\ lt),\ z)$ where st is a list that stands for the stack, lt a list that stands for the registers, and where z is the initialization flag mentioned above.

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

$$states\ maxs\ maxr\ maxpc \equiv err\ (Pow\ (state\text{-}types\ maxs\ maxr\ maxpc))$$

We have shown the following lemma.

LEMMA 5.2. *$(states,\ Err.le\ \subseteq,\ Err.sup\ \cup)$ is a semilattice and $Err.le\ \subseteq$ satisfies the ascending chain condition on states.*

Since we know that the state types are finite sets, we can replace them by a list implementation in a real BCV. In the ML code generated from the Isabelle specification (using [4]) we have done so; in the formalization itself it is more convenient to continue with sets.

5.3. Transfer Function

The single transfer function *step* of §5.1 is compact and convenient for describing the abstract typing framework. For a large instantiation, however, it carries too much information in one place to be modular and intuitive. We will therefore first refine *step* into applicability and effect of instructions in §5.3.1, and then instantiate the parts in §5.3.2.

5.3.1. *Refining step*

We refine *step* into two functions: one that checks the applicability of the instruction in the current state, and one that carries out the instruction assuming it is applicable. These two functions will be called *app* and *eff*. Furthermore, the state space σ will be of the form $\tau\ err$ for a suitable type τ, in which case the error element $\top$ is *Err* itself. Given functions $app :: nat \Rightarrow \tau \Rightarrow bool$ and $eff :: nat \Rightarrow \tau \Rightarrow (nat \times \tau)\ list$, and the size n of the instruction sequence, *step* is defined as follows:

$$step\ p\ Err = error\ n$$
$$step\ p\ (Ok\ t') = if\ app\ p\ t'\ then\ map\text{-}snd\ OK\ (eff\ p\ t')\ else\ error\ n$$

In this definition *error* n is a function that returns a list with *Err* for each position below n in the program, and *map-snd* f is $map\ (\lambda(x,y).\ (x,\ f\ y))$. Similarly, we can refine the notion of a welltyping w.r.t. *step* to a welltyping w.r.t. *app* and *eff*:

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

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

The notions *wt-step* and *wt-app-eff* coincide. We have proved the following two lemmas:

LEMMA 5.3. *If the composed function step is bounded by size ts, and all elements of ts are in err A, then*

$$\textit{wt-step ts} \longrightarrow \textit{wt-app-eff } (\textit{map ok-val ts})$$

where ok-val $(OK\ x) = x$.

In the other direction:

LEMMA 5.4. *If the composed function step is bounded by size ts, and all elements of ts are in A, then*

$$\textit{wt-app-eff ts} \longrightarrow \textit{wt-step } (\textit{map OK ts})$$

In our previous formalization [23], there was a slight asymmetry between the lemmas 5.3 and 5.4. The more general type of *step* and the function *error* make it unnecessary here.

5.3.2. *Applicability and Effect*

In this section we will 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. This subdivision is not as modular and abstract as the one into applicability and effect, but it helps to manage the size of the definitions.

Since the BCV verifies one method at a time, we can see the context of a method and a program as fixed for the definition. The context consists of the following values:

Γ	:: *program*	the program,
C'	:: *cname*	the class the method we are verifying is declared in,
mn	:: *mname*	the name of the method,
ini	:: *bool*	true iff the method is a constructor,
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 *eff* and *app* in the Isabelle formalization. We treat them as global here to spare the reader endless parameter lists in each definition.

