Verified Java Bytecode Verification

Gerwin Klein

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Verified Java Bytecode Verification

Gerwin Klein

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Alois Knoll

Prüfer der Dissertation:
1. Univ.-Prof. Tobias Nipkow, Ph.D.
2. Univ.-Prof. David Basin, Ph.D.
   Eidgenössische Technische Hochschule
   Zürich/Schweiz

Die Dissertation wurde am 28. November 2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 7. Februar 2003 angenommen.

Kurzfassung

Der Bytecode Verifier ist ein essentieller Bestandteil der Sicherheitsarchiktektur der Programmierplattform Java. Die Dissertation präsentiert eine formale, ausführbare Spezifikation des Bytecode Verifiers sowie den Beweis, dass dieser korrekt ist. Die Formalisierung im Theorembeweiser Isabelle besteht aus einem abstrakten Framework für Bytecode-Verifikation, das mit zunehmend ausdrucksstarken Typsystemen instantiiert wird. Diese decken sämtliche interessanten Eigenschaften der Java-Plattform ab: Klassen, Objekte, virtuelle Methoden, Vererbung, Ausnahmebehandlung, Konstruktoren, Objekt-Initialisierung, Subroutinen und Felder. Die Formalisierung liefert zwei ausführbare verifizierte Bytecode Verifier: den iterativen Standard-Algorithmus sowie einen Lightweight Bytecode Verifier für Geräte mit eingeschränkten Ressourcen.

Abstract

The bytecode verifier is an important part of Java's security architecture. This thesis presents a fully formal, executable, and machine checked specification of a representative subset of the Java Virtual Machine and its bytecode verifier together with a proof that the bytecode verifier is safe.

The specification consists of an abstract framework for bytecode verification which is instantiated step by step with increasingly expressive type systems covering all of the interesting and complex properties of Java bytecode verification: classes, objects, inheritance, virtual methods, exception handling, constructors, object initialization, bytecode subroutines, and arrays.

The instantiation yields two executable verified bytecode verifiers: the iterative data flow algorithm of the standard Java platform and also a lightweight bytecode verifier for resource-constrained devices such as smart cards.

All specifications and proofs have been carried out in the interactive theorem prover Isabelle/HOL. Large parts of the proofs are written in the human-readable proof language Isabelle/Isar making it possible to understand and reproduce the reasoning independently of the theorem prover. All formal proofs in this thesis are machine checked and generated directly from Isabelle sources.

Acknowledgements

I thank Tobias Nipkow for supervising this thesis. His support and advice have been invaluable. I thank Gilles Barthe and David Basin for acting as referees.

I am indebted to Norbert Schirmer, Sebastian Skalberg, and Martin Strecker for reading and commenting on drafts of this thesis. I thank my wife, Elisabeth Meister, not only for proof reading. The remaining errors are mine, and they exist because I have ignored good advice.

I also thank the other members of the Isabelle team in Munich, current and former, for an excellent, friendly, and relaxed working atmosphere: Clemens Ballarin, Gertrud Bauer, Stefan Berghofer, David von Oheimb, Farhad Mehta, Leonor Prensa-Nieto, Cornelia Pusch, Markus Wenzel, and Martin Wildmoser.

Contents

1 Introduction

1.1 Motivation

The bytecode verifier is an important, integral part of Java's security architecture. It has become popular to download and execute untrusted web applets and JavaCard applets often even without the user's approval or intervention. Contrary to the approach of a certain widely used operating platform, Java provides a scheme to execute untrusted code safely: the *sandbox*. The sandbox is an insulation layer that implements a control policy preventing unsafe access to hardware and operating system resources. It relies on three security measures:

- Java is not compiled into machine code, but rather into an intermediate format for a virtual machine [51].
- Hardware access is mediated by a set of API classes [30] that implement a suitable access control policy.
- Before execution, the bytecode is statically verified [51] to ensure that it does not bypass the two security measures above.

The last of the three components is the bytecode verifier (BV). It is a central component of the architecture: any bug in the BV causing an unsafe applet to be accepted potentially renders the sandbox useless. McGraw and Felten [52, 53] give a list of examples. On the other hand, the BV is itself a large and complex program that performs an elaborate data flow analysis. It is thus mandatory, but highly nontrivial, to ensure its correctness.

This thesis presents a fully formal, executable, and machine checked specification of a representative subset of the Java Virtual Machine (JVM) and its bytecode verifier together with a proof that the bytecode verifier is correct.

A large body of literature has evolved around the JVM and its BV. This research has made clear that the BV is an interesting candidate for a formal development; it has made apparent that looking at different properties of the bytecode language in isolation is not enough, as it is the interactions between them that may lead to unsafe behaviour; and it has shown that the BV is inherently complex.

Apart from Freund [26], the literature about the BV either restricts itself to an isolated feature of the BV, or only shows vague proof sketches. This is not the fault of the respective authors: the sheer size of the formalizations involved makes it almost impossible to present a fully detailed, comprehensive proof of soundness and to expect a human reader to be able to understand and reproduce the proof, let alone to assure its correctness. Unsurprisingly, Freund's PhD thesis [26], which presents such a proof in reasonable detail, contains precisely the kind of slight errors (see also Section 4.5.1) that are to be expected for a complex formal development of this size on paper alone. The formalization in [26] is an exceptional piece of work, and it is most probably sound, but an—if only ever so slightly—incorrect proof defeats at least some of the purpose of strict formality. How many more of the sub proofs are incomplete? Has another small but important assumption been overlooked? Without redoing and manually checking the 103 pages of formal proof (not including the specification), it is not possible to answer these questions.

This is where the theorem prover comes in. Formalizing the BV in Isabelle does not make the complexity of the problem disappear, but it has three important advantages:

Correctness Most obviously, proofs are checked mechanically, and trust in them is improved significantly. If a proof does not happen to trigger a soundness bug in the theorem prover, it is undeniably correct. Isabelle is an LCF style [31] theorem prover that isolates the soundness critical part in a small proof kernel. Hence, soundness bugs are extremely rare.

Validation The specification is executable and can be validated against existing implementations of the BV. With Isabelle's document generation capabilities, the second approach to validation remains available, too: the Isabelle specification can be read conveniently and can be compared to the official JVM specification.

Readability Ensuring correctness of a statement is not the only purpose of a proof; often, it is even more important that the proof also leads to a deeper understanding of the problem. This property of formal proofs is lost in most theorem provers. The Isabelle/Isar language used in large parts of this formalization retains the full formality of the theorem prover and at the same time makes the reasoning accessible for humans.

The next section (1.2) presents the contributions of this thesis. Sections 1.3 and 1.4 give an informal introduction to the JVM and the BV. Section 1.5 surveys the literature on bytecode verification and gives pointers to related work. Section 1.6 contains an introduction to Isabelle notation and Section 1.7 provides an overview of the remainder of this thesis.

1.2 Contributions

The focus of this thesis is the bytecode verifier—the algorithm as well as the properties of the Java bytecode language it is concerned with. Of these, I present classes, objects, inheritance, virtual methods, exception handling, constructors, object initialization, bytecode subroutines, and arrays in this thesis. The specification yields two executable verified bytecode verifiers: the iterative data flow algorithm of the standard Java platform and also a lightweight bytecode verifier for resource-constrained devices.

The main contributions of this thesis are the following (Chapter 7 contains a more detailed discussion).

- The formalization of the BV in this thesis is one of the most comprehensive formalizations of the BV that have been published, and it is the most comprehensive one in a theorem prover. It shows that the type system used in the JVM and the BV is safe and can be proved correct in a theorem prover.
- The abstract typing framework in this thesis makes it possible to cleanly distinguish between executable algorithm and type system. It enables a uniform treatment of verification algorithms, which leads to a lightweight bytecode verifier that handles properties beyond both the original version by Rose [74, 75] and the industrial version by Sun Microsystems [87, 88]. It is important to note that these verification algorithms are not only executable in theory, but that ML prototypes have been generated from the specification.
- This thesis studies multiple type systems. I discuss the merging type system that is used in current BV implementations, and the more recent set based type system that is especially useful for bytecode subroutines. For the merging type system with object initialization I show the first proof of correctness in a theorem prover that considers a representative subset of the JVM. For the set based type system I also present the first proof of correctness in a theorem prover, and I show that the approach scales to a representative, object oriented subset of the JVM. The original version by Coglio [15, 16] did not even contain classes.

The focus of this thesis *is not* a complete model of the JVM that conforms precisely to the JVM specification [51] down to every technical detail. Such a model in a theorem prover, of course, has its merits, but as in every formal development, a balance between abstraction and detail has to be found. Since the focus of this thesis is the BV, the model of the JVM I present here abstracts as far as possible from everything that does not concern bytecode verification.

The about 200 instructions in the JVM are condensed into 22 instructions in the last stage of the formalization. For instance, the 41 arithmetic instructions of Java bytecode all behave the same way in the BV and hence are modeled by one representative instruction in the formalization. The formalization contains one representative instruction of each class that is interesting for the BV. The subset of the Java language in this thesis is called μJava, its virtual machine μJVM.

Unlike Java, the μJava bytecode language does not contain interfaces, access modifiers, packages, threads, class initializers, static methods, class loading, wide instructions, or wide data types. Most of these are only important for the JVM, not for the BV. Section 7.2 discusses those that concern bytecode verification in more detail and gives pointers on how to include them in the formalization.

1.3 The Java Virtual Machine

As in the Java source language, bytecode programs are organized in classes and methods. The JVM is a stack based abstract machine for bytecode programs. 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 1.1 illustrates, each frame contains its own program counter, operand stack, and register set. Bytecode programs specify the number of stack and register slots they use; this allows an implementation to allocate an activation record of the correct size for each method invocation.

Bytecode instructions manipulate either the heap, the registers, or the operand stack. For example, the *IAdd* instruction removes the topmost two values (integers) from the operand stack, adds them, and pushes the result back onto the stack. In the example in Figure 1.1, the JVM would execute the *Getfield F A* instruction, removing the reference to the object from the stack, and putting the value (*Addr 9*) of the field *F* of the referenced object (at *Addr 8*) on top.

Apart from the operand stack, the JVM uses registers to store the working data and arguments of the method. The first register (number *0*) is reserved for the *this* pointer of the method. The next p registers are reserved for the p parameters of the method, and the rest is usually dedicated to local variables declared inside the method.

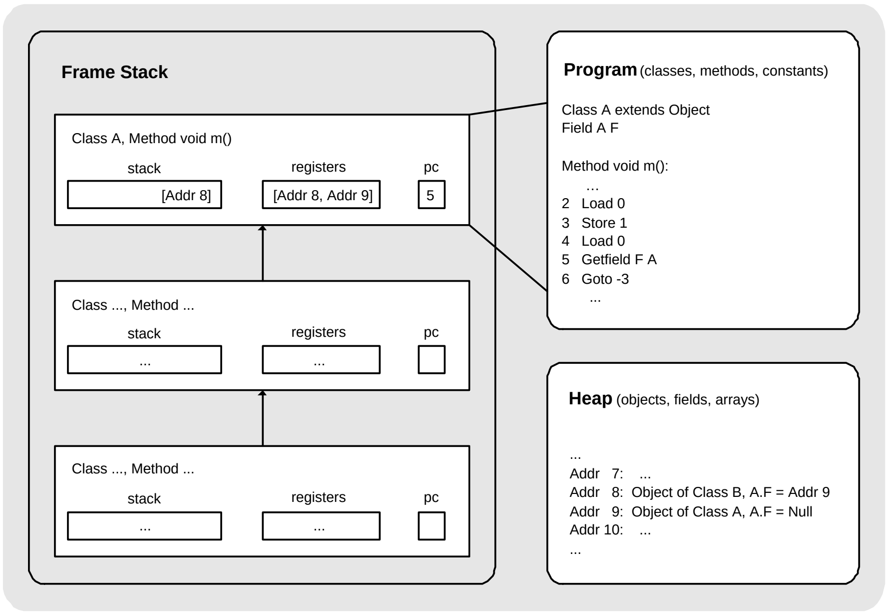

Figure 1.1: The JVM.

The heap stores dynamically created objects, while the operand stack and registers only contain references to objects.

Exception handlers for each method are specified in a table of tuples (s,e,h,C). If an exception of class E is raised by an instruction in the asymmetric interval $[s,e)$ and E is a subclass of C, the table entry is said to **match** the exception. The JVM looks for the first table entry that matches E, and transfers control to the instruction at h, the **exception handler**.

The instructions in the JVM are typed. This means that, for example, the *IAdd* instruction only works on integers, not addresses. Similarly, a *Getfield F A* instruction that accesses the field F of an object on the heap only works on references of the correct class (any subclass of A). Using the *Getfield* or *Invoke* instructions on an integer would be an attempt to forge an object reference.

1.4 The Bytecode Verifier

The JVM relies on the following assumptions for executing bytecode:

Correct types All bytecode instructions are provided with arguments of the type they

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

Figure 1.2: Example of a method welltyping.

expect on operand stack, registers, and heap.

No overflow and underflow No instruction tries to retrieve a value from an empty stack, no instruction tries to put more elements on the stack than statically specified in the method, and no instruction accesses more registers than statically specified in the method.

Code containment The program counter never leaves the code array of the method. Specifically, it must not fall off the end of the method's code or branch into the middle of an instruction encoding.

Initialized registers All registers apart from the *this* pointer and the method parameters must be written to before they are first read. This corresponds to the *definite assignment* requirement for local variables on the source level.

Initialized objects Before fields or methods of an object can be accessed, its constructor must be called. Each constructor in turn must first call the superclass constructor before it accesses fields and methods of the object.

It is the purpose of the bytecode verifier to ensure statically that these assumptions are met at any time.

Bytecode verification is an abstract interpretation of bytecode methods: instead of values, we only consider their types. The BV can be viewed as a finite state machine working on **state types**. A state type characterizes a set of runtime states by giving type information for the operand stack and registers. For example, the first state type in Figure 1.2 ([],[*Class B*, *Integer*]) 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. A method is called **welltyped** if we can assign a welltyping to each instruction. A state type (*st*,*lt*) is a welltyping 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's look at an example. Figure 1.2 shows the instructions on the left and the type of stack elements and registers on the right. The **method type** is the full 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. The *Some* before each of the entries means that it was possible to predict some type for each of the instructions. If one of the instructions had been unreachable, the type entry would have been *None*. 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 holding a reference to an object of class B and an integer. The first instruction loads register *0*, a reference to a B object, on the stack. The type information associated with the following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that usage of 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 *Integer* or *Class A*, which are incompatible: 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 1.2 was an example for a welltyped method: we were able to find a welltyping. 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 a welltyping.

1.5 Related Work

This section provides an overview of the literature on bytecode verification and gives pointers to work related to this thesis.

Most closely related are other formalizations of the JVM or the BV in theorem provers:

- Barthe et al. [5, 6] employ the Coq system [23] for proofs about the JavaCard [54] virtual machine and its BV. They formalize the full JavaCard bytecode language, but have only a simplified treatment of subroutines. In [2, 3, 4], they show how to increase automation in the process of specifying a defensive machine [22] (with safety checks), an aggressive machine (without safety checks), and an abstract machine (on the type level), together with their proofs of correspondence.

- Bertot [10] also uses the Coq system to prove the correctness of a bytecode verifier based on the type system by Freund and Mitchell [27]. He focuses on object initialization only.

- Posegga and Vogt [67] 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, and Gawkowski [7] use Isabelle/HOL, μJava, and the abstract BV framework, of which I present an extended version here, to prove the model checking approach correct.

- The formalizations in this thesis are part of the work on the Java language of the Isabelle team in Munich, mainly in the projects Bali [1] and VerifiCard [89]. The specification of the BV is based on groundwork by Nipkow [58] and Pusch [68]. Nipkow, von Oheimb, and Schirmer [60, 62, 64, 65, 66, 76, 90, 91] have formalized the Java source language in Isabelle. Strecker [40, 85, 86] has proved correct a compiler for μJava from source to bytecode language in Isabelle, and has also shown that all welltyped programs of the source language are accepted by the bytecode verifier. Earlier, restricted forms of my formalization for standard and lightweight bytecode verification have appeared in [37, 38, 39].

The following projects use tool support to specify or implement bytecode verification:

- Working towards a verified implementation in SPECWARE, Qian, Goldberg and Coglio have specified and analyzed large portions of the bytecode verifier [18, 19]. Goldberg [29] rephrases and generalizes the overly concrete description of the BV given in the JVM specification [51] as an instance of a generic data flow framework. Qian [69] specifies the BV as a set of typing rules, a subset of which was proved correct formally by Pusch [68]. Qian [70] also proves the correctness of an algorithm for turning his type checking rules into a data flow analyzer. However, his algorithm is still quite abstract.

- The Kimera project [80] treats bytecode verification in an empirical manner. Its aim is to check bytecode verifiers by automated testing.

- Casset et al. [12, 13, 14, 72] use the B method to specify a bytecode verifier for a defensive JavaCard VM, which they then refine into an executable program that provably satisfies the specification. They focus on the scalability of the B method for such proofs. Due to its commercial environment, the full specification itself does not seem to be publicly available. Hence, it is difficult to judge what exactly they have proved, and which parts of the bytecode language and its properties their formalization contains. The most recent article in the series, by Requet [72], sheds some light on this. Although they claim to handle a large subset of the JavaCard VM and over one hundred instructions, he writes [72, p. 288]:

  > As the aim of this work was to verify the scalability of the approach, instructions that would drastically increase the complexity of the model have been left out. Especially, those instructions include the instructions used for subroutines, for method calls and for objects handling.

- Stärk et al. [81, 82, 83] use Java and the JVM as a case study for abstract state machines. They formalize the process from compilation of Java programs down to bytecode verification, and also provide an executable version in ASM Gofer [77]. Their main theorem says that the bytecode verifier accepts all bytecode programs the compiler generates from valid Java sources. Proofs, however, are by pen and paper. They argue that this theorem does not hold for the full Java language. Therefore they introduce a stronger constraint for definite assignment than the JVM specification. The type system presented in Chapter 5 of this thesis makes this restriction unnecessary.

The following publications present type systems for the JVM; Hartel and Moreau [34] as well as Leroy [47, 50] provide a more detailed overview and discussion, Wildmoser [93] concentrates on articles related to bytecode subroutines.

- Stata and Abadi [84] were the first to specify a type system for a subset of Java bytecode that supports subroutines. The typing rules they use are clearer and more precise than the JVM specification, but they accept fewer safe programs.

- Freund and Mitchell [27, 28] develop typing rules for increasingly large subsets of the JVM, including exception handling, object initialization, and subroutines. Freund surveys the costs and benefits of subroutines [25], and reaches the conclusion that they should have been left out of the bytecode language. The final formalization [26] considers an instruction set comparable to the one presented here. The formalization, especially the treatment of subroutines, is more complex and more restrictive than the one in this thesis. The proof of type safety for object initialization in Chapter 4 is based on the one in [26].

- Leroy [47, 50] gives a very good overview on bytecode verification, and proposes a polyvariant data flow analysis in the bytecode verifier to solve the subroutine problem. He also addresses the problem of on-card bytecode verification for Java smart cards by program transformation combined with a simplified BV [48, 49].

- Coglio [15, 16] and, independently, Brisset [11] provide a simple solution to the subroutine problem in bytecode verification that is akin to model checking. Chapter 5 uses this scheme as basis for the formalization of subroutines in μJava. Together with Qian and Golberg, Coglio formally specifies dynamic class loading [17, 21, 71]. Coglio also gives an overview of the description of the bytecode verifier in the JVM specification [51] and suggests several improvements [20].

- Hagiya and Tozawa [33] use indirect types in rules similar to those of Stata and Abadi [84] to tackle subroutines. These indirect types are of the form $\mathit{last}(x)$, denoting the type register x held before the subroutine. This avoids the loss of precision type merges induce for unused registers.

- O'Callahan [63] uses type variables and continuations to handle subroutines. Although his approach accepts a large portion of type safe programs—even recursive subroutines could be supported—it remains unclear whether it can be realized efficiently.

- Rose [73, 74, 75] presents a lightweight bytecode verification scheme that works similar to Necula's proof carrying code [56]: the program is annotated with typing information which is checked by a simplified on-card verifier. In Section 2.5, I formalize a more general version of this algorithm and prove it correct.

- Laneve and Bigliardi [44, 45, 46] have implemented a bytecode verifier that checks proper handling of thread monitors.

- Knoblock and Rehof [42, 43] show how to turn the type system into a lattice even if it contains interfaces.

- Yelland [94] reduces bytecode verification to Haskell type inference.