As they are relatively uniform, we begin the definitions of *app* and *eff* with the exception handling part of *app*. It builds on *xcpt-names*, the purpose of which is to determine the exceptions that are handled in the same method. The *xcpt-names* function looks up for instruction i at position *pc* which handlers of the exception table *et* are possible successors. It returns a list of the exception class names that match. The functions *match* and *match-any* (neither shown here) in Figure 8 do the actual lookup: *match X pc et* returns $[X]$ if there is

$$
\begin{aligned}
&\mathit{xcpt\text{-}names} :: \mathit{instr} \times \mathit{nat} \times \mathit{ex\text{-}table} \Rightarrow \mathit{cname\ list}\\
&\mathit{xcpt\text{-}names}\ (\mathit{Getfield}\ F\ C,\ pc,\ et) &&= \mathit{match}\ \mathit{NullPointer}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (\mathit{Putfield}\ F\ C,\ pc,\ et) &&= \mathit{match}\ \mathit{NullPointer}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (\mathit{New}\ C,\ pc,\ et) &&= \mathit{match}\ \mathit{OutOfMemory}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (\mathit{Checkcast}\ C,\ pc,\ et) &&= \mathit{match}\ \mathit{ClassCast}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (\mathit{Throw},\ pc,\ et) &&= \mathit{match\text{-}any}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (\mathit{Invoke}\ C\ m\ p,\ pc,\ et) &&= \mathit{match\text{-}any}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (\mathit{Invoke\text{-}spcl}\ C\ m\ p,\ pc,\ et) &&= \mathit{match\text{-}any}\ pc\ et\\
&\mathit{xcpt\text{-}names}\ (i,\ pc,\ et) &&= []
\end{aligned}
$$

Figure 8. Exception names.

a handler for the exception X, and [] otherwise, while *match-any* returns the exception class names of all handlers that protect the instruction at *pc*.

Applicability in the end only requires that these class names are declared in the program:

$$
\begin{aligned}
&\mathit{xcpt\text{-}app} :: \mathit{instr} \Rightarrow \mathit{bool}\\
&\mathit{xcpt\text{-}app}\ i \equiv \forall\, C{\in}\mathit{set}(\mathit{xcpt\text{-}names}\ (i,pc,et)).\ \mathit{is\text{-}class}\ \Gamma\ C
\end{aligned}
$$

To determine the successors pc', 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* and the initialization flag of the current state type s remain the same; the stack is cleared, and a reference to the (initialized) exception object is pushed. The Isabelle notation f ‘ A is the image of a set A under a function f. The successor instruction pc' in the data flow graph marks the beginning of the exception handler. This effect occurs for every exception class C the instruction may possibly throw (determined by *xcpt-names* as for *xcpt-app* above).

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

This concludes the exception case and we proceed to the normal, non-exception case. Since the definitions become larger here, we first describe one simple instruction to give some intuition and then move on to the actual definition of *app* and *eff*.

The first observation is that it suffices to look at the elements of the state type separately: *app′* and *eff′* work on single stacks and on single register sets; *app* and *eff* then lift these to sets, i.e., complete state types. The second observation is that the successors pc' in the normal case can be defined separately from the resulting state type (with the exception of *Ret* which we discuss below).

For the *Dup* instruction the definitions are as follows:

$$\begin{array}{ll} app'\ (Dup,\ (t\#st,lt)) & = 1{+}size\ st < mxs \\ eff'\ (Dup,\ (t\#st,lt)) & = (t\#t\#st,lt) \\ succs\ Dup\ pc\ s & = [pc{+}1] \end{array}$$

The instruction is applicable if there is at least one element t on the stack, and if there is enough space to push another element onto it. The effect is the duplication of that element t and the successor is $pc{+}1$. Most of the instructions follow this simple pattern.

We now describe the full definition of applicability in the normal, non-exception case. In app', a few new functions occur: $typeof :: val \Rightarrow ty\ option$ returns *None* for addresses, and the type of the value otherwise; *field* is analogous to *method* and looks up declaration information of object fields (defining class and type); $rev :: \alpha\ list \Rightarrow \alpha\ list$, $zip :: \alpha\ list \Rightarrow \beta\ list \Rightarrow (\alpha \times \beta)\ list$, and $take :: nat \Rightarrow \alpha\ list \Rightarrow \alpha\ list$ are the obvious functions on lists.

Apart from these new functions, there is also the subtype ordering $\preceq$ that builds on the direct subclass relation *subcls* Γ induced by the program Γ. It satisfies:

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

where $(C,D) \in (subcls\ \Gamma)^*$ means that C is a subclass of D. The ordering $\preceq$ extends canonically to the object initialization layer: *PartInit* and *UnInit* are only related to themselves, for *Init* t we use the old $\preceq$. Formally:

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

Note that although the subtype relation is no longer used as the semilattice order in the BCV, it is still used 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 wether 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 pair of stack st and register set lt in the state type; the object initialization flag z (that app' has not handled yet) must be true for *Return* instructions in constructors, and it must be false when we invoke a superclass constructor; finally, to ensure that *step* in the end is bounded, we require that all successor program counters are in the method ($pc' < mpc$ is sufficient, because pc' is of type *nat*; hence $0 \leq pc'$ is trivially true):

$app :: instr \Rightarrow state\text{-}type \Rightarrow bool$
$app\ i\ s \equiv$

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

Figure 9. Applicability of instructions.

$$
\begin{array}{l}
succs :: instr \Rightarrow nat \Rightarrow state\text{-}type \Rightarrow nat\ list \\
succs\ (Ifcmpeq\ b)\ pc\ s = [pc{+}1,\ nat\ (int\ pc\ +\ b)] \\
succs\ (Goto\ b)\ pc\ s \quad = [nat\ (int\ pc\ +\ b)] \\
succs\ Return\ pc\ s \quad = [] \\
succs\ Throw\ pc\ s \quad = [] \\
succs\ (Jsr\ b)\ pc\ s \quad = [nat\ (int\ pc\ +\ b)] \\
succs\ (Ret\ x)\ pc\ s \quad = (SOME\ l.\ set\ l = theRA\ x\ `\ s) \\
succs\ i\ pc\ s \quad = [pc{+}1]
\end{array}
$$

Figure 10. Successor program counters for the non-exception case.

$$
\begin{array}{l}
(\forall\,((st,lt),z) \in s. \\
\quad xcpt\text{-}app\ i \wedge app'\ (i,(st,lt)) \wedge (ini \wedge i = Return \longrightarrow z) \wedge \\
\quad (\forall\ C\ m\ p.\ i = Invoke\text{-}spcl\ C\ m\ p \wedge st!size\ p = PartInit\ C' \longrightarrow \neg z)) \wedge \\
(\forall\,(pc',s') \in set\ (eff\ i\ s).\ pc' < mpc)
\end{array}
$$

This concludes applicability. It remains to build the normal, non-exception 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. Figure 10 shows the successors. Again, most 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 *theRA* x is defined by *theRA* x $((st,lt),z) =$ *the-RA* $(lt!x)$ and *the-RA* $(OK\ (Init\ (RA\ pc))) = pc$; it works on elements $((st,lt),z)$ of state types and extracts the return address that is stored in register x. The image of s under *theRA* x is the set of all different return addresses that occur in register x in s. In *app* we made sure that each element of s does have a return address at position x in the register set, so *theRA* is defined for all elements of s. Since *succs* returns lists and not sets, we use Hilbert's epsilon operator *SOME* to pick any list that converts to this set. Remember that in the implementation we plan to use lists for state types instead of sets, so this *SOME* is just the identity function here. In the proofs, *SOME* is not a problem, because we know that s is finite and therefore there always is such a list l to pick.

Because of this behavior of the *Ret* instruction in *succs*, our data flow analysis must be flexible enough to let the shape of the data flow graph change and depend on the current state of the calculation.

As with *app* we first define the effect *eff'* on single stack and register sets (Figure 11). It uses *theClass*, defined by *theClass* $(PartInit\ C) = Class\ C$ and *theClass* $(UnInit\ C\ pc) = Class\ C$, and *replace* $a\ b\ xs$ which replaces a by b in the list xs. The *method* expression for *Invoke* merely determines the return type of the method in question.