1.6 Isabelle

This section gives a short and not so gentle introduction to Isabelle. It is by no means comprehensive, but it introduces the Isabelle/HOL notation that is used in this thesis. For a gentler, deeper, and eminently readable introduction, I recommend [61].

Isabelle [35] is a generic, interactive theorem prover. It is generic in the sense that it can be instantiated with different object logics. The most widely used of these object logics is Isabelle/HOL, simply typed higher order logic. Formalizations are organized in an acyclic graph of *theories*, each containing a set of declarations, definitions, and theorems. For the most part, the notation is the same as in standard mathematics and functional programming.

Function application is written in curried style as in functional programming, so $f\ a$ is the function f applied to the argument a.

The notation for set comprehension deviates from standard mathematics: $\{x.\ P\ x\}$ is the set of all x for which $P\ x$ holds. The common $\{y.\ \exists\, x.\ f\, x = y \wedge P\ x\}$, in mathematics written as $\{f\,x \mid P\ x\}$, is abbreviated by $\{f\ x\ |x.\ P\ x\}$ in Isabelle/HOL. The following, for instance, defines the image of a set A under a function f:

$$f\ `\ A \equiv \{f\ x\ |x.\ x \in A\}$$

Function update is written $f\ \ (x := y)$. The formal definition uses λ-abstraction and an *if-then-else* expression.

$$f\ \ (x := y) \equiv \lambda x'.\ \mathit{if}\ x' = x\ \mathit{then}\ y\ \mathit{else}\ f\ x'$$

The equivalence sign $\equiv$ is used for definitions that are true abbreviations. Recursive definitions are formulated with simple equality $=$.

New data types can be introduced with the **datatype** keyword, simple type abbreviations use **types**. Examples are:

$$\begin{aligned} \textbf{types}\ \mathit{nat\text{-}pair} &= \mathit{nat} \times \mathit{nat} \\ \textbf{datatype}\ \alpha\ \mathit{list} &= \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\alpha\ \mathit{list}) \end{aligned}$$

The first line declares *nat-pair* to be the Cartesian product of *nat* and *nat* (the type of natural numbers). The second line declares the polymorphic data type of lists. The type constructor *list* takes the type variable α as argument (written in prefix as α *list*). The declaration says that a list on type α is either *Nil*, or a *Cons* with an element of type α as head and a list on type α as tail. Isabelle provides special syntax for *Nil* and *Cons*: the empty list is [], and $x\#xs$ stands for *Cons x xs*.

Isabelle/HOL has a rich library of list functions: $xs!n$ is the n-th element of the list xs, the operator @ is append, the notation $[1..n(]$ is the list of natural numbers from 1 to $n{-}1$, and $xs\ [n{:=}\ x]$ sets the n-th element of xs to x. Apart from these, I will use functions known from functional programming:

$$\begin{aligned}
&\textit{size} &&:: \alpha\ \textit{list} \Rightarrow \textit{nat}\\
&\textit{rev} &&:: \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}\\
&\textit{take} &&:: \textit{nat} \Rightarrow \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}\\
&\textit{drop} &&:: \textit{nat} \Rightarrow \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}\\
&\textit{zip} &&:: \alpha\ \textit{list} \Rightarrow \beta\ \textit{list} \Rightarrow (\alpha \times \beta)\ \textit{list}\\
&\textit{map} &&:: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ \textit{list} \Rightarrow \beta\ \textit{list}\\
&\textit{filter} &&:: (\alpha \Rightarrow \textit{bool}) \Rightarrow \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}
\end{aligned}$$

The *filter* function has special syntax: $[x{\in}xs.\ P\ x]$ is short for *filter* $(\lambda x.\ P\ x)\ xs$. There are also functions particular to Isabelle: *set* transforms a list into a set, *list-all2* is an executable universal quantifier on a pair of lists, satisfying

$$\textit{list-all2}\ P\ xs\ ys = ((\forall (x,y) \in \textit{set}\ (\textit{zip}\ xs\ ys).\ P\ x\ y) \wedge \textit{size}\ xs = \textit{size}\ ys)$$

HOL is a logic of total functions. For modeling partial functions, the *option* datatype is useful:

$$\textbf{datatype}\ \alpha\ \textit{option} = \textit{None} \mid \textit{Some}\ \alpha$$

A function $f :: \alpha \Rightarrow \beta$ *option* then returns *Some* x for defined results x, and *None* if it is undefined for an argument value. A function $f :: \alpha \Rightarrow \beta$ *option* is also called a **map** from α to β. Function update for maps has special syntax: $f\ (x \mapsto y)$ is short for $f\ (x := \textit{Some}\ y)$.

Isabelle/HOL also knows Hilbert's classical choice operator. The term $\textit{SOME}\ x.\ P\ x$ returns some x that satisfies P. As all functions in HOL are total, it also returns a value if no such x exists. In this case, nothing is known about this value.

The formal proofs in this thesis use the Isabelle/Isar proof language [59, 92]. Isabelle/Isar proofs are fully formal and machine checked, but contrary to the proof script style usually found in theorem provers, the resulting proofs are readable for humans.

I will neither introduce nor define Isabelle/Isar formally here, but rather give an example that should explain how to read the Isabelle proofs in this thesis.

```
lemma example:
  assumes pq: ∃x. P x ∧ Q x
  shows (∃x. P x) ∧ (∃x. Q x)
proof −
  from pq obtain y where p: P y and q: Q y by auto
  from p have ∃x. P x ..
  moreover
  from q have ∃x. Q x ..
```

```
    ultimately
    show ?thesis ..
  qed
```

The example above is formulated in an especially verbose way to demonstrate as many language constructs as possible. The first three lines say that the proposition to be proved is $(\exists\, x.\ P\ x \wedge Q\ x) \longrightarrow (\exists\, x.\ P\ x) \wedge (\exists\, x.\ Q\ x)$. The resulting lemma is stored under the name *example*. Proofs in Isar can either be compound (**proof** ... **qed**), or simple one-liners (**by** ...). Intermediate, named facts in a proof can be established with the **have** and **obtain** commands.

The proof above begins with the technically complex, but conceptually simple **obtain** command: from the assumption *pq*, we can get a witness y such that $P\ y$ and $Q\ y$ holds. The proof uses the *auto* proof method to show that this is actually true. From these two facts we can then conclude that both $\exists\, x.\ P\ x$ and $\exists\, x.\ Q\ x$ are true. The abbreviation .. indicates a trivial proof that only needs the application of one single rule. The **moreover** command is used to collect facts (here provided by **have**), while **ultimately** makes the collected facts available for another subproof. Below, on the left hand side is a proof fragment with **moreover**, and on the right side an equivalent one without.

```
have P₁ ...                  have fact1: P₁ ...
moreover have P₂ ...         have fact2: P₂ ...
moreover have P₃ ...         have fact3: P₃ ...
ultimately have P₄ ...       from fact1 fact2 fact3 have P₄ ...
```

The last command in the proof body (**show** *?thesis*) solves the pending goal with a trivial one-rule proof. The term abbreviation *?thesis* refers to the stated goal in the current proof block. In this case, *?thesis* is $(\exists\, x.\ P\ x) \wedge (\exists\, x.\ Q\ x)$. Term bindings like these can also be used in a pattern matching style: the fragment $f\ (x{+}1) = y$ (*is f ?z = -*) binds *?z* to $x{+}1$.

Another concept of Isabelle that I will use extensively in this thesis is that of **locales**. A locale in Isabelle is a collection of constants, assumptions, and definitions. It is a useful tool for structuring theorems in a large development, because it defines a common context for a group of theorems. An example is:

```
locale A =                          locale B = A +
  fixes xs :: α list                  fixes ys :: α list
  assumes p: P xs                     assumes q: Q xs ys
  defines g ≡ λf. map f xs
```

The locale A above fixes a constant xs (a list of type α) for which $P\ xs$ holds. It also defines an abbreviation g for map applied to xs. The locale B extends the context that A has built up by another constant ys, for which it assumes $Q\ xs\ ys$. The current version of Isabelle only allows proper abbreviations (using **defines**) in locales, no recursive function definitions (using, for instance, multiple equations). With a small trick[1], it is still possible to use recursive functions as if they were defined in the context of a locale. For the presentation in this thesis, I will therefore pretend that this technical limitation does not exist.

1.7 Overview

The formalization of bytecode verification for μJava rests on the abstract framework introduced in Chapter 2. It contains the lattice-theoretic concepts for the framework, an abstract definition of welltypings that builds on a semilattice and a transfer function, and two equally abstract definitions of executable algorithms for bytecode verification: the standard iterative data flow analysis and a lightweight bytecode verifier for resource-constrained devices. The remaining chapters instantiate this framework step by step with increasingly expressive type systems.

Chapter 3 contains a first simple type system for classes, objects, inheritance, virtual methods, and exception handling. As in the following chapters, the instantiation results in two executable bytecode verifiers for μJava: Kildall's algorithm and the lightweight bytecode verifier. Chapter 3 also describes in detail the formalization of the μJVM and the proof of correctness for both bytecode verifiers.

Chapter 4 extends the type system of Chapter 3 by constructors and object initialization. This feature of the bytecode verifier ensures that all objects are properly initialized before they are used.

Chapter 5 in turn extends the formalization of Chapter 4 by bytecode subroutines. This entails a substantial change in the type system. It is the first formalization in a theorem prover of a type system for Java that contains bytecode subroutines together with object initialization and exception handling.

Chapter 6 adds arrays to the language.

Chapter 7 concludes with a summary and pointers to further work.

[1]First define the recursive function outside the locale context, then define an abbreviation of this function inside the locale, and finally derive the defining equations in the locale as theorems.

2 An Abstract Framework

The formalizations of bytecode verification in this thesis are instances of the abstract typing framework in this chapter. The framework takes a semilattice and a transfer function as parameters and yields a description of welltypings, an executable, verified version of Kildall's algorithm, and an executable, verified lightweight bytecode verifier for resource-constrained devices.

2.1 Introduction

This chapter presents an abstract framework for bytecode verification. It builds on the work by Nipkow [58]. Compared to [58] and further work by Nipkow and myself [39], it is more general and more flexible in that it can be instantiated with more type systems.

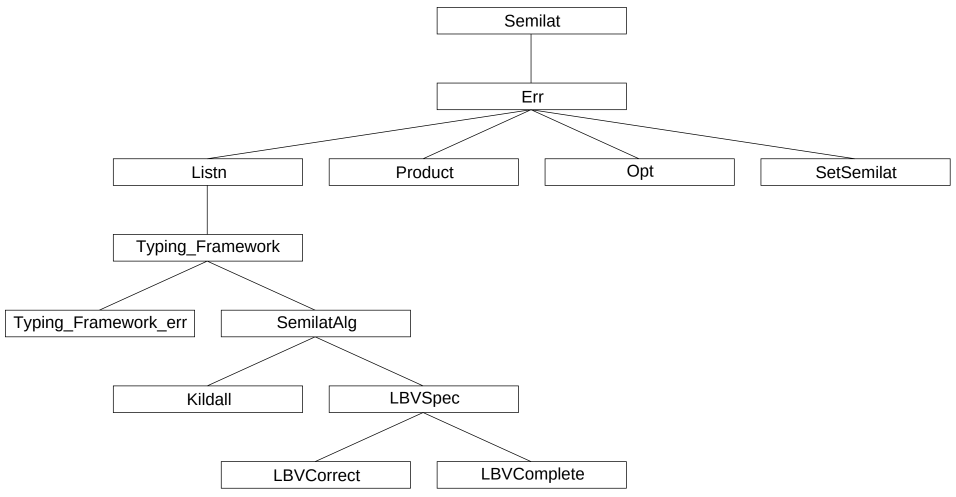

Figure 2.1: Abstract framework overview.

Figure 2.1 gives an overview of the Isabelle theories that the abstract framework comprises. Section 2.2 describes the lattice-theoretic concepts of the framework, the upper three levels of Figure 2.1. Section 2.3 brings the definition of welltypings (theory *Typing-Framework*), constraints on the transfer function (theory *SemilatAlg*), and a refinement of the transfer function (theory *Typing-Framework-Err*). This part is more general than [39, 58]. The last two sections present two algorithms for bytecode verification: Section 2.4 shows the formalization of Kildall's algorithm, Section 2.5 the lightweight bytecode verifier. Both algorithms can be instantiated with different type systems, are executable, and verified in Isabelle/HOL.

2.2 Semilattices

This section introduces the formalization of the basic lattice-theoretic concepts required for data flow analysis and its application to the JVM. Since most of this already appeared in [39, 58], I here only reproduce the definitions and main properties without proof.

2.2.1 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 $x \le_r y = r\ x\ y$ and $x <_r y = (x \le_r y \wedge x \neq y)$, $r :: \alpha\ \mathit{ord}$ is by definition a **partial order** iff the predicate $\mathit{order} :: \alpha\ \mathit{ord} \Rightarrow \mathit{bool}$ holds for r:

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

A partial order r satisfies the **ascending chain condition** on A if there is no infinite ascending chain $x_0 <_r x_1 <_r \ldots$ in A; $\top$ is a **top element** if $x \le_r \top$ for all x, and $\bot$ a **bottom element** if $\bot \le_r x$ for all x:

$$\begin{aligned}&\mathit{acc} :: \alpha\ \mathit{ord} \Rightarrow \mathit{bool}\\ &\mathit{acc}\ r \equiv \mathit{wf}\ \{(y,x).\ x{\in}A \wedge y{\in}A \wedge x <_r y\}\end{aligned}$$

$$\begin{array}{ll}\mathit{top} :: \alpha\ \mathit{ord} \Rightarrow \alpha \Rightarrow \mathit{bool} & \mathit{bottom} :: \alpha\ \mathit{ord} \Rightarrow \alpha \Rightarrow \mathit{bool}\\ \mathit{top}\ r\ \top \equiv \forall x.\ x \le_r \top & \mathit{bottom}\ r\ \bot \equiv \forall x.\ \bot \le_r x\end{array}$$

2.2.2 Semilattices

Based on the supremum notation $x \sqcup_f y = f\ x\ y$ and the two type synonyms $\alpha\ \mathit{binop} = \alpha \Rightarrow \alpha \Rightarrow \alpha$ and $\alpha\ \mathit{sl} = \alpha\ \mathit{set} \times \alpha\ \mathit{ord} \times \alpha\ \mathit{binop}$, the tuple $(A,r,f) :: \alpha\ \mathit{sl}$ is by definition a **semilattice** iff the predicate $\mathit{semilat} :: \alpha\ \mathit{sl} \Rightarrow \mathit{bool}$ holds:

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

where $\mathit{closed}\ A\ f \equiv \forall x\ y \in A.\ x \sqcup_f y \in A$.

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

The following Isabelle locale is used below.

locale *semilat* =
  fixes $A :: \alpha\ \mathit{set}$ **and** $r :: \alpha\ \mathit{ord}$ **and** $f :: \alpha\ \mathit{binop}$
  assumes *semilat*: $\mathit{semilat}(A,r,f)$

The next sections look at a few data types and the corresponding semilattices which are required for the construction of the μJVM bytecode verifier. The definition of those semilattices follows a pattern: they lift an existing semilattice to a new semilattice with more structure. They extend the carrier set and define 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. Qualified names are of the form *Theoryname.localname*, and they apply to constant definitions and functions as well as type constructions. So *Err.sup* later on refers to the *sup* functional defined for the error type in Section 2.2.3.

2.2.3 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. It introduces both a data type and an equivalent construction on sets:

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

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

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

Lemma 2.1 If r is a partial order that satisfies the ascending chain condition, then *le* r also is a partial order that satisfies the ascending chain condition.

The following lifting functional is frequently useful:

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

This leads to the notion of an err-semilattice. It is a variation of a semilattice with top element. Because the behaviour of the ordering and the supremum on the top element is 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:

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

A tuple $L :: \alpha\ \mathit{esl}$ is by definition an **err-semilattice** iff *sl* L is a semilattice. Conversely, we get Lemma 2.2.

Lemma 2.2 *esl* L is an err-semilattice if L is a semilattice.

The supremum operation of $\mathit{sl}(\mathit{esl}\ L)$ is useful on its own:

$$\mathit{sup}\ f\ = \mathit{lift2}\ (\lambda x\ y.\ \mathit{OK}(x \sqcup_f\ y))$$

2.2.4 The Option Type

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

$$\textbf{datatype}\ \alpha\ \mathit{option} = \mathit{None} \mid \mathit{Some}\ \alpha \qquad \mathit{opt}\ A \equiv \{\mathit{None}\} \cup \{\mathit{Some}\ a\ |a.\ a \in A\}$$

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

$$
\begin{array}{llll@{\qquad}lll}
\mathit{le}\ r\ (\mathit{Some}\ x) & (\mathit{Some}\ y) & = x \le_r y & & \mathit{sup}\ f\ (\mathit{Some}\ x) & (\mathit{Some}\ y) & = \mathit{Some}(f\ x\ y) \\
\mathit{le}\ r\ \mathit{None} & \_ & = \mathit{True} & & \mathit{sup}\ f\ \mathit{None} & z & = z \\
\mathit{le}\ r\ (\mathit{Some}\ x) & \mathit{None} & = \mathit{False} & & \mathit{sup}\ f\ z & \mathit{None} & = z
\end{array}
$$

Lemma 2.3 $\mathit{sl}(A,r,f) = (\mathit{opt}\ A,\ \mathit{le}\ r,\ \mathit{sup}\ f)$ maps semilattices to semilattices.

Lemma 2.4 *le* preserves the ascending chain condition.

2.2.5 Products

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

$$
\begin{array}{l}
\mathit{esl} :: \alpha\ \mathit{esl} \Rightarrow \beta\ \mathit{esl} \Rightarrow (\alpha \times \beta)\ \mathit{esl} \\
\mathit{esl}\ (A,r_A,f_A)\ (B,r_B,f_B) = (A \times B,\ \mathit{le}\ r_A\ r_B,\ \mathit{sup}\ f_A\ f_B) \\[4pt]
\mathit{le} :: \alpha\ \mathit{ord} \Rightarrow \beta\ \mathit{ord} \Rightarrow (\alpha \times \beta)\ \mathit{ord} \\
\mathit{le}\ r_A\ r_B = \lambda(a_1,b_1)(a_2,b_2).\ a_1 \le_{r_A} a_2 \wedge b_1 \le_{r_B} b_2 \\[4pt]
\mathit{sup} :: \alpha\ \mathit{ebinop} \Rightarrow \beta\ \mathit{ebinop} \Rightarrow (\alpha \times \beta)\ \mathit{ebinop} \\
\mathit{sup}\ f\ g = \lambda(a_1,b_1)(a_2,b_2).\ \mathit{Err.sup}\ (\lambda x\ y.(x,y))\ (a_1 \sqcup_f a_2)\ (b_1 \sqcup_g b_2)
\end{array}
$$

Note that $\times$ is used both on the type and the set level.

Lemma 2.5 If both L_1 and L_2 are err-semilattices, so is $\mathit{esl}\ L_1\ L_2$,

Lemma 2.6 If both r_A and r_B satisfy the ascending chain condition, so does $\mathit{le}\ r_A\ r_B$.

2.2.6 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:

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

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

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

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. Below, I use the notation $xs \le[r]\ ys$ for $xs \le_{(le\ r)} ys$.

Lemma 2.7 If L is a semilattice, so is $sl\ n\ L$.

Lemma 2.8 If r is a partial order and satisfies the ascending chain condition, then $le\ r$ also is a partial order that satisfies the ascending chain condition.

In case we want to combine lists of different lengths, or if the supremum on the elements of the list may return *Err* (not to be confused with *Err.sup* the *sup* functional defined in Theory *Err*, Section 2.2.3), the following function is useful:

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

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

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

Lemma 2.9 If L is an err-semilattice, so is $uptoesl\ n\ L$.

2.2.7 Sets

Theory *SetSemilat* shows that finite sets form a semilattice.

The order is the usual subset relation $\subseteq$, and the supremum is union $\cup$. It is easy to see that $(Pow\ A, \subseteq, \cup)$ is a semilattice (where $Pow\ A$ is the power set of A). Unfortunately, the subset relation allows infinitely ascending chains, and hence violates the ascending chain condition, which is needed below. Even if we only take the finite subsets in $Pow\ A$, there may be infinitely ascending chains. For example, consider the following sets of natural numbers:

$$\{\} \subset \{0\} \subset \{0,1\} \subset \{0,1,2\} \subset \ldots$$

Each of these is finite, but the chain continues ad infinitum.

If the carrier set A itself is finite, however, $\subseteq$ does satisfy the ascending chain condition on *Pow* A:

Lemma 2.10 If A is finite, (*Pow* A, $\subseteq$, $\cup$) is a semilattice and $\subseteq$ satisfies the ascending chain condition on *Pow* A.

2.3 Stability

This section describes welltypings abstractly. The framework presented here is an extended, more general version of [39, 58]. I begin with the notion of welltyping in Section 2.3.1, continue with restrictions on the transfer function in Section 2.3.2, and conclude with a refinement of the transfer function in Section 2.3.3.

2.3.1 Welltypings

In this abstract setting, there is no need yet to talk about the instruction sequences themselves. They will be hidden inside a function that characterizes their behaviour. This function and a semilattice form the parameters of the model.

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. At this level, programs are sequences of instructions, and 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 2.2 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 (and for the *Ret* instruction necessary) that the structure of the data flow graph dynamically changes in

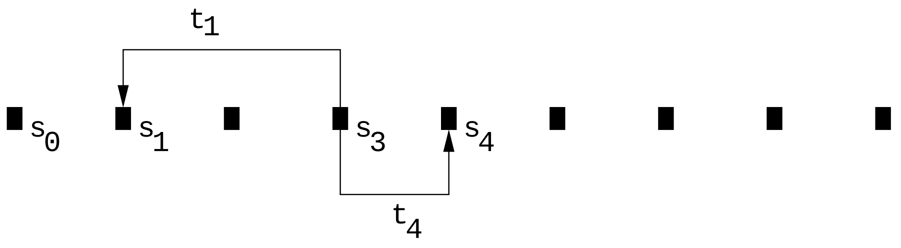

Figure 2.2: Data flow graph for *step* 3 $s_3 = [(1,t_1),(4,t_4)]$.

the iteration process of the analysis. It may not change freely, however. Section 2.3.2 will introduce certain constraints on the *step* function that the analysis needs in order to succeed.

The two definitions below are in the following context.

$$\textbf{locale}\ \mathit{stability} = \mathit{semilat} + \\ \textbf{fixes}\ \mathit{step} :: \mathit{nat} \Rightarrow \sigma \Rightarrow (\mathit{nat} \times \sigma)\ \mathit{list}$$

Data flow analysis is concerned with solving data flow equations, which are systems of equations involving the flow functions over a semilattice. In this 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 I define what it means that a method type $\varphi :: \sigma\ \mathit{list}$ is **stable at** p:

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

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

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

$\top$ is assumed to be a special element in the state space (the top element of the ordering). It indicates a type error.

An instruction sequence is **welltyped**, if there is a welltyping φ such that *wt-step* φ.

2.3.2 Constraints on the Transfer Function

This section defines constraints on the transfer functions that the algorithms in Section 2.4 and 2.5 need to succeed.

The transfer function *step* is called **monotone up to** n iff the following holds:

$$\begin{array}{l} \mathit{mono\ step\ n\ r\ A} \equiv \\ \forall\, p{<}n.\ \forall\, s \in A.\ \forall\, t \in A.\ s \le_r t \longrightarrow \mathit{set}\ (\mathit{step}\ p\ s) \le_{\{r\}} \mathit{set}\ (\mathit{step}\ p\ t) \end{array}$$

where

$$A \le_{\{r\}} B \equiv \forall\, (p,s) \in A.\ \exists\, s'.\ (p,s') \in B \wedge s \le_r s'$$

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).

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

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

If for all $p < n$ and $s \in A$ the values *step p s* returns are in A, then *step* **preserves** A **up to** n:

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

Finally, the soundness proofs for the algorithms in Section 2.4 and 2.5 both use the supremum not only of two, but of a list of elements. Because there is not always a bottom element available, I circumvent the empty list case by making $\bigsqcup_f$ a binary operation (similar to *foldl f*)[1] that takes a list and a start element.

$$\begin{array}{l} \bigsqcup\ ::\ \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{binop} \Rightarrow \alpha \Rightarrow \alpha \\ \qquad\quad [\,]\ \bigsqcup_f\ y = y \\ (x\#xs)\ \bigsqcup_f\ y = xs\ \bigsqcup_f\ (x \sqcup_f y) \end{array}$$

The following lemmas, proved by induction on *xs*, show that the characteristics of $\sqcup_f$ carry over to $\bigsqcup_f$:

Lemma 2.11 If f is closed, so is $\bigsqcup_f$. The following holds in the semilattice context:

$$\mathit{set}\ xs \subseteq A \wedge y \in A \longrightarrow xs\ \bigsqcup_f\ y \in A$$

Lemma 2.12 In a semilattice, $xs \bigsqcup_f y$ is an upper bound for y:

[1]Because of associativity and commutativity of *f*, the definition here is in fact equivalent to *foldl f* and *foldr f*. Lemmas 2.11 to 2.14 follow more directly with this definition, though.

$$set\ xs \subseteq A \wedge y \in A \longrightarrow y \le_r xs \bigsqcup_f y$$

Lemma 2.13 In a semilattice, $xs \bigsqcup_f y$ is an upper bound for all elements of xs:

$$set\ xs \subseteq A \wedge y \in A \wedge x \in set\ xs \longrightarrow x \le_r xs \bigsqcup_f y$$

Lemma 2.14 In a semilattice, any upper bound z of xs and y is greater than or equal to $xs \bigsqcup_f y$:

$$z \in A \wedge y \in A \wedge set\ xs \subseteq A \wedge (\forall x \in set\ xs.\ x \le_r z) \wedge y \le_r z \longrightarrow xs \bigsqcup_f y \le_r z$$

Lemmas 2.12 to 2.14 together say that $xs \bigsqcup_f y$ is the least upper bound of xs and y.

2.3.3 Refining the Transfer Function

The single transfer function *step* of Section 2.3.1 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. I will therefore first refine *step* into a part for applicability and a part for effect of instructions here, and then instantiate these parts in Chapters 3 to 6.

We can 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 $eff :: nat \Rightarrow \tau \Rightarrow (nat \times \tau)\ list$, *step* is defined as follows:

$$\begin{array}{lcl}
step\ n\ p\ Err & = & error\ n \\
step\ n\ p\ (OK\ t') & = & if\ app\ p\ t'\ then\ map\text{-}snd\ OK\ (eff\ p\ t')\ else\ error \\
error\ n & \equiv & map\ (\lambda x.\ (x, Err))\ [0..n( ] \\
map\text{-}snd\ f & \equiv & map\ (\lambda(x,y).\ (x,\ f\ y))
\end{array}$$

The parameter n is the size of the instruction list. It is used to propagate the error element *Err* to every position in the method type.

Given an err-semilattice (A,r,f), we can similarly refine the notion of a welltyping w.r.t. *step* to a welltyping w.r.t. *app* and *eff*:

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

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

If we take n to be *size* φ, we can instantiate the *stability* context with the partially applied function *step* (*size* φ) composed of *app* and *eff* as defined above. This *step* (*size* φ) is of the type $nat \Rightarrow \tau\ err \Rightarrow (nat \times \tau\ err)\ list$ the *stability* context expects; it works on the semilattice *Err.sl* (A,r,f) (see also Section 2.2.3).

With the *stability* context instantiated, we can use *wt-step* and conclude that the notions *wt-step* and *wt-app-eff* coincide.

Lemma 2.15 If the composed function *step* (*size* φ) is bounded by *size* φ, and all elements of φ are in *err* A, then

$$\textit{wt-step}\ \varphi \longrightarrow \textit{wt-app-eff}\ (\textit{map ok-val}\ \varphi) \quad \text{where } \textit{ok-val}\ (OK\ x) = x$$

In the other direction:

Lemma 2.16 If the composed function *step* (*size* φ) is bounded by *size* φ, and all elements of φ are in A, then

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

In the earlier version by Nipkow and myself [39], there was an asymmetry between Lemmas 2.15 and 2.16. The more general type of *step* and the function *error* make it unnecessary here.

2.4 Kildall's Algorithm

A welltyping is a witness of welltypedness in the sense of stability. Now I 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 $bcv :: \sigma\ list \Rightarrow \sigma\ list$ is a **bytecode verifier** w.r.t. $n :: nat$ and $A :: \sigma\ set$ iff

$$\forall \varphi_0 \in \textit{list}\ n\ A.\ (\forall p{<}n.\ (bcv\ \varphi_0)!p \neq \top) = (\exists \varphi \in \textit{list}\ n\ A.\ \varphi_0 \leq[r]\ \varphi \wedge \textit{wt-step}\ \varphi)$$

The notation $\leq[r]$ lifts $\leq_r$ to lists (see Section 2.2.6). The definition is in the *stability* context of above, and $\top$ is the top element of the semilattice. In practice, $bcv\ \varphi_0$ itself will be the welltyping, and it will also be the least welltyping. However, it is simpler not to require this.

This section first defines and then verifies a functional version of Kildall's algorithm [36, 55], a standard data flow analysis tool. In fact, the description of bytecode verification in the official JVM specification [51, pages 129–130] is essentially Kildall's algorithm, an iterative computation of the solution to the data flow problem. The main loop operates on a method type φ and a **worklist** w :: *nat set*. The worklist contains the indices of those elements of φ that have changed and whose changes still need to be propagated to their successors. Each iteration picks an element p from w, executes instruction number p, and propagates the new states to the successor instructions of p. Iteration terminates once w becomes empty: in each iteration, p is removed but new elements can be added to w. The algorithm is expressed in terms of a predefined *while*-combinator of type $(\alpha \Rightarrow \mathit{bool}) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$ which satisfies the recursion equation

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

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

$$\begin{array}{l} \mathit{iter}\ \varphi\ w = \mathit{while}\ (\lambda(\varphi,w).\ w \neq \{\}) \\ \qquad\qquad (\lambda(\varphi,w).\ \mathit{let}\ p = \mathit{SOME}\ p.\ p \in w \\ \qquad\qquad\qquad\quad \mathit{in}\ \mathit{propa}\ (\mathit{step}\ p\ (\varphi!p))\ \varphi\ (w-\{p\})) \\ \qquad\qquad (\varphi,w) \end{array}$$

Since the choice *SOME* $p.\ p \in w$ in *iter* is guarded by $w \neq \{\}$, we know that there is a $p \in w$. An implementation is free to choose whichever element it wants.

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

$$\begin{array}{lcl} \mathit{propa}\ []\ \varphi\ w & = & (\varphi,w) \\ \mathit{propa}\ (q'\#qs)\ \varphi\ w & = & \mathit{let}\ (q,t) = q'; \\ & & \quad u = t \sqcup_f \varphi!q; \\ & & \quad w' = (\mathit{if}\ u = \varphi!q\ \mathit{then}\ w\ \mathit{else}\ \mathit{insert}\ q\ w) \\ & & \mathit{in}\ \mathit{propa}\ qs\ (\varphi[q := u])\ w' \end{array}$$

In the terminology of the official JVM specification [51, page 130], t is merged with the state of all successor instructions q, i.e., the supremum is computed. If this results in a change of $\varphi!q$, then q is inserted into w.

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

$$kildall\ \varphi_0 = fst(iter\ \varphi_0\ \{p.\ p < size\ \varphi_0 \wedge \neg stable\ \varphi_0\ p\})$$

Essentially the same algorithm was presented in [39, 58], but in this thesis it works with the more general *step* function, a less restrictive monotonicity condition (the one of Section 2.3.2), and a weaker ascending chain condition (restricted to A). The key theorem—that Kildall's algorithm is a bytecode verifier as defined above—is therefore stronger:

Theorem 2.1 If (A, r, f) is a semilattice, r meets the ascending chain condition on A, and *step* is monotone, preserving, and bounded w.r.t. A and n, then *kildall* is a bytecode verifier w.r.t. A and n.

The basic structure of the proof is the same as in [39]: the work list either becomes smaller, or, if new positions are introduced, the elements they point to are larger than the element at the position that was taken out. Since there are no infinitely ascending chains in r, the algorithm must terminate. During execution, all positions not in the worklist are stable, and the computed method type is always smaller than (or equal to) a true welltyping (which is stable everywhere). Because *step* is monotone, bounded, and preserves A, this remains invariant. At termination, the worklist is empty, and φ is stable everywhere. In the error case, where there is no welltyping, φ is trivially stable because it contains $\top$ everywhere.

This specification of Kildall's algorithm is executable: using [9], I have generated ML code of it. Instantiated with any of the type systems presented in Chapters 3 to 6, it can verify μJava programs. The worklist (in the specification a set) is implemented by a list, the *SOME* operator by *hd*.

2.5 Lightweight Bytecode Verification

2.5.1 Introduction

The lightweight bytecode verifier (LBV) is a bytecode verifier for resource-bounded JVM implementations. The Connected Limited Device Specification [88] proposes an LBV for embedded devices and smart cards.

Because of the relatively high space and time consumption, many resource-bounded JVM implementations still do not provide bytecode verification. They either do not allow dynamic loading of JVM code at all, or they rely on cryptographic methods to ensure that bytecode verification has taken place off-card. In order to allow on-card verification, Rose and Rose [74, 75] proposed a sparse annotation of JVM code with types to enable a one-pass verification of welltypedness. Roughly speaking, this transforms the iterative type reconstruction problem of Section 2.4 into a type checking problem, which is easier. Type checking only needs a single pass to check consistency of the type annotations with the code.

In contrast to the formalization in [38], the algorithm I present here is abstract in the same sense Kildall's algorithm is in Section 2.4. It works on semilattices instead of concrete Java types and takes the transfer function as a parameter. The algorithm is general and powerful enough to handle all type systems for the BV of chapters 3 to 6. This and the level of abstraction sets it apart from the original formalization by Rose.

Type inference is the computation of a method type, while type checking means checking that a given method type fits an instruction sequence. Lightweight bytecode verification is in between: only crucial bits of the method type are given, the rest is computed.

Abstractly, lightweight bytecode verification can be seen as a combination of two principles:

- Result checking: instead of computing the method type, it is merely checked that the given method type fits.
- Trading space for time: it is sufficient to store only the state type for the entry point to each *basic block* (a code sequence with only one entry and exit point) because the remaining state types in that block can be computed in linear time.

The same principles can be applied to any data flow analysis problem.

Data flow analysis of bytecode is nontrivial because multiple execution paths may lead to the same instruction, in which case the state types on these paths have to be merged. This can only occur at the targets of jumps. If the merge produces a new (more general) type, the basic block must be analyzed again with the new type.

The basic idea of lightweight bytecode verification is to eliminate iteration by providing the result of the type reconstruction process at these merge points beforehand. This additional outside information is called the **certificate**. It reduces the type reconstruction to a single linear pass over the instruction sequence: each time we would have to consider more than one path of execution, the result is already there and only needs to be checked, not constructed. The second effect is that apart from the certificate we only need constant memory: the type reconstruction can be reduced to a function that calculates the state type at *pc+1* from the state type at *pc*. After calculating the type at *pc+1*, we can immediately forget about the one at *pc*.

Figure 2.3 shows the situation at the start of the lightweight bytecode verification process for the example program in Figure 1.2.

instruction	certificate
Load 0	*None*
Store 1	*Some* ([*Class A*], [*Class B*, *Err*])
Load 0	*None*
Getfield F A	*None*
Goto −3	*None*

Figure 2.3: Example for lightweight bytecode verification.

At this point, I use the option data type with the following meaning: *None* indicates that the certificate contains no entry at this point. *Some* means that the certificate stores a state type of a reachable instruction.

From the certificate, the whole method type is reconstructed in a single linear pass: The state type *Some* ([], [*Class B*, *Integer*]) from Section 1.4 for the *Load* instruction will be filled in as initialization. The state type for *Store 1* is in the certificate, since *Store* is the target of the *Goto −3* jump. The LBV calculates the effect of *Load 0*, which is *Some* ([*Class B*], [*Class B*, *Integer*]), and checks if the certificate *Some* ([*Class A*], [*Class B*, *Err*]) correctly approximates this result. The computation continues with the certificate value. The types before execution of the next instructions *Load*, *Getfield*, and *Goto* are easily calculated from the current state type and the effect of the instructions alone. We arrive at *Goto* with a current state type *Some* ([*Class A*], [*Class B*, *Class A*]). The LBV now checks if the calculated state type is compatible with the jump target. We did not store the state type of the target, but since it is a jump target, we have an entry in the certificate: we only need to check if the entry correctly approximates our calculated state type.

Note that all execution paths joining at *Store 1* were checked, but no iteration or additional memory was required.

In the terminology of data flow analysis (see for instance [55]), the certificate records the type information at the entry points to basic blocks (and potentially additional points). This is completely standard in (global) data flow analysis where basic blocks are viewed as atomic and their local structure is immaterial. What is more, this view has significant advantages not just for lightweight but also for standard bytecode verification: during the iterative computation of the method type, it is sufficient to store those state types that correspond to entry points of basic blocks. This is a significant reduction in space at no additional cost in time.

In the following sections I present an abstract implementation of the LBV in Isabelle that (as in Section 2.4) can be instantiated with different type systems. Section 2.5.2 defines the algorithm itself, Section 2.5.3 shows safety and soundness, and Section 2.5.4 completeness of lightweight bytecode verification.

2.5.2 The Algorithm

This section describes the Isabelle formalization of the LBV. Like Kildall's algorithm, the LBV builds on a semilattice (A,r,f). Additionally, I assume that there is a top element $\top$ and a bottom element $\bot$ in the semilattice.

In Isabelle, this context is the following:

$$\begin{array}{l}\textbf{locale}\ \mathit{lbv} = \mathit{semilat} + \\ \quad \textbf{fixes}\ T :: \sigma\ (\top)\ \textbf{and}\ B :: \sigma\ (\bot) \\ \quad \textbf{fixes}\ \mathit{step} :: \mathit{nat} \Rightarrow \sigma \Rightarrow (\mathit{nat} \times \sigma)\ \mathit{list} \\ \quad \textbf{assumes}\ \mathit{top}\text{: }\mathit{top}\ r\ \top\ \textbf{and}\ \mathit{T\text{-}A}\text{: } \top \in A \\ \quad \textbf{assumes}\ \mathit{bot}\text{: }\mathit{bottom}\ r\ \bot\ \textbf{and}\ \mathit{B\text{-}A}\text{: } \bot \in A\end{array}$$

The top layer of the algorithm *wtl* is a single sweep through the instruction list that stops if any step returns the error element $\top$.[2]

$$\begin{array}{lll}\mathit{wtl} :: \alpha\ \mathit{list} \Rightarrow \sigma\ \mathit{cert} \Rightarrow \mathit{nat} \Rightarrow \sigma \Rightarrow \sigma & & \\ \mathit{wtl}\ [] & c\ p\ s = s & \\ \mathit{wtl}\ (i\#is) & c\ p\ s = \mathit{let}\ s' = \mathit{wtc}\ c\ p\ s\ \mathit{in} & \\ & \quad \mathit{if}\ s'{=}\top \vee s{=}\top\ \mathit{then}\ \top\ \mathit{else}\ \mathit{wtl}\ \mathit{is}\ c\ (p{+}1)\ s' & \end{array}$$

The function *wtl* takes the instruction list, the certificate, a position in the instruction list, and an element of the semilattice. It yields an element of the semilattice. If this element is not $\top$, the instructions are welltyped. In fact, as it is formulated here, the LBV will return exactly $\bot$ for success and $\top$ for error. However, it is easier in the proofs not to require $\bot$.

The certificate is just a list of semilattice elements:

$$\textbf{types}\ \sigma\ \mathit{cert} = \sigma\ \mathit{list}$$

The LBV expects the certificate to contain the result state type at jump targets and the bottom element otherwise. The **normal** successor of an instruction at position p is $p{+}1$; all other successors are called the **jump targets** of the instruction.

[2]The definition of *wtl* checks if $s{=}\top$. If we assume that *step* is monotone (as it will be later), this check is unnecessary. With the check, however, we can prove soundness even without monotonicity.

Each single step *wtc* of the LBV first looks at the certificate. If it contains $\bot$, we proceed with the current state type *s*, if not, the current instruction is a jump target, which means the correct state type is more general than we expect, and we proceed with the information in the certificate instead. We also check that the certificate does not change *s* arbitrarily: it may only increase *s*.

$$wtc :: \sigma\ cert \Rightarrow nat \Rightarrow \sigma \Rightarrow \sigma$$
$$wtc\ c\ p\ s \equiv if\ c!p = \bot\ then\ wti\ c\ p\ s\ else\ if\ s \le_r c!p\ then\ wti\ c\ p\ (c!p)\ else\ \top$$

The check $s \le_r c!p$ is what makes the LBV safe: *wtc* does not rely completely on the certificate. The certificate is only allowed to make the computed information less precise, it must *not* make *s* more specific or even change *s* to something completely unrelated.