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

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

Figure 11. Effect of instructions on the state type.

in the beginning of this section, so *pc* is the program counter of the current instruction.

Before we turn our attention to *Ret*, we define the object initialization layer *eff-bool* that sets the z flag to true if the current instruction is *Invoke-spcl*, and leaves it unchanged otherwise. We abbreviate the rather long type expression (*init-ty list* × *init-ty err list*) × *bool* by *state-bool*.

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

If it was not for *Ret*, we could apply this *eff-bool* to every element of the state type. 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 if *i* is a *Ret* instruction.

norm-eff :: *instr* ⇒ *nat* ⇒ *state-type* ⇒ *state-type*
norm-eff i pc′ s ≡
(*eff-bool i*) ‘ (*if isRet i then* {*s′*| *s′*∈*s* ∧ *theRA* (*theIdx i*) *s′* = *pc′*} *else s*)

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 :: *instr* ⇒ *state-type* ⇒ (*nat* × *state-type*) *list*
eff i s ≡ (*map* (λ*pc′*. (*pc′*,*norm-eff i pc′ s*)) (*succs i pc s*)) @ (*xcpt-eff i s*)

5.4. Instantiating the Framework

Having defined the semilattice and the transfer function in §5.2 and §5.3, we show in this section how the parts are put together and how they form an executable bytecode verifier.

The basis of our bytecode verifier is Theorem 5.1. If we manage to satisfy its premises, the function *kildall* :: *state-type list* ⇒ *state-type list* is a bytecode verifier in the sense of §5.1. The first two of the premises of Theorem 5.1 require that the structure is a semilattice and that the order satisfies the ascending chain condition. Lemma 5.2 tells us that this is the case. We have also proved the other two premises.

LEMMA 5.5. *The transfer function step constructed from app and eff as shown in §5.3.1 is bounded and monotone up to mpc, and it preserves the carrier set up to mpc.*

The proof that *step* is bounded is easy, since *app* explicitly checks this condition. Monotonicity is not much harder. 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.

To turn *kildall* into a bytecode verifier in the sense of the JVM specification, i.e. into a function of type *bool*, 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, the next registers contain the parameters of the method, the rest of the registers is reserved for local variables (which do not have a value yet). Note that definitions are still in the context of a fixed method as defined in §5.3.2. The *this* pointer has type *Init* (*Class C′*) for normal methods, and *PartInit C′* for constructors. The initialization flag is set to the value

of C'=*Object*, because only class *Object* does not need to call its super class constructor. 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* of §5.3.2 by *mxr* = *1+size ps+mxl*). The definition of the start value φ_0 uses *ins*, the instruction list of the method. The state types of the other instructions are initialized with the empty set, the bottom element of the ordering.

$$this = OK\ (\mathit{if}\ ini \wedge C' \neq Object\ \mathit{then}\ PartInit\ C'\ \mathit{else}\ Init\ (Class\ C'))$$
$$S_0 = (([],this\#(map\ (OK \circ Init)\ ps)@(replicate\ mxl\ Err)),C'{=}Object)$$
$$\varphi_0 = (OK\ \{S_0\})\#(replicate\ (size\ ins{-}1)\ (OK\ \{\}))$$

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

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

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.

From Theorem 5.1 and lemmas 5.3 and 5.4 we get the following corollary.

COROLLARY 5.6. *wt-kil holds iff ins is not empty and there is a welltyping* φ *with* $S_0 \in \varphi!0$ *such that wt-app-eff* φ *holds.*

It is useful to introduce a predicate summarizing the conditions in the corollary above: *wt-method* φ holds by definition iff *ins* is not empty, S_0 is an element of $\varphi!0$, and *wt-app-eff* φ is true.

6. Type Safety

This section presents the type safety theorem. It says that the bytecode verifier is correct, that it guarantees safe execution.