The computation step *wti* for single instructions executes the transfer function *step* at position *p* and state type *s* and merges the results for the normal successor (the *p+1* edge), while checking that the result is compatible with the certificate at all other successors (the jump targets). If *p+1* is not among the successors, the next instruction must be a jump target to be reachable at all, so we can take the value in the certificate as the result.

$$wti :: \sigma\ cert \Rightarrow nat \Rightarrow \sigma \Rightarrow \sigma$$
$$wti\ c\ p\ s \equiv merge\ c\ p\ (step\ p\ s)\ (c!(p+1))$$

$$merge :: \sigma\ cert \Rightarrow nat \Rightarrow (nat \times \sigma)\ list \Rightarrow \sigma \Rightarrow \sigma$$
$$merge\ c\ p\ []\ x = x$$
$$merge\ c\ p\ ((q,t)\#ls)\ x = merge\ c\ p\ ls\ (if\ q{=}p{+}1\ then\ t \sqcup_f x\ else\ if\ t \le_r c!q\ then\ x\ else\ \top)$$

The executable version of *merge* above is hard to reason about. If x is in A and $snd\ `\ set\ ss \subseteq A$, the following equality holds:

$$merge\ c\ p\ ss\ x =$$
$$if\ \forall (q,t) \in set\ ss.\ q{\neq}p{+}1 \longrightarrow t \le_r c!q\ then\ (map\ snd\ [(q,t) \in ss.\ q{=}p{+}1])\ \bigsqcup_f x\ else\ \top$$

The $(map\ \ldots)\ \bigsqcup_f x$ expression is the start element x (the certificate at $p{+}1$) plus the sum over all (normal) successor state types (those (q,t) where $q{=}p{+}1$).

Figure 2.4 demonstrates the *merge* function in an example. For the parameter values $p{=}3$, $ss{=}[(4,t_1),(1,t_2),(4,t_3)]$, and $x{=}c!4$, *merge* checks that $t_2 \le_r c!1$ and returns $c!4 \sqcup_f t_1 \sqcup_f t_3$ as the result (assuming the check was successful).

If the instruction at position *4* is not a jump target (the regular case), the certificate will contain bottom, $c!4{=}\bot$, and we get the desired $t_1 \sqcup_f t_3$. If the instruction is a jump

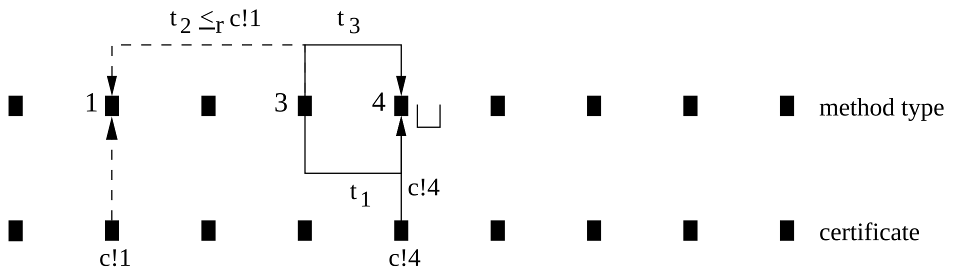

Figure 2.4: The *merge* function with $p{=}3$, $ss = [(4,t_1),(1,t_2),(4,t_3)]$, and $x{=}c!4$.

target, however, the certificate should contain a value with $t_1 \le_r c!4$ and $t_3 \le_r c!4$, so we get $c!4$ as result. Note that this does not make the safety check in *wtc* obsolete; quite to the contrary: we used the check ($t_1 \le_r c!4$ and $t_3 \le_r c!4$) to come to the result $c!4$.[3]

To demonstrate all possible cases, the example above is unrealistically complicated. In practice, *step* will most often return one element only (with $q{=}p{+}1$) and rarely a jump or a list with more than one successor (in the μJVM, this is only the case for conditional jumps, exception handlers, and the subroutine return instruction *Ret*). Although the algorithm above is directly executable in ML (using Isabelle's code generator [9]), real implementations for embedded devices would optimize for the usual case of very short *step* lists.

2.5.3 Soundness

Since the LBV relies on outside information—the certificate—the immediate question is if this is a safe thing to do. The short answer is: yes. If the LBV accepts a piece of code as welltyped, the traditional bytecode verifier accepts it, too, regardless of what certificate was used. This specifically includes the case where the certificate or the program was tampered with. What happens in reality is that in such a case the LBV either rejects the program as not welltyped, or, if it does accept, the program is indeed welltyped and the tampering did no harm.

The soundness theorem uses the notion of welltyping from Section 2.3.1.

Theorem 2.2 If *step* is bounded by *size ins* and preserves A up to *size ins*, if the certificate is wellformed up to *size ins*, and if $s_0 \in A$, then

$$\mathit{wtl\ ins\ c\ 0\ s_0} \neq \top \longrightarrow (\exists\,\varphi.\ \mathit{wt\text{-}step}\ \varphi)$$

[3]One could change the $\le_r$ in *wtc* to $=$, but the correspondence to stability is more visible this way.

```
lemma (in lbv-sound) phi-not-top:
  assumes wtl: wtl ins c 0 s₀ ≠ ⊤ and p: p < size ins
  shows φ!p ≠ ⊤
proof (cases c!p = ⊥)
  case False with p
  have φ!p = c!p ..  also from cert p have ... ≠ ⊤ ..
  finally show ?thesis .
next
  case True with p
  have φ!p = wtl (take p ins) c 0 s₀ ..  also from wtl have ... ≠ ⊤ ..
  finally show ?thesis .
qed
```

Figure 2.5: Proof of Lemma *phi-not-top* in Isabelle/Isar.

A **certificate** c is **wellformed** up to n if $c!n{=}\bot$, and if for all positions i below n, $c!i$ is an element of A other than the top element.

$$\textit{wf-cert}\ c\ n \equiv (\forall\, i{<}n.\ c!i \in A \wedge c!i \neq \top) \wedge (c!n = \bot)$$

The proof of Theorem 2.2 constructs a witness φ for *wt-step* φ. Welltypedness requires that φ is stable and not equal to $\top$ at all positions $p < \textit{size}\ \varphi$. Such a φ is easy to find: the LBV reconstructs φ during the sweep through the instructions. If available, we take the information in the certificate, for the rest we observe what state types the LBV calculates.

Locale *lbv-sound* defines the context of the soundness proof, and with it, the witness φ.

```
locale lbv-sound = lbv +
  fixes s₀ :: σ and c :: σ cert and ins :: α list and phi :: σ list (φ)
  assumes s0: s₀ ∈ A and bounded: bounded step (size ins)
  assumes cert: wf-cert c (size ins) and pres: preserves step (size ins) A
  defines phi-def:
  φ ≡ map (λp. if c!p = ⊥ then wtl (take p ins) c 0 s₀ else c!p) [0..size ins(]
```

The first part of *wt-step*, $\varphi!p \neq \top$, is easy. The proof is by case distinction on the certificate: if there is an entry in p, then φ will have the same entry, and it cannot be $\top$ since the certificate is wellformed. If there is no entry, then φ is the intermediate state type calculated by *wtl* up to p. This cannot be $\top$ either, because otherwise *wtl* would not have succeeded at all. Figure 2.5 shows this proof in Isabelle/Isar.

```
lemma (in lbv-sound) wtl-stable:
  assumes wtl: wtl ins c 0 s_0 ≠ ⊤ and p: p < size ins
  shows stable φ p
proof (unfold stable-def, clarify)
  fix q s' assume step: (q,s') ∈ set (step p (φ!p)) (is - ∈ set (?step p))

  from bounded p step have q: q < size ins by (rule boundedD)
  have tkp: wtl (take p ins) c 0 s_0 ≠ ⊤ (is ?s1 ≠ -) ..
  have s_2: wtl (take (p+1) ins) c 0 s_0 ≠ ⊤ (is ?s2 ≠ -)..
  from wtl p have wt-s_1: wtc c p ?s1 ≠ ⊤ ..
  have c-Some: ∀ p t. p < size ins ⟶ c!p ≠ ⊥ ⟶ φ!p = c!p by (simp add: phi-def)
  have c-None: c!p = ⊥ ⟹ φ!p = ?s1 ..
  from wt-s_1 p c-None c-Some
  have inst: wtc c p ?s1  = wti c p (φ!p) by (simp add: wtc split: split-if-asm)
  have ?s1 ∈ A by (rule wtl-pres)
  with p c-Some cert c-None  have φ!p ∈ A by (cases c!p = ⊥) (auto dest: cert-okD1)
  with p pres have step-in-A: snd`set (?step p) ⊆ A by (auto dest: pres-typeD2)

  show s' ≤_r φ!q
  proof (cases q = p+1)
    case True — q is a normal successor
    with q cert have cert-in-A: c!(p+1) ∈ A by (auto dest: cert-okD1)
    from True q have p1: p+1 < size ins by simp
    with tkp have ?s2 = wtc c p ?s1 by − (rule wtl-Suc)
    with inst have merge: ?s2 = merge c p (?step p) (c!(p+1)) by (simp add: wti)
    also from s_2 merge have ... ≠ ⊤ (is ?merge ≠ -) by simp
    with cert-in-A step-in-A
    have ?merge = (map snd [(q,t)∈?step p. q=p+1] ⨆_f (c!(p+1))) ..
    finally have s' ≤_r ?s2 using step-in-A cert-in-A True step by (auto intro: pp-ub1')
    also from wtl p1 have ?s2 ≤_r φ!(p+1) by (rule wtl-suc-pc)
    also note True [symmetric]
    finally show ?thesis by simp
  next
    case False — q is a jump target
    from wt-s_1 inst have merge c p (?step p) (c!(p+1)) ≠ ⊤ by (simp add: wti)
    with step-in-A have ∀ (q,s')∈set (?step p). q≠p+1 ⟶ s' ≤_r c!q by − rule
    with step False have ok: s' ≤_r c!q by blast
    moreover from ok have c!q = ⊥ ⟹ s' = ⊥ by simp
    moreover from c-Some q have c!q ≠ ⊥ ⟹ φ!q = c!q by auto
    ultimately show ?thesis by (cases c!q = ⊥) auto
  qed
qed
```

Figure 2.6: Proof of Lemma *wtl-stable* in Isabelle/Isar.

The second part of *wt-step*, stability, is more involved. Let's take a look at Lemma *wtl-stable* in Figure 2.6: we may assume $p < \mathit{size}\ \mathit{ins}$ and $(q,s') \in \mathit{set}\ (\mathit{step}\ p\ (\varphi!p))$ for some fixed p, q, and s'. We have to show that φ is stable at p, i.e., that $s' \leq_r \varphi!q$.

The proof begins with a series of observations that concern wellformedness conditions and the nature of $\varphi!q$: since *step* is bounded, q is below *size ins*; *wtl* executed up to p and $p{+}1$ is not $\top$, which means that all checks in the computation up to $p{+}1$ have been successful; φ is either *wtl* up to p or the same as the certificate $c!p$; all state types in the proof are in the carrier set (because *step* preserves A).

After these observations, the proof proceeds with a case distinction whether q is a normal successor or a jump target.

If q is a normal successor, $q{=}p{+}1$, we can conclude that the computation step *wtc* at position p results in $s_2 = \mathit{map}\ \mathit{snd}\ [(q,t){\in}\mathit{step}\ p\ (\varphi!p).\ q{=}p{+}1] \bigsqcup_f (c!q)$. By definition of $\varphi!q$, we know $s_2 \leq_r \varphi!q$ (actually, we know $s_2 = \varphi!q$, since $c!q$ is an element of the sum, but $\leq_r$ is all we need). Since s' is also an element of the sum in the merge expression, it follows that s' is smaller than the sum and therefore also smaller than $\varphi!q$. This concludes the normal successor case.

If q is a jump target of the instruction at p, i.e., $q{\neq}p{+}1$, then we have on the one hand that $\varphi!q = c!q$ and on the other hand that the computation step *wtc* at p must have checked that $s' \leq_r c!q$. The computation step was successful, so we again have $s' \leq_r \varphi!q$.

For the instantiation of the LBV in Chapters 3 to 6, a slightly more precise version of Theorem 2.2 is more convenient.

Theorem 2.3 In the *lbv-sound* context, the following holds:

$$\mathit{wtl}\ \mathit{ins}\ c\ 0\ s_0 \neq \top \wedge \mathit{ins} \neq [] \longrightarrow (\exists\,\varphi \in \mathit{list}\ (\mathit{size}\ \mathit{ins})\ A.\ \mathit{wt\text{-}step}\ \varphi \wedge s_0 \leq_r \varphi!0)$$

The proof just repeats the reasoning of lemma *wtl-stable* for $q{=}0$. The additional premise $\mathit{ins} \neq []$ is there because we need $q < \mathit{size}\ \mathit{ins}$.

2.5.4 Completeness

The soundness theorem ensures that the LBV is safe. The completeness theorem in this sections says that the LBV is also useful: if an instruction sequence is welltyped, it is possible to create a certificate such that the LBV succeeds.

The certificate c is easy to construct: we take a welltyping, computed off-card by a standard BV or by the compiler as in [40, 86], and remove all entries that are not jump targets. For the certificate to be wellformed (in the sense of Section 2.5.3), we append $\bot$ as the last element.

The completeness theorem, then, is the following:

Theorem 2.4 If c is constructed as above, *step* is monotone up to *size* φ, preserves A up to *size* φ, and is bounded by *size* φ, if *set* $\varphi \subseteq A$, $s_0 \in A$, $\bot \neq \top$ and *size ins* $=$ *size* φ, then

$$\textit{wt-step}\ \varphi \wedge s_0 \leq_r \varphi!\mathit{0} \longrightarrow \textit{wtl ins c 0}\ s_0 \neq \top$$

In addition to the conditions of the soundness theorem, we now need *step* to be monotone, the size of φ to coincide with the instruction sequence (otherwise it might cover only a part of *ins*), and $\bot \neq \top$. The latter is required for the simple reason that we need to be able to distinguish success $\bot$ from error $\top$ in the result of *wtl*. The premise $s_0 \leq_r \varphi!\mathit{0}$ is reminiscent of the start condition the JVM bytecode verifier places on the first instruction (see for example Section 3.3.3). It plays a similar role here: it ensures a correct start state type.

In Isabelle, the proof context for completeness is the following. Note that $\varphi!p \neq \top$ is part of the *wt-step* premise in Theorem 2.4, so the *phi* assumption collapses to *set* $\varphi \subseteq A$ there.

> **locale** *lbv-complete* = *lbv* +
>   **fixes** *phi* :: σ *list* (φ) **and** *c* :: σ *cert*
>   **assumes** *mono*: *mono r step* (*size* φ) *A* **and** *pres*: *preserves step* (*size* φ) *A*
>   **assumes** *phi*: $\forall p <$ *size* φ. $\varphi!p \in A \wedge \varphi!p \neq \top$ **and** *bounded*: *bounded step* (*size* φ)
>   **assumes** *B-neq-T*: $\bot \neq \top$
>   **defines** *cert-def*: $c \equiv$ *map* (λp. *if is-target p then* $\varphi!p$ *else* $\bot$) [*0..length* φ(] @ [$\bot$]

The function *is-target* determines whether an instruction at position q is a jump target. If q is a jump target, there must be a predecessor p such that q is a successor, but not a normal successor of p.

$$\begin{array}{l}\textit{is-target} :: \textit{nat} \Rightarrow \textit{bool}\\ \textit{is-target}\ q \equiv \exists\, p\ t.\ q \neq p{+}1 \wedge p < \textit{size}\ \varphi \wedge (q,t) \in \textit{set}\ (\textit{step}\ p\ (\varphi!p))\end{array}$$

In the following, I will sketch the proof of Theorem 2.4. It builds on three important lemmas.

Lemma 2.17 (*stable-wtc*) In the *lbv-complete* context, if $p <$ *size* φ, then

$$\textit{stable}\ \varphi\ p \longrightarrow \textit{wtc}\ c\ p\ (\varphi!p) \neq \top$$

```
lemma (in lbv-complete) lbv-complete-lemma:
  assumes wt-step: wt-step $\varphi$
  shows $\bigwedge p\ s.\ p+size\ ls = size\ \varphi \Longrightarrow s \le_r \varphi!p \Longrightarrow s \in A \Longrightarrow s{\neq}\top \Longrightarrow wtl\ ls\ c\ p\ s \neq \top$
proof (induct ls)
  fix p s assume $s{\neq}\top$ thus $wtl\ []\ c\ p\ s \neq \top$ by simp
next
  fix p s i ls
  assume $\bigwedge p\ s.\ p+size\ ls{=}size\ \varphi \Longrightarrow s \le_r \varphi!p \Longrightarrow s \in A \Longrightarrow s{\neq}\top \Longrightarrow wtl\ ls\ c\ p\ s \neq \top$
  moreover assume p-l: $p + size\ (i\#ls) = size\ \varphi$
  hence suc-p-l: $Suc\ p + size\ ls = size\ \varphi$ by simp
  ultimately
  have IH: $\bigwedge s.\ s \le_r \varphi!Suc\ p \Longrightarrow s \in A \Longrightarrow s \neq \top \Longrightarrow wtl\ ls\ c\ (Suc\ p)\ s \neq \top$ .

  from p-l obtain p: $p < size\ \varphi$ by simp
  with wt-step have stable: stable $\varphi$ p by (simp add: wt-step-def)
  hence wt-phi: $wtc\ c\ p\ (\varphi!p) \neq \top$ by (rule stable-wtc)
  from phi p have phi-p: $\varphi!p \in A$ by simp
  moreover assume s: $s \in A$ and s-phi: $s \le_r \varphi!p$
  ultimately
  have wt-s-phi: $wtc\ c\ p\ s \le_r wtc\ c\ p\ (\varphi!p)$ by $-$ (rule wtc-mono)
  with wt-phi have wt-s: $wtc\ c\ p\ s \neq \top$ by simp
  moreover assume s: $s \neq \top$
  ultimately
  have $ls = [] \Longrightarrow wtl\ (i\#ls)\ c\ p\ s \neq \top$ by simp
  moreover {
    assume $ls \neq []$
    with p-l have suc-p: $Suc\ p < size\ \varphi$ by (auto simp add: neq-Nil-conv)
    with stable have $wtc\ c\ p\ (\varphi!p) \le_r \varphi!Suc\ p$ by (rule wtc-less)
    with wt-s-phi have $wtc\ c\ p\ s \le_r \varphi!Suc\ p$ by (rule trans-r)
    moreover
    from cert suc-p have $c!p \in A$ and $c!(p{+}1) \in A$
      by (auto simp add: cert-ok-def)
    with pres have $wtc\ c\ p\ s \in A$ by (rule wtc-pres)
    ultimately
    have $wtl\ ls\ c\ (Suc\ p)\ (wtc\ c\ p\ s) \neq \top$ using IH wt-s by blast
    with s wt-s have $wtl\ (i\#ls)\ c\ p\ s \neq \top$ by simp
  }
  ultimately show $wtl\ (i\#ls)\ c\ p\ s \neq \top$ by (cases ls) blast+
qed
```

Figure 2.7: Proof of *wtl-complete-lemma* in Isabelle/Isar.

Lemma 2.18 (*wtc-less*) In the *lbv-complete* context, if $p+1 < size\ \varphi$, then

$$stable\ \varphi\ p \longrightarrow wtc\ c\ p\ (\varphi!p) \leq_r \varphi!(p+1)$$

Lemma 2.19 (*wtc-mono*) In the *lbv-complete* context, if $p < size\ \varphi$, $s_1 \in A$, and $s_2 \in A$, then

$$s_2 \leq_r s_1 \longrightarrow wtc\ c\ p\ s_2 \leq_r wtc\ c\ p\ s_1$$

The first two lemmas, 2.17 and 2.18, are based on the observation that *stable* requires for all successors (q,t) in *step* p $(\varphi!p)$ that $t \leq_r \varphi!q$. The LBV focuses on those where $q{=}p{+}1$ for the computation, the rest is only checked (with the same condition). If all t with $q{=}p{+}1$ are smaller than $\varphi!q$, then also the sum of the t must be smaller than $\varphi!q$. The sum also contains the certificate $c!q$, but $c!q$ is either $\bot$ or equal to $\varphi!q$. In both cases we get $c!q \leq_r \varphi!q$, and therefore $wtc\ c\ p\ (\varphi!p) \leq_r \varphi!q$ with $q{=}p{+}1$. Lemma 2.19 just lifts the monotonicity of *step* up to *wtc*.