Our type safety theorem talks about execution of whole programs, not only single methods for which we have defined the bytecode verifier in the sections above. Therefore we need to lift welltypings φ as well as the welltypedness condition *wt-method* from methods to programs. Welltypings of programs are functions Φ :: *cname* $\Rightarrow$ *sig* $\Rightarrow$ *state-type list* that return a welltyping for each method and each class in the program. We call a program welltyped if there is a welltyping Φ such that *wt-prog* Γ Φ holds. The function *wt-prog* returns true if *wt-method* (Φ *C sig*) holds for every *C* and *sig* such that *C* is a class in Γ and *sig* a method signature declared in *C*. Additionally, *wt-prog* checks that Γ is wellformed, i.e. that the class hierarchy is a well founded single inheritance hierarchy (otherwise the method and field lookup functions might not terminate). We lift the executable bytecode verifier to full programs in the same way: *wt-prog*$_k$ Γ holds by definition if Γ is wellformed and *wt-kil* holds for all methods in the program. Lemma 6.1 states that the relationship between the welltypedness predicate and the bytecode verifier is preserved.

LEMMA 6.1. *wt-prog*$_k$ Γ *holds iff there is a* Φ *such that wt-prog* Γ Φ *holds.*

The type safety theorem then is the following: if the bytecode verifier succeeds and we start the program Γ in its canonical start state (see §4.3), the defensive μJVM will never return a type error.

THEOREM 6.2. *If C is a class in Γ with a main method, then*

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

To prove this theorem, we set out from a program Γ for which the bytecode verifier returns true, i.e. for which $\textit{wt-prog}_K\ \Gamma$ holds. Lemma 6.1 tells us that there is a Φ such that *wt-prog* $\Gamma\ \Phi$ holds. The proof builds on the observation that all runtime states σ that conform to the types in Φ are type safe. If Φ conforms to σ, we write $\Phi \vdash \sigma\surd$. For $\Phi \vdash \sigma\surd$ to be true, the following must hold: if in state σ execution is at position *pc* of method (C,sig), then there must be an element s of the state type $(\Phi\ C\ sig)!pc$ such that for every value v on the stack or in the register set the type of v is a subtype of the corresponding entry in its static counterpart s. We have shown that conformance is invariant during execution if the program is welltyped.

LEMMA 6.3.

$$\textit{wt-prog}\ \Gamma\ \Phi \wedge \Phi \vdash \sigma\surd \wedge \sigma \xrightarrow{\text{jvm}} \tau \Longrightarrow \Phi \vdash \tau\surd$$

The proof of this central lemma is 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 $(\Phi\ C\ sig)$, and we conclude from the conformance of σ together with the *app* part of *wt-prog* 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 τ conforms to $t = \textit{eff pc s}$. This t is the element of $(\Phi\ C\ sig)!pc'$ that shows $\Phi \vdash \tau\surd$.

For the proof to go through, the intuitive notion of conformance we have given above is not enough, the formal conformance relation $\Phi \vdash \sigma\surd$ is stronger. It describes the states that can occur during execution, the form of the heap, the form of the method invocation stack, and the state of partly and completely uninitialized objects. As it is very large and technical (about four pages of Isabelle code) and [34, 30, 23, 21] already contain detailed descriptions of it, we will not formally define it here. It is not necessary to understand the conformance relation in detail in order to trust the proof as it is an intermediate proof device only. The correctness theorem itself does not contain the notion of conformance; it states the absence of type errors, which is a much easier concept. Since the proof is mechanically checked in Isabelle (and also human readable for later inspection), it is immaterial how large and complex the proof and its intermediate constructions are, as long as the final result is clear.

To conclude the final type safety theorem, Lemma 6.3 is still not enough. It might be the case that there is no σ such that $\Phi \vdash \sigma\surd$. Lemma 6.4 shows that this is not so.