The proof of the main completeness theorem (Theorem 2.4) is then by induction on the instruction sequence. For the induction to go through, we have to strengthen the goal. Under the assumption *wt-step* φ, we show:

$$\forall p\ s.\ p{+}size\ ls = size\ \varphi \land s \leq_r \varphi!p \land s \in A \land s{\neq}\top \longrightarrow wtl\ ls\ c\ p\ s \neq \top$$

Figure 2.7 contains the Isabelle/Isar proof for this lemma.

The base case of the induction is as easy as it should be: from the assumption $s{\neq}\top$ we immediately get $wtl\ []\ c\ p\ s \neq \top$.

In the step case, there is a first instruction i and a rest list ls for which we have to show $wtl\ (i\#ls)\ c\ p\ s \neq \top$. The first steps in Figure 2.7 reduce the induction hypothesis to $\bigwedge s.\ s \leq_r \varphi!Suc\ p \Longrightarrow s \in A \Longrightarrow s \neq \top \Longrightarrow wtl\ ls\ c\ (Suc\ p)\ s \neq \top$. If we instantiate s with $wtc\ c\ p\ s$ and can get our hands on the conclusion of the induction hypothesis $wtl\ ls\ c\ (Suc\ p)\ (wtc\ c\ p\ s) \neq \top$, we have proved the goal, because $wtl\ ls\ c\ (Suc\ p)\ (wtc\ c\ p\ s)$ is the same as $wtl\ (i\#ls)\ c\ p\ s$ provided $wtc\ c\ p\ s \neq \top$. We may assume that $s \leq_r \varphi!p$ and (from *wt-step* φ) that *stable* φ p holds.

To use the induction hypothesis, we have to show $wtc\ c\ p\ s \leq_r \varphi!(p{+}1)$, $wtc\ c\ p\ s \in A$, and $wtc\ c\ p\ s \neq \top$. The proof in Figure 2.7 begins with the last of these premises. Using Lemma 2.17 (*stable-wtc*), we conclude $wtc\ c\ p\ (\varphi!p) \neq \top$ from *stable* φ p. With $s \leq_r \varphi!p$ we know by monotonicity (Lemma 2.19) of *wtc* that $wtc\ c\ p\ s \leq_r wtc\ c\ p\ (\varphi!p)$ and thus the desired $wtc\ c\ p\ s \neq \top$. The next lines in Figure 2.7 handle the special case where $ls = []$. Here, we immediately have $wtl\ (i\#ls)\ c\ p = wtc\ c\ p\ s \neq \top$. The interesting case $ls \neq []$ gives us $p{+}1 < size\ \varphi$ and we use Lemma 2.18 to get

$wtc\ c\ p\ (\varphi!p) \leq_r \varphi!Suc\ p$. Because we already know $wtc\ c\ p\ s \leq_r wtc\ c\ p\ (\varphi!p)$, we thereby also have the first premise of the induction hypothesis, $wtc\ c\ p\ s \leq_r \varphi!Suc\ p$, by transitivity of $\leq_r$. The remaining premise, $wtc\ c\ p\ s \in A$, follows from the fact that *step* is type preserving. With this, we can use the induction hypothesis and conclude the final goal $wtl\ (i\#ls)\ c\ p\ s \neq \top$.

This concludes the proof of the induction lemma. Theorem 2.4 is a corollary of it.

2.5.5 Conclusion

Above, I have presented a framework for lightweight bytecode verification. It contains the lightweight bytecode verifier as an abstract and executable functional program that is sound and complete. Both properties are generic in the type system, proved with respect to the typing framework of Section 2.3.1. In this abstract setting, the main Isabelle/Isar proofs are small enough to be shown here.

The specification of the lightweight bytecode verifier consists of only 37 lines of Isabelle definitions. The proofs of soundness and completeness including all related lemmas take up about 1000 lines of human-readable Isabelle/Isar text. The LBV-specific instantiation of the framework is about another 300 lines for each type system.

The abstract setting of an arbitrary semilattice and the general *step* function allows the LBV to be used for all type systems I present in this thesis. Most notably, the structure of the data flow graph may again depend on the current state type, which enables the LBV to be used for the notorious bytecode subroutines [25] which are not supported by Sun's KVM (they must be eliminated by expansion before verification).

In comparison to my formalization, the original approach of Rose [74, 75] is less general and also a bit more complex. It is less general, because she cannot handle bytecode subroutines and because she formulates it for the JVM only.[4] It is more complex because she does not distinguish type system from algorithm in the formalization, and because she includes a compression optimization for certificates: she only needs the certificate when a type merge really produces a different type than expected, which leads to a smaller type annotation. It does, however, not save space during the verification pass itself, since the state type at all jump targets has to be saved for later checks anyway. My completeness result includes the simpler and easier to implement notion that the certificate should contain all jump targets. This is also used in Sun's KVM verifier.

The soundness theorem states that the lightweight bytecode verifier accepts only type correct programs, and that it is safe to rely on outside information. The complete-

[4]I have instantiated the LBV for the μJVM only, but in principle this algorithm is applicable wherever standard bytecode verification can be used.

ness theorem states that the lightweight bytecode verifier will accept the same well-typed programs as the traditional bytecode verifier. Both theorems combined tell us that lightweight bytecode verification is functionally completely equivalent to traditional bytecode verification. The functional implementation shows that the algorithm is linear in time and constant in space. All these results together enable a secure scheme for on-card verification with Java smartcards: programs are annotated with a certificate, produced by a traditional bytecode verifier or directly by the compiler off-card. On-card verification can then take place with the efficient and compact lightweight bytecode verifier as part of the card's JVM. This scheme provides easy, seamless use for developers while maintaining all security properties from bytecode verification that we have become accustomed to. The major advantage over cryptographic methods is that no trust is needed at all in the certifying party and authenticity of the certificate.

3 Instantiating the Framework

In this chapter, I instantiate the abstract typing framework with a first, simple type system, show that this type system is sound, and derive two verified executable bytecode verifiers from the specification. I also describe in detail the JVM formalization and the structure of programs. Both the JVM and the BV will serve as basis for the extensions in Chapters 4 to 6.

3.1 Introduction

This chapter shows the formalization of the μJVM and its BV. The JVM formalization begins by defining types, values, programs, declarations, wellformedness conditions, and lookup functions. It proceeds with the memory and state model of the μJVM and finally shows the definition of the operational semantics. The semantics is split in two parts: an aggressive machine that executes programs without type and safety checks, and a defensive machine that includes those checks. The aggressive machine is closer to a real implementation, the defensive machine is useful to express type safety. It is shown that the two are equivalent if no type errors occur.

The formalization of the BV is an instantiation of the framework in Chapter 2. It defines a suitable semilattice and transfer function and uses the theorems and definitions of the framework to derive a static description of welltypings together with two executable bytecode verifiers. With the definition of welltypings, it is then possible to show that all welltyped programs are type safe.

The type system presented here is an extended, more modular version of [60]. It contains classes, inheritance, virtual methods, and exception handling. I keep it relatively simple to introduce the μJVM formalization, which is the basis for the proof of type safety, and to demonstrate a first instantiation of the abstract typing framework.

The type safety proof uses the defensive machine for stating the theorem, and an invariant argument to show that execution remains safe. I describe the definition of the invariant as well as an exemplary case of the proof in Isabelle/Isar notation.

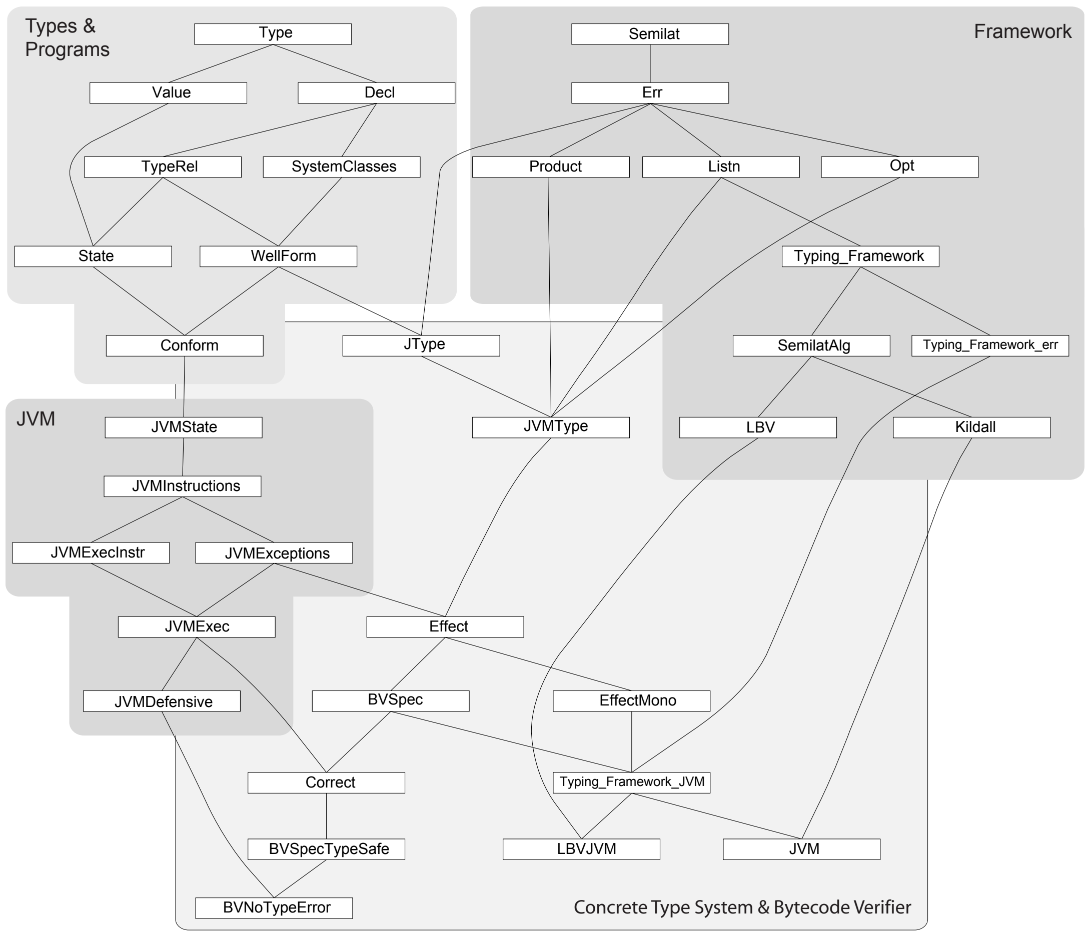

Figure 3.1: Instantiating the framework: overview.

Figure 3.1 gives an overview of the Isabelle theories involved.

The theory dependency graph in Figure 3.1 contains four clusters. The upper right corner shows the framework of Chapter 2 (the three LBV theories are merged into one to disentangle the picture). Section 3.2.1 will be concerned with the upper left corner, the machine independent part of μJava: types, declarations, and the structure of programs. The rest of Section 3.2 is dedicated to the definition of the μJVM and its operational semantics: the lower left of Figure 3.1. The remaining theories in Figure 3.1 contain the type system with the executable bytecode verifiers and its proof of soundness. Section 3.3 shows the type system and the bytecode verifiers, Section 3.4 the proof of type safety.

3.2 The μJava Virtual Machine

This section gives a detailed introduction to the μJava Virtual Machine. It contains the structure of μJava programs (Section 3.2.1), the state space of the μJVM (Section 3.2.2), and its operational semantics, first without (Section 3.2.3) and then with (Section 3.2.4) runtime type checks.

3.2.1 Structure

Declarations

Since μJava formalizes both the source and the bytecode language, it distinguishes code from declarations in a special way: declarations are shared, code is a type variable that is instantiated later (with either bytecode language instructions or source language terms). This thesis will only be concerned with the bytecode level.

The declaration structure of programs is the following in Isabelle:

$$
\begin{array}{lll}
\textbf{types} & \mathit{fdecl} & = \mathit{vname} \times \mathit{ty} \\
 & \mathit{sig} & = \mathit{mname} \times \mathit{ty\ list} \\
 & \gamma\ \mathit{mdecl} & = \mathit{sig} \times \mathit{ty} \times \gamma \\
 & \gamma\ \mathit{class} & = \mathit{cname} \times \mathit{fdecl\ list} \times \gamma\ \mathit{mdecl\ list} \\
 & \gamma\ \mathit{cdecl} & = \mathit{cname} \times \gamma\ \mathit{class} \\
 & \gamma\ \mathit{prog} & = \gamma\ \mathit{cdecl\ list}
\end{array}
$$

This is best read from bottom to top: a program γ *prog* with method bodies of type γ is a list of class declarations. A class declaration γ *cdecl* consists of the name of the class and the class itself. A class γ *class* is again a tuple: it consists of the name of the superclass, a list of field declarations, and a list of method declarations. A method declaration γ *mdecl* has a signature *sig* (name and parameter types of the method), a return type *ty*, and the body γ. A field declaration *fdecl* is a pair of field name and field type.

Class names are based on a not further specified set of names *cnam*:

$$
\begin{array}{l}
\textbf{datatype}\ \mathit{cname} = \mathit{Object} \mid \mathit{Xcpt\ xcpt} \mid \mathit{Cname\ cnam} \\
\textbf{datatype}\ \mathit{xcpt} = \mathit{NullPointer} \mid \mathit{ClassCast} \mid \mathit{OutOfMemory}
\end{array}
$$

The names of system exceptions and the name of class *Object* are predefined, all other classes have names out of *cnam*.

For the bytecode level, the names of methods *mname* and the names of fields *vname* can be left unspecified like *cnam*.

The HOL data type *ty* of basic types in μJava is the following:

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

The above means that a type *ty* is either a primitive type or a reference type. Primitive types can be the usual *Void*, *Boolean*, or *Integer*. Reference types are the null type (for the *Null* reference), and classes. For readability, μJava uses the following abbreviations, implemented as *syntax translations* in Isabelle:

$$\begin{array}{lrcl}
\textbf{translations} & \mathit{NT} & \rightleftharpoons & \mathit{RefT}\ \mathit{NullT} \\
 & \mathit{Class}\ C & \rightleftharpoons & \mathit{RefT}\ (\mathit{ClassT}\ C)
\end{array}$$

A program $\Gamma :: \gamma\ \mathit{prog}$ gives rise to a lookup function *class* that takes a program and a class name and returns the class if it is declared (and *None* otherwise). The function $\mathit{map\text{-}of} :: (\alpha \times \beta)\ \mathit{list} \Rightarrow (\alpha \Rightarrow \beta\ \mathit{option})$ turns a list of pairs into a map.

$$\begin{array}{l}
\mathit{class} :: \gamma\ \mathit{prog} \Rightarrow \mathit{cname} \Rightarrow \gamma\ \mathit{class}\ \mathit{option} \\
\mathit{class}\ \Gamma \equiv \mathit{map\text{-}of}\ \Gamma
\end{array}$$

With this, we can determine if a class or a type is declared in a program:

$$\begin{array}{l}
\mathit{is\text{-}class} :: \gamma\ \mathit{prog} \Rightarrow \mathit{cname} \Rightarrow \mathit{bool} \\
\mathit{is\text{-}class}\ \Gamma\ C \equiv \mathit{class}\ \Gamma\ C \neq \mathit{None}
\end{array}$$

$$\begin{array}{l}
\mathit{is\text{-}type} :: \gamma\ \mathit{prog} \Rightarrow \mathit{ty} \Rightarrow \mathit{bool} \\
\mathit{is\text{-}type}\ \Gamma\ (\mathit{PrimT}\ t) = \mathit{True} \\
\mathit{is\text{-}type}\ \Gamma\ (\mathit{RefT}\ t) = (\mathit{case}\ t\ \mathit{of}\ \mathit{NullT} \Rightarrow \mathit{True} \mid \mathit{ClassT}\ C \Rightarrow \mathit{is\text{-}class}\ \Gamma\ C)
\end{array}$$

Subtypes

Each program Γ also induces a subtype ordering $\preceq$. It builds on the direct subclass relation $\mathit{subcls}\ \Gamma$ and satisfies:

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

The expression $(C,D) \in (\mathit{subcls}\ \Gamma)^*$ means that C is a subclass of D in Γ. In Isabelle, *subcls* Γ is an inductive definition with the following sole introduction rule:

$$[\![\mathit{class}\ \Gamma\ C = \mathit{Some}\ (D,\mathit{decl});\ C \neq \mathit{Object}]\!] \Longrightarrow (C,D) \in \mathit{subcls}\ \Gamma$$

For every class hierarchy, which means for every program, this subtype ordering may be a different one. In the Isabelle formalization, the ordering $\preceq$ therefore has Γ as an additional parameter. Here, I treat Γ as a global constant in most definitions.

Wellformedness

Like the Java Language Specification (JLS) [32], μJava imposes certain wellformedness constraints on programs.

Wellformedness of fields and method headers is easy. A field declaration is wellformed if the field has a declared type, and a method header is wellformed if the return type and all parameter types are declared:

$$\begin{array}{l}\mathit{wf\text{-}fdecl} :: \gamma\ \mathit{prog} \Rightarrow \mathit{fdecl} \Rightarrow \mathit{bool}\\ \mathit{wf\text{-}fdecl}\ \Gamma\ (\mathit{fn},\mathit{ft}) \equiv \mathit{is\text{-}type}\ \Gamma\ \mathit{ft}\end{array}$$

$$\begin{array}{l}\mathit{wf\text{-}mhead} :: \gamma\ \mathit{prog} \Rightarrow \mathit{sig} \Rightarrow \mathit{ty} \Rightarrow \mathit{bool}\\ \mathit{wf\text{-}mhead}\ \Gamma\ (\mathit{mn},\mathit{ps})\ \mathit{rt} \equiv \mathit{is\text{-}type}\ \Gamma\ \mathit{rt} \wedge (\forall\, T{\in}\mathit{set}\ \mathit{ps}.\ \mathit{is\text{-}type}\ \Gamma\ T)\end{array}$$

Since method bodies γ are not instantiated yet, wellformedness of programs is parameterized with a function *wf-mb* that checks wellformedness of method bodies. For bytecode, this will later be the bytecode verifier. The type of *wf-mb* is

$$\textbf{types}\ \gamma\ \mathit{wf\text{-}mb} = \gamma\ \mathit{prog} \Rightarrow \mathit{cname} \Rightarrow \gamma\ \mathit{mdecl} \Rightarrow \mathit{bool}$$

This means the function *wf-mb* gets the program, the current class, and the current method declaration as input. With this information, it decides whether the method body is wellformed.

A full method declaration is then wellformed if its header and its body are wellformed:

$$\begin{array}{l}\mathit{wf\text{-}mdecl} :: \gamma\ \mathit{wf\text{-}mb} \Rightarrow \gamma\ \mathit{prog} \Rightarrow \mathit{cname} \Rightarrow \gamma\ \mathit{mdecl} \Rightarrow \mathit{bool}\\ \mathit{wf\text{-}mdecl}\ \mathit{wf\text{-}mb}\ \Gamma\ C\ (\mathit{sig},\mathit{rt},\mathit{mb}) \equiv \mathit{wf\text{-}mhead}\ \Gamma\ \mathit{sig}\ \mathit{rt} \wedge \mathit{wf\text{-}mb}\ \Gamma\ C\ (\mathit{sig},\mathit{rt},\mathit{mb})\end{array}$$

Wellformedness of class declarations is more involved.

$$\begin{aligned}
&\textit{wf-cdecl} :: \gamma\ \textit{wf-mb} \Rightarrow \gamma\ \textit{prog} \Rightarrow \gamma\ \textit{cdecl} \Rightarrow \textit{bool}\\
&\textit{wf-cdecl}\ \textit{wf-mb}\ \Gamma\ (C,(D,fs,ms)) \equiv\\
&(\forall f{\in}\textit{set}\ fs.\ \textit{wf-fdecl}\ \Gamma\ f) \wedge \textit{unique}\ fs \wedge\\
&(\forall m{\in}\textit{set}\ ms.\ \textit{wf-mdecl}\ \textit{wf-mb}\ \Gamma\ C\ m) \wedge \textit{unique}\ ms \wedge\\
&(C \neq \textit{Object} \longrightarrow \textit{is-class}\ \Gamma\ D \wedge (D,C) \notin (\textit{subcls}\ \Gamma)^* \wedge\\
&\quad (\forall (sig,rt,b){\in}\textit{set}\ ms.\ \forall D'\ rt'\ b'.\ \textit{method}\ (\Gamma,D)\ sig = \textit{Some}(D',rt',b') \longrightarrow rt \preceq rt'))
\end{aligned}$$

The beginning of *wf-cdecl* is canonical: all field and method declarations must be well-formed, and each field and method should only be declared once (*unique*). The next line makes sure that all superclasses that are mentioned in declarations are actually declared in the program and that the subclass relation is acyclic. The last line is a bit less restrictive than the JLS: it requires that a method declared in class C has a return type smaller than the one of methods with the same signature, but defined in a superclass D of C. The JLS requires overriding methods to have the same return type as the overridden method. The method lookup function *method* will be defined formally below. Before that, I finish wellformedness:

$$\begin{aligned}
&\textit{wf-syscls} :: \gamma\ \textit{prog} \Rightarrow \textit{bool}\\
&\textit{wf-syscls}\ \Gamma \equiv \textit{let}\ cs = \textit{set}\ \Gamma\ \textit{in}\ \textit{Object} \in \textit{fst}`cs \wedge (\forall x.\ \textit{Xcpt}\ x \in \textit{fst}`cs)\\
&\textit{wf-prog} :: \gamma\ \textit{wf-mb} \Rightarrow \gamma\ \textit{prog} \Rightarrow \textit{bool}\\
&\textit{wf-prog}\ \textit{wf-mb}\ \Gamma \equiv\ \textit{unique}\ \Gamma \wedge \textit{wf-syscls}\ \Gamma \wedge (\forall c \in \textit{set}\ \Gamma.\ \textit{wf-cdecl}\ \textit{wf-mb}\ \Gamma\ c)
\end{aligned}$$

The above says a program is wellformed (*wf-prog*) if each class is declared only once, if it is wellformed w.r.t. system classes, and if all class declarations are wellformed. It is wellformed w.r.t. system classes (*wf-syscls*) if it contains a declaration of the class *Object* and a declaration for each system exception.