LEMMA 6.4. *If C is a class in* Γ *with a main method, then*

$$\textit{wt-prog}\ \Gamma\ \Phi \Longrightarrow \Phi \vdash (\textit{start}\ \Gamma\ C)\surd$$

Lemmas 6.3 and 6.4 together say that all states that occur in any execution of program Γ conform to Φ if we start Γ in the canonical way.

The last step in the proof of Theorem 6.2 is Lemma 6.5: an execution step started in a conformant state cannot produce a type error in welltyped programs.

LEMMA 6.5.

$$\textit{wt-prog}\ \Gamma\ \Phi \wedge \Phi \vdash \sigma\surd \Longrightarrow \textit{exec-d}\ (\textit{Normal}\ \sigma) \neq \textit{TypeError}$$

The proof of Lemma 6.5 is a case distinction on the current instruction in σ. Similar to the proof of Lemma 6.3, the conformance relation together with the *app* part of *wt-prog* ensure *check-instr* in *exec-d* returns true. Because we know that all states during execution conform, we can conclude Theorem 6.2: there will be no type errors in welltyped programs.

7. Conclusion

We have presented a formalization of the μJVM with subroutines that are not artificially restricted for the sake of bytecode verification. We have instantiated our previous formalization of an abstract verified data flow analysis with a type system that supports classes, subroutines, object initialization, and exception handling. The bytecode verifier we have specified is fully executable (in Isabelle itself, and also as ML code generated from the Isabelle specification). We have proved in Isabelle/HOL that it is correct.

The treatment of subroutines caused the formalization to grow from about 16,000 to 17,000 lines of Isabelle code (including all specifications, lemmas, examples, and proofs). These 16,000 lines of Isabelle code already included the source language, a lightweight bytecode verifier, exception handling, object initialization, and arrays (which we have not shown here). In comparison to our earlier work in [22, 23, 34], the type safety statement has become clearer and easier to understand, because it is formulated with an additional defensive operational semantics instead of the large, complex invariant alone.

The type system we use is based on Coglio's idea [9] of using sets to avoid type merges altogether. Our formalization is more than a version of [9] in Isabelle/HOL, though: we have shown that the idea scales up to a realistic model of the JVM ([9] did not even have classes), and that subroutines do not necessarily interfere with exception handling

or object initialization as it is the case in [39] or [13] (both propose to restrict subroutines in the BCV to work around that).

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). The sets could grow at every join operation, i.e. at every join point of the data flow graph, or every time a usual bytecode verifier would perform a type merge. In practice, type merges occur rarely, because at join points the types on all paths are often already equal. Leroy finds in [25] that each instruction is analyzed 1.6 times on average before the fixpoint is reached (in a test case of 7077 JCVM instructions). Usually instructions are analyzed once, rarely twice. Our own experience agrees: even for contrived examples (taken from [39]), most sets were singletons; the maximum size of the sets was 4. Given an efficient implementation for sets of that size, there is no reason for a bytecode verifier with this type system to be slow in practice.

Contrary to the other approaches, the type system presented here is also directly applicable to lightweight bytecode verification [22, 37], eliminating the need to expand subroutines prior to verification on embedded devices.

Acknowledgements

We thank Stefan Friedrich, Tobias Nipkow, Norbert Schirmer, and Martin Strecker for many helpful discussions and for reading drafts of this paper.

References

1. Barthe, G., G. Dufay, L. Jakubiec, S. M. de Sousa, and B. Serpette: 2001, 'A Formal Executable Semantics of the JavaCard Platform'. In: D. Sands (ed.): *Proceedings of ESOP'01*, Vol. 2028 of *Lect. Notes in Comp. Sci.* pp. 302–319.
2. Barthe, G., G. Dufay, L. Jakubiec, S. M. de Sousa, and B. Serpette: 2002, 'A formal correspondence between offensive and defensive JavaCard virtual machines'. In: A. Cortesi (ed.): *Proceedings of VMCAI'02*. to appear.
3. Basin, D., S. Friedrich, and M. Gawkowski: 2002, 'Verified Bytecode Model Checkers'. In: *Theorem Proving in Higher Order Logics (TPHOLs'02)*, Vol. 2410 of *Lect. Notes in Comp. Sci.* Virginia, USA, pp. 47–66.
4. Berghofer, S. and T. Nipkow: 2002, 'Executing Higher Order Logic'. In: P. Callaghan, Z. Luo, J. McKinna, and R. Pollack (eds.): *Types for Proofs and Programs (TYPES 2000)*, Vol. 2277 of *Lect. Notes in Comp. Sci.* pp. 24–40.
5. Bertot, Y.: 2001, 'Formalizing a JVML Verifier for Initialization in a Theorem Prover'. In: *Computer Aided Verification (CAV'2001)*, Vol. 2102 of *LNCS*. pp. 14–24.

6. Casset, L.: 2002, 'Development of an Embedded Verifier for Java Card Byte Code Using Formal Method'. In: L.-H. Eriksson and P. A. Lindsay (eds.): *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, Vol. 2391 of *Lect. Notes in Comp. Sci.* pp. 290–309.
7. Coglio, A.: 2001a, 'Improving the Official Specification of Java Bytecode Verification'. In: *3rd ECOOP Workshop on Formal Techniques for Java programs*.
8. Coglio, A.: 2001b, 'Simple Verification Technique for Complex Java Bytecode Subroutines'. Technical report, Kestrel Institute.
9. Coglio, A.: 2002, 'Simple Verification Technique for Complex Java Bytecode Subroutines'. In: *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*.
10. Coglio, A. and A. Goldberg: 2001, 'Type Safety in the JVM: Some Problems in Java 2 SDK 1.2 and proposed solutions'. In: *Concurrency and Computation: Practice and Experience*. pp. 1153–1171.
11. Coglio, A., A. Goldberg, and Z. Qian: 2000, 'Toward a Provably-Correct Implementation of the JVM Bytecode Verifier'. In: *Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2*. pp. 403–410.
12. Freund, S. N.: 1998, 'The Costs and Benefits of Java Bytecode Subroutines'. In: *OOPSLA'98 Workshop Formal Underpinnings of Java*.
13. Freund, S. N.: 2000, 'Type Systems for Object-Oriented Intermediate Languages'. Ph.D. thesis, Stanford University.
14. Freund, S. N. and J. C. Mitchell: 1998a, 'A Type System for Object Initialiazation in the Java Bytecode Language'. In: *ACM Transactions on programming languages and systems*.
15. Freund, S. N. and J. C. Mitchell: 1998b, 'A Type System for Object Initialization in the Java Bytecode Language'. In: *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*.
16. Freund, S. N. and J. C. Mitchell: 1999, 'Specification and verification of Java bytecode subroutines and exceptions'. Technical report, Stanford University.
17. Goldberg, A.: 1998, 'A Specification of Java Loading and Bytecode Verification'. In: *Proc. 5th ACM Conf. Computer and Communications Security*.
18. Hagiya, M. and A. Tozawa: 1998, 'On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines'. In: G. Levi (ed.): *Static Analysis (SAS'98)*, Vol. 1503 of *Lect. Notes in Comp. Sci.* pp. 17–32.
19. Isabelle: 2002, 'Isabelle home page'. `http://isabelle.in.tum.de/`.
20. Kildall, G. A.: 1973, 'A Unified Approach to Global Program Optimization'. In: *Proc. ACM Symp. Principles of Programming Languages*. pp. 194–206.
21. Klein, G.: 2003, 'Verified Java Bytecode Verification'. Ph.D. thesis, Institut für Informatik, Technische Universität München.
22. Klein, G. and T. Nipkow: 2001, 'Verified Lightweight Bytecode Verification'. *Concurrency and Computation: Practice and Experience* **13**(13), 1133–1151. Invited contribution to special issue on Formal Techniques for Java.
23. Klein, G. and T. Nipkow: 2002, 'Verified Bytecode Verifiers'. *Theoretical Computer Science*. to appear.
24. Leroy, X.: 2001, 'Java bytecode verification: an overview'. In: G. Berry, H. Comon, and A. Finkel (eds.): *Computer Aided Verification, CAV 2001*, Vol. 2102 of *Lecture Notes in Computer Science*. pp. 265–285.
25. Leroy, X.: 2002, 'Bytecode Verification for Java Smart Card'. *Software Practice & Experience* **32**, 319–340.