Finally, a program is called **well structured** (*ws-prog* Γ), if it wellformed without the method bodies:

$$\textit{ws-prog}\ \Gamma \equiv \textit{wf-prog}\ (\lambda\Gamma\ C\ \textit{mdecl}.\ \textit{True})\ \Gamma$$

Lookup functions

The method lookup function *method* and its analogue *fields* for fields are large and tedious recursive definitions in Isabelle. Slightly more readable are the following equations that hold if $\textit{wf}\ ((\textit{subcls}\ \Gamma)^{-1})$ and $\textit{class}\ \Gamma\ C = \textit{Some}\ (D,fs,ms)$:

$$\begin{array}{l} \mathit{method} :: \gamma\ \mathit{prog} \times \mathit{cname} \Rightarrow (\mathit{sig} \Rightarrow (\mathit{cname} \times \mathit{ty} \times \gamma)\ \mathit{option}) \\ \mathit{method}\ (\Gamma,C) = (\mathit{if}\ C{=}\mathit{Object}\ \mathit{then}\ \mathit{empty}\ \mathit{else}\ \mathit{method}\ (\Gamma,D))\ {+}{+} \\ \qquad\qquad\qquad \mathit{map\text{-}of}\ (\mathit{map}\ (\lambda(\mathit{sig},\mathit{rt},b).\ (\mathit{sig},(C,\mathit{rt},b)))\ \mathit{ms}) \end{array}$$

$$\begin{array}{l} \mathit{fields} :: \gamma\ \mathit{prog} \times \mathit{cname} \Rightarrow ((\mathit{vname} \times \mathit{cname}) \times \mathit{ty})\ \mathit{list} \\ \mathit{fields}\ (\Gamma,C) = \mathit{map}\ (\lambda(\mathit{fn},\mathit{ft}).\ ((\mathit{fn},C),\mathit{ft}))\ \mathit{fs}\ @\ (\mathit{if}\ C{=}\mathit{Object}\ \mathit{then}\ []\ \mathit{else}\ \mathit{fields}\ (\Gamma,D)) \end{array}$$

The *method* function looks up whether a method with signature *sig* exists in class C and program Γ. If it exists, it yields the name of the class the method is defined in (it could be a superclass of C), the return type of the method, and the body of the method. If it does not exist, it yields *None*. The equation for *method* above first recurses into the superclass D and overwrites (using the ++ operator) the resulting map with the methods of the current class. Note that *class* Γ C = *Some* (D,fs,ms) ensures that *ms* is the list of method declarations in class C and that D is the superclass of C.

The *fields* function is similar. For a class C in program Γ, it returns the list of all fields of this class (including those defined in superclasses). For each field, it returns its name, the name of the class it is defined in, and the field's type. The equation above collects the fields of the current class and then recurses into the superclass D.

Both functions walk up the class hierarchy. It is therefore necessary that the converse of *subcls* Γ is wellfounded (otherwise they would not terminate), hence the condition $\mathit{wf}\ ((\mathit{subcls}\ \Gamma)^{-1})$. Note that whenever *wf-prog mb* Γ holds for any wellformedness condition *mb* of method bodies, $\mathit{wf}\ ((\mathit{subcls}\ \Gamma)^{-1})$ must be true. Both functions are only specified if the class C in which they are supposed to look up a method or field does exist in Γ, hence the condition *class* Γ C = *Some* (D,fs,ms). The way both functions are formulated coincides with Java's rules for inheritance and overriding.

The *fields* function has a variant *field* that takes a program Γ, a class C, and a field name *fn*, and that returns *Some* (D,ft) if the field exists (where D is the class the field is defined in and *ft* the type of the field). If there is no field with name *ft* accessible in C, then *field* (Γ,C) *fn* returns *None*. It is defined by:

$$\begin{array}{l} \mathit{field} :: \gamma\ \mathit{prog} \times \mathit{cname} \Rightarrow (\mathit{vname} \Rightarrow (\mathit{cname} \times \mathit{ty})\ \mathit{option}) \\ \mathit{field} \equiv \mathit{map\text{-}of} \circ (\mathit{map}\ (\lambda((\mathit{fn},C),\mathit{ft}).\ (\mathit{fn},(C,\mathit{ft})))) \circ \mathit{fields} \end{array}$$

Values

The type *val* of values is defined by

$$\textbf{datatype}\ \mathit{val} = \mathit{Unit} \mid \mathit{Null} \mid \mathit{Bool}\ \mathit{bool} \mid \mathit{Intg}\ \mathit{int} \mid \mathit{Addr}\ \mathit{loc}$$

Unit is a (dummy) result value of void methods, *Null* the null reference. *Bool* and *Intg* are injections from the HOL types *bool* and *int* into *val*, similarly *Addr* from the type *loc* of locations.

In the spirit of JavaCard, the type *loc* reserves space for preallocated system exception objects:[1]

$$\textbf{datatype}\ \mathit{loc} = \mathit{XcptRef}\ \mathit{xcpt} \mid \mathit{Loc}\ \mathit{locs}$$

The type *locs* again remains unspecified.

3.2.2 State Space

The state space of the μJVM is modeled closely after the real JVM. 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).

$$\textbf{types}\ \mathit{jvm\text{-}state} = \mathit{val\ option} \times \mathit{aheap} \times \mathit{frame\ list}$$

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

$$\textbf{types}\ \mathit{aheap} = \mathit{loc} \Rightarrow \mathit{obj\ option}$$

An object *obj* 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).

$$\textbf{types}\ \mathit{obj} = \mathit{cname} \times (\mathit{vname} \times \mathit{cname} \Rightarrow \mathit{val\ option})$$

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 (name and parameter types) of the method and arrive at:

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

[1]Thus also avoiding the situation where there is no space left for a new *OutOfMemory* exception object.

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
| *Return*	return from method
| *Dup*	duplicate top element
| *Dup-x1*	duplicate top element and push 2 values down
| *IAdd*	integer addition
| *Goto int*	goto relative address
| *Ifcmpeq int*	branch if equal
| *Throw*	throw exception

Figure 3.2: The μJava bytecode instruction set.

3.2.3 Operational Semantics

This section defines the state transition relation of the μJVM. Figure 3.2 shows the instruction set. Method bodies are lists of such instructions together with the exception handler table and two numbers *mxs* and *mxl*. These are the maximum operand stack size and the number of local variables (not counting the *this* pointer and the parameters of the method, which are stored in the first *0* to *n* registers). So the type parameter γ for method bodies gets instantiated with *nat* × *nat* × *instr list* × *ex-table*; *mdecl* becomes the following:

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

Like in the JVM, the exception table is a list of tuples (*start-pc*, *end-pc*, *handler-pc*, *C*):

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

The asymmetric interval [*start-pc*, *end-pc*) denotes those instructions in the method body that correspond to the *try* block on the Java source level. The *handler-pc* points to the first instruction of the corresponding *catch* block. The code starting at *handler-pc* is the **exception handler**. An exception handler **protects** a program position *pc* iff $pc \in [start\text{-}pc, end\text{-}pc)$. An exception table entry **matches** an exception *e* if the handler protects the current *pc* and if the class of *e* is a subclass of *C*.

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

```
exec :: jvm-state ⇒ jvm-state option
exec (xp, hp, [])          = None
exec (Some xp, hp, frs)    = None
exec (None, hp, f#frs)     = let (stk,reg,C,sig,pc) = f;
                                 ins = 5th (the (method (Γ,C) sig));
                                 i = ins ! pc
                             in find-handler (exec-instr i hp stk reg C sig pc frs)
```

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 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.

Exception handling in *find-handler* is the same 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*:

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

The definition of *exec-instr* in Figure 3.3 is large, but straightforward. One of the smaller definitions in *exec-instr* is the one for the *IAdd* instruction:

```
exec-instr IAdd hp stk regs C' sig pc frs =
        let $i_1$ = the-Intg (hd stk);  $i_2$ = the-Intg (hd (tl stk))
        in (None, hp, (Intg ($i_1$+$i_2$))#(tl (tl stk)), regs, C', sig, pc+1)#frs)
```

It takes the top two values from the stack, converts them to HOL integers (using the equality *the-Intg* (*Intg i*) = *i*), adds them and puts the result back onto the stack. The program counter is incremented, the rest remains untouched. Most instructions in Figure 3.3 are of that simple form. Instructions with exceptions and heap access have larger definitions (due to case distinctions), but it should become clear that *exec-instr* follows the JVM specification closely.

```
exec-instr :: instr ⇒ aheap ⇒ opstack ⇒ registers ⇒ cname ⇒ sig ⇒ nat ⇒ frame list ⇒
          jvm-state

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

exec-instr (Store idx) hp stk regs C′ sig pc frs =
                          (None, hp, (tl stk, regs[idx:=hd stk], C′, sig, pc+1)#frs)

exec-instr (LitPush v) hp stk regs C′ sig pc frs =
                          (None, hp, (v # stk, regs, C′, sig, pc+1)#frs)

exec-instr (New C) hp stk regs C′ sig pc frs =
                          let (a,xp′) = new-Addr hp;
                              hp′ = if xp′=None then hp(a↦blank Γ C) else hp;
                              pc′ = if xp′=None then pc+1 else pc
                          in (xp′, hp′, ((Addr a) # stk, regs, C′, sig, pc′)#frs)

exec-instr (Getfield F C) hp stk regs C′ sig pc frs =
                          let r = hd stk;
                              xp′ = raise-system-xcpt (r=Null) NullPointer;
                              (c,fs) = the(hp(the-Addr r));
                              pc′ = if xp′=None then pc+1 else pc
                          in (xp′, hp, (the(fs(F,C))#(tl stk), regs, C′, sig, pc′)#frs)

exec-instr (Putfield F C) hp stk regs C′ sig pc frs =
                          let (v,r)= (hd stk, hd(tl stk));
                              xp′ = raise-system-xcpt (r=Null) NullPointer;
                              a = the-Addr r;
                              (c,fs) = the (hp a);
                              hp′ = if xp′=None then hp(a↦(c,fs((F,C)↦v))) else hp;
                              pc′ = if xp′=None then pc+1 else pc
                          in (xp′, hp′, (tl (tl stk), regs, C′, sig, pc′)#frs)

exec-instr (Checkcast C) hp stk regs C′ sig pc frs =
                          let r = hd stk;
                              xp′ = raise-system-xcpt (¬ cast-ok Γ C hp r) ClassCast;
                              stk′ = if xp′=None then stk else tl stk;
                              pc′ = if xp′=None then pc+1 else pc
                          in (xp′, hp, (stk′, regs, C′, sig, pc′)#frs)
```

Figure 3.3: Single step execution (part 1).

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

exec-instr Return hp $stk_0$ regs C' $sig_0$ pc frs =
          if frs=[] then (None, hp, []) else
          let v = hd $stk_0$;
              (stk,regs,C,sig,pc) = hd frs;
              (mn,ps) = $sig_0$;
              n = size ps
          in (None, hp, (val#(drop (n+1) stk),regs,C,sig,pc+1)#tl frs)

exec-instr Pop hp stk regs C' sig pc frs =
          (None, hp, (tl stk, regs, C', sig, pc+1)#frs)

exec-instr Dup hp stk regs C' sig pc frs =
          (None, hp, (hd stk # stk, regs, C', sig, pc+1)#frs)

exec-instr Dup-x1 hp stk regs C' sig pc frs =
          (None, hp, (hd stk # hd (tl stk) # hd stk # (tl (tl stk)), regs, C', sig, pc+1)#frs)

exec-instr IAdd hp stk regs C' sig pc frs =
          let (i1,i2) = (the-Intg (hd stk), the-Intg (hd (tl stk)))
          in (None, hp, (Intg (i1+i2)#(tl (tl stk)), regs, C', sig, pc+1)#frs)

exec-instr (Ifcmpeq b) hp stk regs C' sig pc frs =
          let ($v_1$,$v_2$) = (hd stk, hd (tl stk));
              pc' = if $v_1$ = $v_2$ then nat(int pc+b) else pc+1
          in (None, hp, (tl (tl stk), regs, C', sig, pc')#frs)

exec-instr (Goto b) hp stk regs C' sig pc frs =
          (None, hp, (stk, regs, C', sig, nat(int pc+b))#frs)

exec-instr Throw hp stk regs C' sig pc frs =
          let xp = raise-system-xcpt (hd stk = Null) NullPointer;
              xp' = if xp = None then Some (hd stk) else xp
          in (xp', hp, (stk, regs, C', sig, pc)#frs)
```

Figure 3.4: Single step execution (part 2).

Figure 3.3 uses new functions: the destructor *the-Addr* is analogous to *the-Intg*. The *New* instruction needs *new-Addr*, returning either an *OutOfMemory* exception *xp* or an unused heap address *a*. It leaves memory size and the implementation of memory management unspecified. A new object of class C with all fields set to default values is produced by *blank* Γ C. The *Getfield* definition generates the exception explicitly with *raise-system-xcpt b xp* $\equiv$ *if b then Some* (*Addr* (*XcptRef xp*)) *else None*. Note that the field part of objects is a map from defining class and name to value, so $fs(F,C)$ is the value of field F defined in class C. The *Checkcast* instruction uses *cast-ok* Γ C *hp* v to check if the value v is an address that points to an object of at least class C.

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, for instance for *IAdd* that there are indeed two integers on the stack. 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.

3.2.4 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. The 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 data type.

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

Similar to Section 3.2.3, 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*) and check that the *pc* is valid:

```
$check :: jvm\text{-}state \Rightarrow bool$
$check\ (xp,\ hp,\ [])$        $= True$
$check\ (xp,\ hp,\ f\#frs)$ $= let\ (stk,reg,C,sig,pc) = f;$
                                 $ins = 5th\ (the\ (method\ (\Gamma,C)\ sig));$
                                 $i = ins!pc$
                             $in\ pc < size\ ins \wedge check\text{-}instr\ i\ hp\ stk\ reg\ C\ sig\ pc\ frs$
```

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:

$$\begin{array}{l} \textit{exec-d} :: \textit{jvm-state type-error} \Rightarrow \textit{jvm-state option type-error} \\ \textit{exec-d TypeError} \ = \textit{TypeError} \\ \textit{exec-d} \ (\textit{Normal } s) = \textit{if check } s \textit{ then Normal } (\textit{exec } s) \textit{ else TypeError} \end{array}$$

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

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

It remains to define *check-instr*, the heart of the defensive μJVM. Figure 3.5 shows that this is relatively straightforward. The *IAdd* case looks like this:

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

IAdd requires that the stack has at least two entries ($1 < \textit{size stk}$), and that these entries are of type *Integer* (checked with the *is-Intg* function). For *Load* and *Store* there are no type constraints, because they are polymorphic in μJava. In the real JVM, the definition would be in the style of *IAdd*, requiring integer for `iload`, float for `fload`, and so on. The discriminator functions *is-Addr* and *is-Ref* in Figure 3.5 do the obvious. More interesting is the relation $::\preceq$ between values and arbitrary types:

$$hp \vdash v ::\preceq T \equiv \exists\, T'.\ \textit{typeof hp } v = \textit{Some } T' \wedge T' \preceq T$$

$$\begin{array}{ll} \textit{typeof hp Unit} & = \textit{Some } (\textit{PrimT Void}) \\ \textit{typeof hp } (\textit{Intg } i) & = \textit{Some } (\textit{PrimT Integer}) \\ \textit{typeof hp } (\textit{Bool } b) & = \textit{Some } (\textit{PrimT Boolean}) \\ \textit{typeof hp Null} & = \textit{Some NT} \\ \textit{typeof hp } (\textit{Addr } a) & = \textit{case hp } a \textit{ of None} \Rightarrow \textit{None} \mid \textit{Some } (C,\textit{fs}) \Rightarrow \textit{Some } (\textit{Class } C) \end{array}$$

So $hp \vdash v ::\preceq T$ (v **conforms to** T **in** hp) implies that v is either a value of primitive type T, or that v is *Null* and T some reference type, or, if it is an address a, that there is an object of at least type T at position a on the heap. In the next section I also use *typeof* v without the heap parameter. It is equivalent to *typeof* ($\lambda a.\ \textit{None}$) v and returns *None* for all addresses. The predicate *is-Intg* v is equivalent to $\textit{typeof } v = \textit{Some } (\textit{PrimT Integer})$ as well as to $hp \vdash v ::\preceq (\textit{PrimT Integer})$.

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

$$\begin{array}{l}
\mathit{check\text{-}instr} :: \mathit{instr} \Rightarrow \mathit{aheap} \Rightarrow \mathit{opstack} \Rightarrow \mathit{registers} \Rightarrow \mathit{cname} \Rightarrow \mathit{sig} \Rightarrow \mathit{nat} \Rightarrow \mathit{frame\ list} \Rightarrow \\
\qquad\qquad \mathit{bool}
\end{array}$$

$$\begin{array}{ll}
\mathit{check\text{-}instr}\ (\mathit{Load\ idx})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{idx} < \mathit{size\ regs} \\
\mathit{check\text{-}instr}\ (\mathit{Store\ idx})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{0} < \mathit{size\ stk} \wedge \mathit{idx} < \mathit{size\ regs} \\
\mathit{check\text{-}instr}\ (\mathit{LitPush\ v})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \neg \mathit{is\text{-}Addr\ v} \\
\mathit{check\text{-}instr}\ (\mathit{New\ C})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{is\text{-}class}\ \Gamma\ C
\end{array}$$

$$\begin{array}{l}
\mathit{check\text{-}instr}\ (\mathit{Getfield\ F\ C})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} = \\
\qquad \mathit{0} < \mathit{size\ stk} \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{field}\ (\Gamma,C)\ F \neq \mathit{None} \wedge \\
\qquad (\mathit{let}\ (C',T) = \mathit{the}\ (\mathit{field}\ (\Gamma,C)\ F);\ \mathit{ref} = \mathit{hd\ stk} \\
\qquad \mathit{in}\ C' = C \wedge \mathit{is\text{-}Ref\ ref} \wedge \\
\qquad\quad (\mathit{ref} \neq \mathit{Null} \longrightarrow\ \mathit{hp}\ (\mathit{the\text{-}Addr\ ref}) \neq \mathit{None} \wedge (\mathit{let}\ (D,\mathit{fs}) = \mathit{the}\ (\mathit{hp}\ (\mathit{the\text{-}Addr\ ref})) \\
\qquad\qquad\qquad \mathit{in}\ (D,C) \in (\mathit{subcls}\ \Gamma)^* \wedge \mathit{fs}\ (F,C) \neq \mathit{None} \wedge \mathit{hp} \vdash \mathit{the}\ (\mathit{fs}\ (F,C)) ::\preceq T)))
\end{array}$$

$$\begin{array}{l}
\mathit{check\text{-}instr}\ (\mathit{Putfield\ F\ C})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} = \\
\qquad \mathit{1} < \mathit{size\ stk} \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{field}\ (\Gamma,C)\ F \neq \mathit{None} \wedge \\
\qquad (\mathit{let}\ (C',\ T) = \mathit{the}\ (\mathit{field}\ (\Gamma,C)\ F);\ v = \mathit{hd\ stk};\ \mathit{ref} = \mathit{hd}\ (\mathit{tl\ stk}) \\
\qquad \mathit{in}\ C' = C \wedge \mathit{is\text{-}Ref\ ref} \wedge \\
\qquad\quad (\mathit{ref} \neq \mathit{Null} \longrightarrow \mathit{hp}\ (\mathit{the\text{-}Addr\ ref}) \neq \mathit{None} \wedge \\
\qquad\qquad (\mathit{let}\ (D,\mathit{fs}) = \mathit{the}\ (\mathit{hp}\ (\mathit{the\text{-}Addr\ ref}))\ \mathit{in}\ (D,C) \in (\mathit{subcls}\ \Gamma)^* \wedge \mathit{hp} \vdash v ::\preceq T)))
\end{array}$$

$$\begin{array}{l}
\mathit{check\text{-}instr}\ (\mathit{Checkcast\ C})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} = \\
\qquad \mathit{0} < \mathit{size\ stk} \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{is\text{-}Ref}\ (\mathit{hd\ stk})
\end{array}$$

$$\begin{array}{l}
\mathit{check\text{-}instr}\ (\mathit{Invoke\ C\ mn\ ps})\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} = \\
\qquad \mathit{size\ ps} < \mathit{size\ stk} \wedge \\
\qquad (\mathit{let}\ n = \mathit{size\ ps};\ v = \mathit{stk}!n\ \mathit{in} \\
\qquad \mathit{is\text{-}Ref}\ v \wedge (v \neq \mathit{Null} \longrightarrow \mathit{hp}\ (\mathit{the\text{-}Addr}\ v) \neq \mathit{None} \wedge \\
\qquad\qquad\qquad\qquad \mathit{method}\ (\Gamma,\mathit{cname\text{-}of\ hp}\ v)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \wedge \\
\qquad\qquad\qquad\qquad \mathit{list\text{-}all2}\ (\lambda v\ T.\ \mathit{hp} \vdash v ::\preceq T)\ (\mathit{rev}\ (\mathit{take}\ n\ \mathit{stk}))\ \mathit{ps}))
\end{array}$$

$$\begin{array}{l}
\mathit{check\text{-}instr}\ \mathit{Return\ hp\ stk0\ regs\ Cl\ sig0\ pc\ frs} = \\
\qquad \mathit{0} < \mathit{size\ stk0} \wedge \\
\qquad (\mathit{0} < \mathit{size\ frs} \longrightarrow \mathit{method}\ (\Gamma,\mathit{Cl})\ \mathit{sig0} \neq \mathit{None} \wedge \\
\qquad\quad (\mathit{let}\ v = \mathit{hd\ stk0};\ (C,\mathit{rt},b) = \mathit{the}\ (\mathit{method}\ (\Gamma,\mathit{Cl})\ \mathit{sig0})\ \mathit{in}\ \mathit{Cl} = C \wedge \mathit{hp} \vdash v ::\preceq \mathit{rt}))
\end{array}$$

$$\begin{array}{ll}
\mathit{check\text{-}instr}\ \mathit{Pop\ hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{0} < \mathit{size\ stk} \\
\mathit{check\text{-}instr}\ \mathit{Dup\ hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{0} < \mathit{size\ stk} \\
\mathit{check\text{-}instr}\ \mathit{Dup\text{-}x1\ hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{1} < \mathit{size\ stk} \\
\mathit{check\text{-}instr}\ \mathit{IAdd\ hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{1} < \mathit{size\ stk} \wedge \mathit{is\text{-}Intg}\ (\mathit{hd\ stk}) \wedge \\
 & \quad \mathit{is\text{-}Intg}\ (\mathit{hd}\ (\mathit{tl\ stk})) \\
\mathit{check\text{-}instr}\ (\mathit{Ifcmpeq}\ b)\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{1} < \mathit{size\ stk} \wedge \mathit{0} \leq \mathit{int\ pc}+b \\
\mathit{check\text{-}instr}\ (\mathit{Goto}\ b)\ \mathit{hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{0} \leq \mathit{int\ pc}+b \\
\mathit{check\text{-}instr}\ \mathit{Throw\ hp\ stk\ regs\ Cl\ sig\ pc\ frs} & = \mathit{0} < \mathit{size\ stk} \wedge \mathit{is\text{-}Ref}\ (\mathit{hd\ stk})
\end{array}$$

Figure 3.5: Type checks in the defensive μJVM.

$$\begin{tikzcd}
\text{Normal s} \arrow[r, "\text{exec-d}"] \arrow[d, leftrightarrow] & \text{Normal t} \arrow[d, leftrightarrow] \\
\text{s} \arrow[r, "\text{exec}"'] & \text{t}
\end{tikzcd}$$

Figure 3.6: Aggressive and defensive μJVM commute if there are no type errors.

Theorem 3.1 One-step execution in aggressive and defensive machines commutes if there are no type errors.

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

Figure 3.6 depicts this result as a commuting diagram. The proof is trivial (and fully automatic in Isabelle), because the defensive VM is constructed directly from the aggressive one.

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. I call a method the **main method** of class C if there is a method body b such that *method* (Γ,C) $(\textit{main},[])$ = *Some* $(C,\ b)$ holds. For *main* methods, we can define the canonical start state *start* Γ C as the state with exception flag *None*, a heap *start-hp* Γ, and a frame stack with one element. The heap *start-hp* Γ contains the preallocated system exceptions and is otherwise empty. The single frame has an empty operand stack, the *this* pointer set to *Null*, the rest of the register set filled up with a dummy value *arbitrary*, the class entry set to *C*, the signature to $(\textit{main},[])$, and the program counter *0*.