26. Lindholm, T. and F. Yellin: 1996, *The Java Virtual Machine Specification*. Addison-Wesley.
27. Muchnick, S. S.: 1997, *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
28. Nipkow, T.: 2001, 'Verified Bytecode Verifiers'. In: F. Honsell (ed.): *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, Vol. 2030 of *Lect. Notes in Comp. Sci.* pp. 347–363.
29. Nipkow, T. and D. v. Oheimb: 1998, 'Java$_{light}$ is Type-Safe — Definitely'. In: *Proc. 25th ACM Symp. Principles of Programming Languages*. pp. 161–170.
30. Nipkow, T., D. v. Oheimb, and C. Pusch: 2000, 'μJava: Embedding a Programming Language in a Theorem Prover'. In: F. Bauer and R. Steinbrüggen (eds.): *Foundations of Secure Computation*. pp. 117–144.
31. Nipkow, T., L. C. Paulson, and M. Wenzel: 2002, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *Lect. Notes in Comp. Sci.* Springer.
32. O'Callahn, R.: 1999, 'A simple, comprehensive type system for Java bytecode subroutines'. In: *Proc. 26th ACM Symp. Principles of Programming Languages*. pp. 70–78.
33. Posegga, J. and H. Vogt: 1998, 'Java Bytecode Verification using Model Checking'. In: *OOPSLA'98 Workshop Formal Underpinnings of Java*.
34. Pusch, C.: 1999, 'Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL'. In: W. Cleaveland (ed.): *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, Vol. 1579 of *Lect. Notes in Comp. Sci.* pp. 89–103.
35. Qian, Z.: 1999, 'A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines'. In: J. Alves-Foss (ed.): *Formal Syntax and Semantics of Java*, Vol. 1523 of *Lect. Notes in Comp. Sci.* Springer-Verlag, pp. 271–311.
36. Qian, Z.: 2000, 'Standard fixpoint iteration for Java bytecode verification'. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **22**(4), 638–672.
37. Rose, E. and K. Rose: 1998, 'Lightweight Bytecode Verification'. In: *OOPSLA'98 Workshop Formal Underpinnings of Java*.
38. Sirer, E. G., S. McDirmid, and B. Bershad: 1997, 'Kimera: A Java system security architecture.'. Technical report, University of Washington.
39. Stärk, R. and J. Schmid: 2001, 'Java bytecode verification is not possible'. In: R. Moreno-Díaz and A. Quesada-Arencibia (eds.): *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*. pp. 232–234.
40. Stärk, R., J. Schmid, and E. Börger: 2001, *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer.
41. Stata, R. and M. Abadi: 1998, 'A type system for Java bytecode subroutines'. In: *Proc. 25th ACM Symp. Principles of Programming Languages*. pp. 149–161.
42. Sun Microsystems: 2000, 'Connected, Limited Device Configuration. Specification Version 1.0'. `http://java.sun.com/aboutJava/communityprocess/final/jsr030/`.
43. Verificard: 2002, 'Verificard Project Website in Munich'. `http://isabelle.in.tum.de/verificard/`.
44. Wildmoser, M.: 2002, 'Subroutines and Java Bytecode Verification'. Master's thesis, Technische Universität München.