```
start :: jvm-prog ⇒ cname ⇒ jvm-state type-error
start Γ C ≡ let (-,-,-,mxl,-,-) = the (method (Γ,C) (main,[]));
                                    regs = Null # replicate mxl arbitrary
                  in Normal (None, start-hp Γ, [([], regs, C, (main,[]), 0)])
```

This concludes the formalization of the μJVM. It will serve as the basis for the proof of type safety below.

3.3 The Bytecode Verifier

In the following sections, I instantiate the abstract typing framework from Chapter 2 with a concrete type system for the μJVM. I define the semilattice structure in Section 3.3.1, the data flow functions *app* and *eff* in Section 3.3.2, refine the notion of welltyping to Java-specifics in Section 3.3.3, and finally derive two verified executable bytecode verifiers from the specification in Section 3.3.4.

3.3.1 The Semilattice

This section takes the first step to instantiate the framework of Chapter 2. It defines the semilattice structure on which μJava's bytecode verifier builds. It begins by turning the μJava types *ty* into a semilattice. We can then use the abstract combinators of Section 2.2 to construct the stack and register structure.

The carrier set *types* is easy: the set of all types declared in the program.

$$types = \{T.\ is\text{-}type\ \Gamma\ T\}$$

The order is the standard subtype ordering $\preceq$ of μJava.

The supremum operation follows the ordering.

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

The *lub* function (not shown here) computes the least upper bound of two classes by walking up the class hierarchy until one is a subclass of the other. Since every class is a subclass of *Object* in a well structured program (see also *ws-prog* in Section 3.2.1), this least upper bound is guaranteed to exist.

With these three components, I have proved the following theorem.

Theorem 3.2 If Γ is well structured, the triple $JType.esl \equiv (types, \preceq, sup)$ is an err-semilattice and the subtype ordering $\preceq$ satisfies the ascending chain condition.

The proof is easy: it is obvious that $\preceq$ is transitive and reflexive. If Γ is well structured, $\preceq$ is also antisymmetric, hence a partial order. It satisfies the ascending chain condition, because, if Γ is well structured, the class hierarchy is a tree with *Object* at its top. I have already argued above that *sup* is well defined, and it is easy to see that it is closed w.r.t. *types*. Hence *esl* is an err-semilattice.

We can now construct the stack and register structure. State types in the μJava BV are the same as in the example in Figure 1.2: values on the operand stack must always contain a known type *ty*, values in the local variables may be of an unknown type and therefore be unusable (encoded by *Err*). On the HOL-type level, this is the following.

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

To handle unreachable code, the BV will not directly work on *state-type*, but on *state-type option* instead. If *None* occurs in the welltyping, the corresponding instruction is unreachable. The executable BV also needs to indicate type errors during the algorithm (see also Section 2.3.3), so we arrive at *state-type option err*.

Turning *state-type option err* into a semilattice is easy, because all of its constituent types are (err-)semilattices. The expression stacks form a semilattice because the supremum of stacks of different size is *Err*; the list of registers forms a semilattice because the number *mxr* of registers is fixed:

$$\begin{array}{l}
\mathit{stk\text{-}esl} :: \mathit{nat} \Rightarrow \mathit{ty\ list\ esl}\\
\mathit{stk\text{-}esl}\ \mathit{mxs} \equiv \mathit{upto\text{-}esl}\ \mathit{mxs}\ (\mathit{JType.esl})\\[4pt]
\mathit{reg\text{-}sl} :: \mathit{nat} \Rightarrow \mathit{ty\ err\ list\ sl}\\
\mathit{reg\text{-}sl}\ \mathit{mxr} \equiv \mathit{Listn.sl}\ \mathit{mxr}\ (\mathit{Err.sl}\ (\mathit{JType.esl}))
\end{array}$$

The stack and registers are combined in a coalesced product via *Prod.esl* and then embedded into *option* and *err* to create the final semilattice for σ = *state-type option err*:

$$\begin{array}{l}
\mathit{sl} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{state\text{-}type\ option\ err\ sl}\\
\mathit{sl}\ \mathit{mxs}\ \mathit{mxr} \equiv \mathit{Err.sl}(\mathit{Opt.esl}(\mathit{Prod.esl}\ (\mathit{stk\text{-}esl}\ \mathit{mxs})\ (\mathit{Err.esl}(\mathit{reg\text{-}sl}\ \mathit{mxr}))))
\end{array}$$

It is useful below to have special notation for the ordering on the *state-type option* and *state-type* level:

$$\begin{array}{l}
\le' :: \mathit{state\text{-}type\ option} \Rightarrow \mathit{state\text{-}type\text{-}option} \Rightarrow \mathit{bool}\\
\le_s :: \mathit{state\text{-}type} \Rightarrow \mathit{state\text{-}type} \Rightarrow \mathit{bool}
\end{array}$$

Combining the theorems about the various (err-)semilattice constructions involved in the definition of *esl* (starting from Theorem 3.2, using Lemmas 2.1 to 2.9), it is easy to prove

Corollary 3.1 If Γ is well structured, then *sl* is a semilattice. Its order satisfies the ascending chain condition.

3.3.2 Applicability and Effect

In this section, I instantiate *app* and *eff* for the instruction set of the μJVM. The definitions are divided into one part for normal and one part for exceptional execution.

Since the BV 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,
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.

The context variables are proper parameters of *eff* and *app* in the Isabelle formalization. I treat them as global here to spare the reader endless parameter lists in each definition.

The definitions of *app* and *eff* begin 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 3.7 do the actual lookup: *match X pc et* returns [*Xcpt X*] if there is 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*.

As shown in Figure 3.7, most instructions in *xcpt-names* are straightforward. *Getfield*, for instance, can only generate *NullPointer* exceptions, *Invoke* may propagate up any exception from the invoked method. *Throw* considers all exception handlers for position *pc*, because the runtime type of the exception thrown may be a subclass of the type the BV infers (and therefore be caught by another exception handler as predicted). One could be slightly more precise for the *Throw* instruction, but this would only add clutter to the specification without new insights to the workings of the BV.

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

$$\begin{array}{l} \textit{xcpt-app} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{bool} \\ \textit{xcpt-app}\ i\ pc \equiv \forall\, C{\in}\textit{set}(\textit{xcpt-names}\ (i,pc,et)).\ \textit{is-class}\ \Gamma\ C \end{array}$$

The definition of *xcpt-eff* below, the effect in the exception case, uses the function *match-ex-table C pc et*, returning *Some handler-pc* if there is an exception handler in

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

Figure 3.7: Exception names.

the table *et* for an exception of class *C* thrown at position *pc*, and *None* otherwise. The effect is the same for all instructions: the registers *lt* of the current state type *s* remain the same, the stack is cleared, and a reference to the exception object is pushed. I use *option-map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *option* $\Rightarrow \beta$ *option* to lift functions to the option type:

$$\begin{array}{l}
\textit{option-map}\ f\ \textit{None} = \textit{None} \\
\textit{option-map}\ f\ (\textit{Some}\ x) = \textit{Some}\ (f\ x)
\end{array}$$

The successor instruction pc' in the data flow graph marks the beginning of the exception handler returned by *match-ex-table*. This effect occurs for every exception class *C* the instruction may possibly throw (determined by *xcpt-names* as for *xcpt-app* above).

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

This concludes the exception case and we can proceed to the applicability of instructions in the normal, non-exception case. The intermediate function app', defined in Figure 3.8, works on *state-type*, *app* then lifts it to *state-type option*. The definition is parallel to *check-instr* in Section 3.2.4, it just works on types instead of values. The definition is smaller than the one of *check-instr* because some of the conditions cannot be expressed at the type level alone. These conditions are the ones that access the heap or the frame stack (most notable in the *Getfield*, *Putfield*, and *Return* instructions). The type safety proof in Section 3.4 will show why the BV still manages to guarantee that all checks in the defensive machine are successful.

Let's take a closer look at the *IAdd* example:

$$app'\ (\textit{IAdd},\ pc,\ (t_1\#t_2\#st,lt)) = (t_1 = t_2 \wedge t_1 = \textit{PrimT Integer})$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{app}' :: \mathit{instr} \times \mathit{nat} \times \mathit{state\text{-}type} \Rightarrow \mathit{bool}} \\
\mathit{app}'\ (\mathit{Load}\ \mathit{idx},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{lt} \wedge \mathit{lt}!\mathit{idx} \neq \mathit{Err} \wedge \\
 & \ \mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Store}\ \mathit{idx},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size}\ \mathit{lt} \\
\mathit{app}'\ (\mathit{LitPush}\ v,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{st} < \mathit{mxs} \wedge \\
 & \ \mathit{typeof}\ v \neq \mathit{None} \\
\mathit{app}'\ (\mathit{Getfield}\ F\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge t \preceq \mathit{Class}\ C \wedge \\
 & \ (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,\ t')) \\
\mathit{app}'\ (\mathit{Putfield}\ F\ C,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \ (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,t') \wedge \\
 & \ t_2 \preceq \mathit{Class}\ C \wedge t_1 \preceq t') \\
\mathit{app}'\ (\mathit{New}\ C,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Checkcast}\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{is\text{-}RefT}\ t \\
\mathit{app}'\ (\mathit{Dup},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{1}{+}\mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Dup\text{-}x1},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{2}{+}\mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{IAdd},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \wedge t_1 = \mathit{PrimT}\ \mathit{Integer} \\
\mathit{app}'\ (\mathit{Ifcmpeq}\ b,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{0} \leq \mathit{int}\ \mathit{pc} + b \wedge ((t_1 = t_2) \vee \\
 & \ (\mathit{is\text{-}RefT}\ t_1 \wedge \mathit{is\text{-}RefT}\ t_2)) \\
\mathit{app}'\ (\mathit{Goto}\ b,\ \mathit{pc},\ s) & = \mathit{0} \leq \mathit{int}\ \mathit{pc} + b \\
\mathit{app}'\ (\mathit{Return},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = t \preceq \mathit{rt} \\
\mathit{app}'\ (\mathit{Throw},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}RefT}\ t \\
\mathit{app}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st} \wedge \\
 & \ \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \wedge \\
 & \ \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st}); \\
 & \ \qquad t\ = \mathit{st}!\mathit{size}\ \mathit{ps} \\
 & \ \mathit{in}\ t \preceq \mathit{Class}\ C \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \ \quad (\forall\,(a,f) \in \mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq f) \\
\mathit{app}'\ (i,\mathit{pc},s) & = \mathit{False}
\end{array}
$$

Figure 3.8: Applicability of instructions.

$$\begin{array}{lll}
\mathit{succs} :: \mathit{instr} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat\ list} & & \\
\mathit{succs}\ (\mathit{Ifcmpeq}\ b)\ \mathit{pc} & = & [\mathit{pc}{+}\mathit{1},\ \mathit{nat}\ (\mathit{int}\ \mathit{pc} + b)] \\
\mathit{succs}\ (\mathit{Goto}\ b)\ \mathit{pc} & = & [\mathit{nat}\ (\mathit{int}\ \mathit{pc} + b)] \\
\mathit{succs}\ \mathit{Return}\ \mathit{pc} & = & [] \\
\mathit{succs}\ \mathit{Throw}\ \mathit{pc} & = & [] \\
\mathit{succs}\ i\ \mathit{pc} & = & [\mathit{pc}{+}\mathit{1}]
\end{array}$$

Figure 3.9: Successor program counters for the non-exception case.

This is completely parallel to the defensive machine. The pattern on the left hand side ensures that there are at least two elements on the stack, and the right hand side requires that they both are integers.

With *app'*, we can now build the full applicability function *app*: an instruction is applicable if it is unreachable (then it can do no harm) or if it is applicable in the normal and in the exceptional case. Additionally, we require that the *pc* does not leave the instruction sequence.

$$\begin{array}{l}
\mathit{app} :: \mathit{instr} \Rightarrow \mathit{nat} \Rightarrow \mathit{state\text{-}type\ option} \Rightarrow \mathit{bool} \\
\mathit{app}\ i\ \mathit{pc}\ s \equiv \mathit{case}\ s\ \mathit{of}\ \ \mathit{None} \ \ \Rightarrow \mathit{True} \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{Some}\ s \Rightarrow \mathit{xcpt\text{-}app}\ i\ \mathit{pc} \wedge \mathit{app'}\ (i,\mathit{pc},s) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall (\mathit{pc'},s') \in \mathit{set}\ (\mathit{eff}\ i\ s).\ \mathit{pc'} < \mathit{mpc})
\end{array}$$

This concludes applicability. We still need 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 3.9 shows the successors. Again, most instructions are as expected. The relative jumps in *Ifcmpeq* and *Goto* use the *nat* and *int* functions to convert the HOL-types *nat* to *int* and vice versa. *Return* and *Throw* have no successors in the same method (for the non-exception case).

As with *app* we first define the effect *eff'* on *state-type* (Figure 3.10). The destructor *ok-val* is defined by $\mathit{ok\text{-}val}\ (\mathit{OK}\ x) = x$. The *method* expression for *Invoke* merely determines the return type of the method in question.

The *IAdd* instruction is here:

$$\mathit{eff'}\ (\mathit{IAdd},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) = (\mathit{PrimT\ Integer}\#\mathit{st},\mathit{lt})$$

Again, as befits an abstract interpretation, the definition is completely parallel to the operational semantics, this time to *exec-instr* of the aggressive machine.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{eff}' :: \mathit{instr} \times \mathit{state\text{-}type} \Rightarrow \mathit{state\text{-}type}} \\
\mathit{eff}'\ (\mathit{Load\ idx},\ (\mathit{st},\mathit{lt})) & = (\mathit{ok\text{-}val}\ (\mathit{lt}!\mathit{idx})\#\mathit{st},\ \mathit{lt}) \\
\mathit{eff}'\ (\mathit{Store\ idx},\ (t\#\mathit{st},\mathit{lt})) & = (\mathit{st},\ \mathit{lt}[\mathit{idx}{:=}\ \mathit{OK}\ t]) \\
\mathit{eff}'\ (\mathit{LitPush}\ v,\ (\mathit{st},\mathit{lt})) & = (\mathit{the}\ (\mathit{typeof}\ v)\#\mathit{st},\ \mathit{lt}) \\
\mathit{eff}'\ (\mathit{Getfield}\ F\ C,\ (t\#\mathit{st},\mathit{lt})) & = (\mathit{snd}\ (\mathit{the}\ (\mathit{field}\ (\Gamma,C)\ F))\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Putfield}\ F\ C,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{New}\ C,\ (\mathit{st},\mathit{lt})) & = (\mathit{Class}\ C\ \#\ \mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Checkcast}\ C,\ (t\#\mathit{st},\mathit{lt})) & = (\mathit{Class}\ C\ \#\ \mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Dup},\ (t\#\mathit{st},\mathit{lt})) & = (t\#t\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Dup\text{-}x1},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (t_1\#t_2\#t_1\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{IAdd},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (\mathit{PrimT\ Integer}\#\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Ifcmpeq}\ b,\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (\mathit{st},\mathit{lt}) \\
\mathit{eff}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ (\mathit{st},\mathit{lt})) & = \mathit{let}\ \mathit{st}' = \mathit{drop}\ (\mathit{1}{+}\mathit{size}\ \mathit{ps})\ \mathit{st}; \\
 & \qquad (\_,\mathit{rt},\_) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps})) \\
 & \quad\ \mathit{in}\ (\mathit{rt}\#\mathit{st}',\ \mathit{lt})
\end{array}
$$

Figure 3.10: Effect of instructions on the state type.

Lifting *eff'* to *state-type option* is canonical.

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

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*.

$$
\begin{array}{l}
\mathit{eff} :: \mathit{instr} \Rightarrow \mathit{nat} \Rightarrow \mathit{state\text{-}type\ option} \Rightarrow (\mathit{nat} \times \mathit{state\text{-}type\ option})\ \mathit{list} \\
\mathit{eff}\ i\ \mathit{pc}\ s \equiv (\mathit{map}\ (\lambda \mathit{pc}'.\ (\mathit{pc}',\ \mathit{norm\text{-}eff}\ i\ s))\ (\mathit{succs}\ i\ \mathit{pc}))\ @\ (\mathit{xcpt\text{-}eff}\ i\ \mathit{pc}\ s)
\end{array}
$$

3.3.3 Welltypings

Having defined the semilattice and the transfer function in Section 3.3.1 and 3.3.2, I show in this section how the parts are put together to get a definition of welltypings for the μJVM.

The framework of Chapter 2 gives us a predicate *wt-app-eff* describing welltypings φ :: *state-type option list* as method types that fit an instruction sequence. The JVM specification [51] requires an additional start condition for instruction *0* (at method invocation). It also requires that the instruction sequence is not empty.

The JVM specification tells us what the first state type (at method invocation) looks like: the stack is empty, the first register contains the *this* pointer (of type *Class C′*), the next registers contain the parameters of the method, and the rest of the registers are reserved for local variables (which do not have a value yet). Note that the definitions are still in the context of a fixed method as defined in Section 3.3.2, so C' is the class to be verified, *ps* are the parameters, and *mxl* the number of local variables (which is related to *mxr* by $mxr = 1{+}size\ ps{+}mxl$). The $\leq'$ is the semilattice order on *state-type option* of Section 3.3.1.

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

The method type φ is a welltyping if it satisfies *wt-method*.

$$wt\text{-}method\ \varphi \equiv 0 < mpc \wedge map\ OK\ \varphi \in states \wedge wt\text{-}start\ \varphi \wedge wt\text{-}app\text{-}eff\ \varphi$$

The *states* are the carrier set of the semilattice. Remember that the method type φ does not contain the *Err* layer of the semilattice, hence the *map OK* φ. The condition *map OK* $\varphi \in$ *states* is only necessary to prove equivalence with the type inference algorithms below. It does not follow from *wt-app-eff*, because $a \leq_r b$ does not necessarily imply that $a \in states \longrightarrow b \in states$.

For the type safety proof below, it is useful to have a more fine-grained version of *wt-app-eff* for single instructions:

$$wt\text{-}instr\ p\ \varphi \equiv app\ p\ (\varphi!p) \wedge (\forall\,(q,t){\in}set(eff\ p\ (\varphi!p)).\ t \leq' \varphi!q)$$

With this, we get the following equality for *wt-method*:

$$wt\text{-}method\ \varphi = 0 < mpc \wedge map\ OK\ \varphi{\in}states \wedge wt\text{-}start\ \varphi \wedge (\forall\,p{<}mpc.\ wt\text{-}instr\ p\ \varphi)$$

It remains to lift welltypings from methods to programs. Welltypings of programs are functions Φ :: *prog-type* with

$$\textbf{types}\ prog\text{-}type = cname \Rightarrow sig \Rightarrow state\text{-}type\ option\ list$$

These functions return a welltyping for each method and each class in the program. A program is welltyped if there is a welltyping Φ such that *wt-jvm-prog* Γ Φ holds. This *wt-jvm-prog* is nothing else than *wf-prog* of Section 3.2.1 with *wt-method* for method bodies. It fills in the method context of Section 3.3.2 accordingly:

$$\begin{aligned}&wt\text{-}jvm\text{-}prog\ \Gamma\ \Phi \equiv wf\text{-}prog\ (\lambda\Gamma\ C'\ ((mn,ps),rt,(mxs,mxl,ins,et)).\\ &\qquad\qquad wt\text{-}method\ \Gamma\ C'\ mn\ ps\ rt\ mxs\ mxl\ ins\ et\ (\Phi\ C'\ (mn,ps)))\ \Gamma\end{aligned}$$

3.3.4 Executable Bytecode Verifiers

Section 3.3.3 defined welltypings for μJava. This section shows how to instantiate the two type inference algorithms of Section 2.4 and Section 2.5 to get executable bytecode verifiers for μJava.

With the semilattice as defined in Section 3.3.1 and the transfer function of Section 3.3.2, and still within the same method context as *wt-method*, we only need to provide the correct start value to Kildall's algorithm to get an executable BV:

$$\begin{array}{l} \textit{wt-kil} \equiv \textit{0} < \textit{size ins} \wedge \\ \qquad (\textit{let } s_0 = \textit{Some } ([],(\textit{OK } (\textit{Class } C'))\#(\textit{map OK ps})@(\textit{replicate mxl Err})); \\ \qquad\quad \varphi_0 = \textit{OK } s_0 \ \# \ (\textit{replicate } (\textit{size ins} - \textit{1}) \ (\textit{OK None})) \\ \qquad \textit{in } \forall\, n < \textit{size ins}.\ (\textit{kildall } \varphi_0)!n \neq \textit{Err}) \end{array}$$

Position *0* in φ_0 is the same as the start value in *wt-start*. Since we know nothing yet about the positions greater than *0*, we fill in the bottom element *OK None* for those.

Lifting to full programs and filling in the method context is the same as for *wt-method*:

$$\begin{array}{l} \textit{wt-jvm-prog}_k\ \Gamma \equiv \\ \textit{wf-prog}\ (\lambda\Gamma\ C'\ ((\textit{mn},\textit{ps}),\textit{rt},(\textit{mxs},\textit{mxl},\textit{ins},\textit{et})).\ \textit{wt-kil}\ \Gamma\ C'\ \textit{mn ps rt mxs mxl ins et})\ \Gamma \end{array}$$

This definition only gives us a working BV if *step* meets the conditions of Section 2.4.

Lemma 3.1 The transfer function *step*, built from *app* and *eff* as described in Section 2.3.3 and Section 3.3.2, is monotone, bounded, and type preserving (w.r.t. *states* and *size ins*).

Albeit large (a case distinction over the instruction set), the proof that *step* is monotone and type preserving is easy and mostly automatic. That *step* is bounded is checked explicitely by the *app* component of *step*.

Using Theorem 2.1, I have then proved the following:

Theorem 3.3 The executable BV is sound and recognizes all welltyped programs:

$$\textit{wt-jvm-prog}_k\ \Gamma = (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

To show that this verified BV is indeed executable, I have generated ML code from *wt-jvm-prog*$_k$ that verifies μJava bytecode programs.

The instantiation of the lightweight bytecode verifier is similar. Again we need to provide it with the correct start value.

$$\begin{array}{l}
\textit{wt-lbv} :: \textit{state-type option err cert} \Rightarrow \textit{bool} \\
\textit{wt-lbv}\ c \equiv \textit{check-cert}\ (\textit{size ins})\ c \wedge 0 < \textit{size ins} \wedge \\
\qquad (\textit{let}\ s_0 = \textit{OK}\ (\textit{Some}\ ([],(\textit{OK}\ (\textit{Class}\ C'))\#(\textit{map OK ps})@(\textit{replicate mxl Err}))) \\
\qquad \textit{in wtl}\ c\ s_0 \neq \textit{Err})
\end{array}$$

$$\begin{array}{l}
\textit{check-cert} :: \textit{nat} \Rightarrow \textit{state-type option err cert} \Rightarrow \textit{bool} \\
\textit{check-cert}\ n\ c \equiv c \in \textit{states} \wedge \textit{size}\ c = n{+}1 \wedge (\forall\, i{<}n.\ c!i \neq \textit{Err}) \wedge c!n = \textit{OK None}
\end{array}$$

The *check-cert* predicate ensures that the certificate is wellformed.

Certificates *prog-cert* = *cname* ⇒ *sig* ⇒ *state-type option err cert* are lifted to programs the same way method types are. Lifting *wt-lbv* to programs and filling in the method context is again standard:

$$\begin{array}{l}
\textit{wt-jvm-prog}_l :: \textit{jvm-prog} \Rightarrow \textit{prog-cert} \Rightarrow \textit{bool} \\
\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \equiv \textit{wf-prog}\ (\lambda\Gamma\ C'\ ((\textit{mn},\textit{ps}),\textit{rt},(\textit{mxs},\textit{mxl},\textit{ins},\textit{et})). \\
\qquad\qquad \textit{wt-lbv}\ \Gamma\ C'\ \textit{mn ps rt mxs mxl ins et}\ (\textit{Cert}\ C'\ (\textit{mn},\textit{ps})))\ \Gamma
\end{array}$$

Theorems 2.3 and 2.4, together with Lemma 3.1 and the semilattice construction in Section 3.3.1, tell us that the LBV is sound and complete for this type system.

Theorem 3.4 If the LBV accepts a program, it is welltyped:

$$\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \longrightarrow (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

Theorem 3.5 The LBV accepts every welltyped program:

$$\textit{wt-jvm-prog}\ \Gamma\ \Phi \longrightarrow \textit{wt-jvm-prog}_l\ \Gamma\ (\textit{mk-cert}\ \Phi)$$

The function *mk-cert* :: *prog-type* ⇒ *prog-cert* is the certificate as defined in Section 2.5.4 lifted to programs (for every C and *sig*, it selects the method type $\Phi\ C\ \textit{sig}$ and, as in Section 2.5.4, sets everything to *OK None* except jump targets).

As for $\textit{wt-jvm-prog}_k$, I have generated ML code from $\textit{wt-jvm-prog}_l$, showing that the LBV is fully executable.

3.4 Type Safety

This section is about the correctness, the type safety, of the welltyping specification above. It is split into three parts: part one, Section 3.4.1, presents the theorem; part two, Section 3.4.2, shows the conformance relation on which the proof of the type safety theorem builds; part three, Section 3.4.3, sketches the proof that conformance remains invariant during execution.

3.4.1 The Theorem

This section presents the type safety theorem. It states that the bytecode verifier is correct, that it guarantees safe execution. If the bytecode verifier succeeds and we start the program Γ in its canonical start state (see Section 3.2.4), the defensive μJVM will never return a type error. With Theorem 3.1, this implies that the checks of the defensive machine are redundant and the aggressive machine can be used safely instead.

Theorem 3.6 If C is a class in Γ with a *main* method, then

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

To prove this theorem, we set out from a program Γ for which the bytecode verifier returns true, i.e., for which there is a Φ such that *wt-jvm-prog* Γ Φ holds. The proof builds on the observation that all runtime states σ that conform to the types in Φ are type safe. For σ **conforms to** Φ, I 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 $(C,\textit{sig})$, then the state type $(\Phi\ C\ \textit{sig})!\textit{pc}$ must be of the form *Some s*, and for every value v on the stack or in the register set the type of v must be a subtype of the corresponding entry in its static counterpart s. Section 3.4.2 shows the complete formal definition of conformance. I have proved that conformance is invariant during execution if the program is welltyped. Section 3.4.3 sketches the proof.

Lemma 3.2 Conformance is invariant during execution in welltyped programs.

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

Lemma 3.2 is still not sufficient, though: it might be the case that *start* Γ C does not conform to Φ. Lemma 3.3 states that this is not so.

Lemma 3.3 If C is a class in Γ with a *main* method, then

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

Lemmas 3.2 and 3.3 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 3.6 is Lemma 3.4.

Lemma 3.4 An execution step started in a conformant state cannot produce a type error in welltyped programs.

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

The proof of Lemma 3.4 is by case distinction on the current instruction in σ. Similar to the proof of Lemma 3.2, the conformance relation together with the *app* part of *wt-jvm-prog* ensures *check-instr* in *exec-d* returns true. Because we know that all states during execution conform, we can conclude Theorem 3.6: there will be no type errors in welltyped programs.

3.4.2 Conformance

This section shows the formal definition of the conformance relation between dynamic JVM states and static types in the μJava model. For a simpler type system, it already appeared in [60] and [68].

For the proof of the invariance lemma (Lemma 3.2) to go through, the intuitive notion of conformance I have given above is not sufficient, the formal conformance relation $\Phi \vdash \sigma\surd$ is stronger. It describes in detail the states that can occur during execution, the form of the heap, and the form of the method invocation stack.

I begin with the heap: a heap conforms if all objects conform. An object conforms if all fields conform to their declared type. For the definition of single value conformance $hp \vdash v ::\preceq T$ see Section 3.2.4, p. 54.

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

$$\textit{oconf} :: \textit{aheap} \Rightarrow \textit{obj} \Rightarrow \textit{bool}$$
$$\textit{oconf hp}\ (C,\textit{fs}) \equiv \textit{lconf hp fs}\ (\textit{map-of}\ (\textit{fields}\ (\Gamma,C)))$$

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

This part of the conformance invariant ensures that instructions fetching from the heap (like *Getfield*) can only put type conforming values on the stack. Note that I still treat the program Γ as a global constant. In Isabelle, it is a parameter of the conformance relation.

Single value conformance is lifted to register set and stack by *approx-loc* and *approx-stk*. Any value conforms to the unusable type *Err*. This part of the conformance relation is the most obvious: it directly connects the types in the BV with the values at runtime.

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

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

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

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

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

This is still not enough. For the *Return* instruction, we also need information about the structure of the call frame stack. The predicate *correct-frames* below describes the structure of the call frame stack beneath the topmost frame. The parameters rt_0 and sig_0 are the return type and signature of the topmost frame.

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

In the definition above, a list of call frames conforms if it is empty. If it is not empty, the head frame is investigated more closely: the state type $\Phi\ C\ sig\ !\ pc$ for the current instruction must denote a reachable instruction ($= \textit{Some}\ \ldots$); the call frame must belong to a defined method; it must be halted at an *Invoke* instruction which created the call frame above (this is not easily expressed, but we can demand that *mn* and *ps* stem from sig_0, that the return type of a static lookup on C' conforms to the one from the frame above (rt_0), and that the current stack is large enough to hold the actual parameters plus the object on which the method was invoked); finally, the current frame and the rest of the call frame stack must conform. Remember that in the definition of *correct-frames* above, the only data from the top frame is sig_0 and rt_0. In the $f\#frs$ case, the parameter f is the frame below the top frame.

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

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

With *preallocated hp*, the invariant ensures that the special heap locations for system exceptions are allocated with the corresponding system exception objects.

Figure 3.11 is a snapshot of the μJVM state in the middle of a typical program execution. On the left there is the μJVM with its frame stack and heap, on the right there are the method types the BV predicted for this program. The program declarations appear on the lower right side in the static part.

The state on the left in Figure 3.11 conforms to the static type information on the right: all objects in the heap conform, because the values of the field f (declared in class B) are all of type *Class A* (*Null* is of type *Class A*, and the address *Addr 0* points to an object of *Class B* which is a subclass of *Class A*). All frames but the topmost one are halted at the *Invoke* instruction that created the next frame. The dynamic operand stacks conform to the static ones, because their length is the same and all values have conforming type. The topmost frame conforms, too, because its *pc* points to a valid instruction (*Getfield f B*), and the value on the dynamic operand stack is an address that points to an object of class *C*.

3.4.3 Invariance Proof

This section sketches the proof of the invariance lemma:

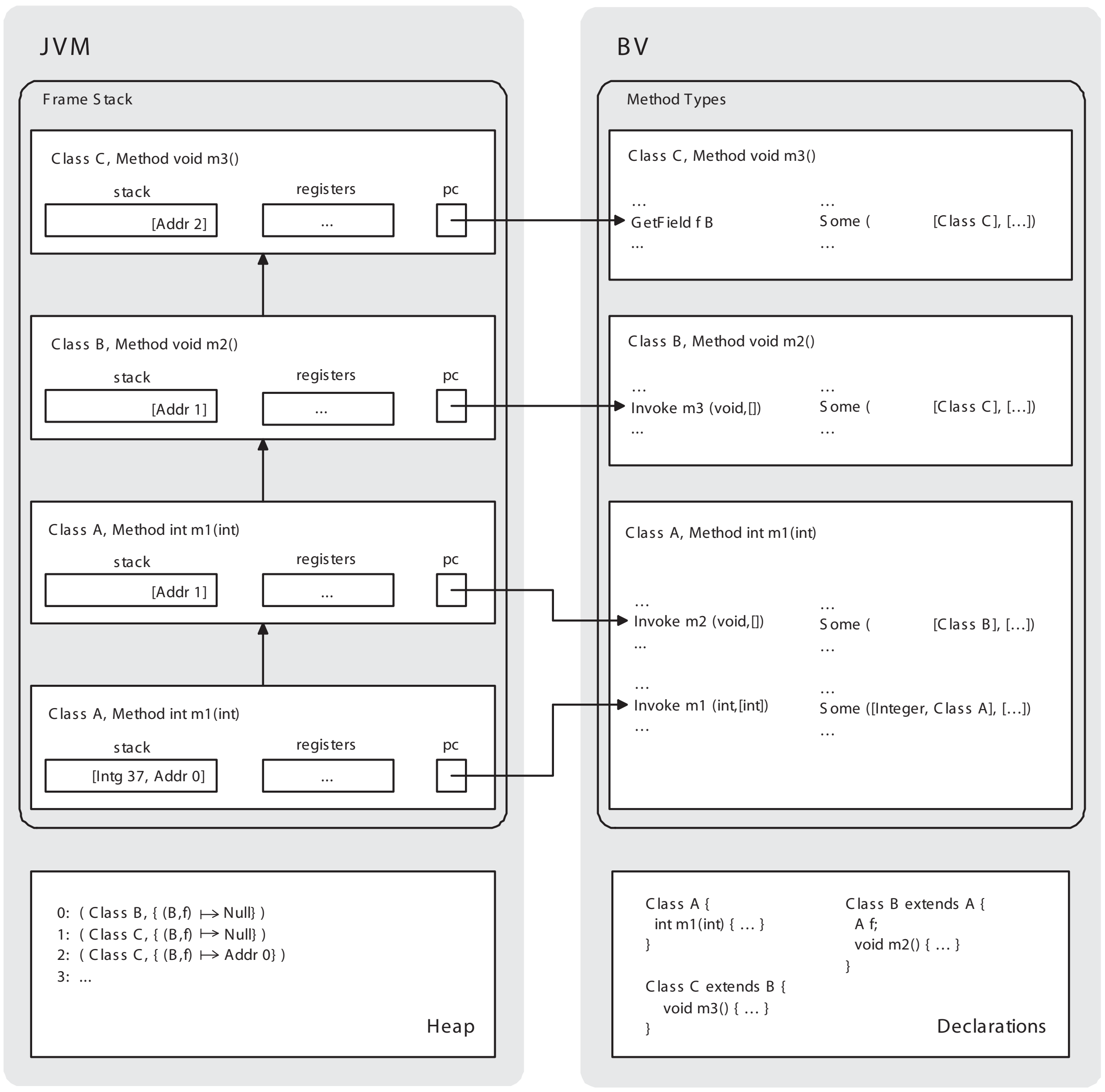
JVM
Frame Stack
Class C, Method void m3()
stack
[Addr 2]
registers
...
pc
Class B, Method void m2()
stack
[Addr 1]
registers
...
pc
Class A, Method int m1(int)
stack
[Addr 1]
registers
...
pc
Class A, Method int m1(int)
stack
[Intg 37, Addr 0]
registers
...
pc
0: (Class B, { (B,f) ↦ Null})
1: (Class C, { (B,f) ↦ Null})
2: (Class C, { (B,f) ↦ Addr 0})
3: ...
Heap
BV
Method Types
Class C, Method void m3()
GetField f B
Some ([Class C], [...])
Class B, Method void m2()
Invoke m3 (void,[])
Some ([Class C], [...])
Class A, Method int m1(int)
Invoke m2 (void,[])
Some ([Class B], [...])
Invoke m1 (int,[int])
Some ([Integer, Class A], [...])
Class A { int m1(int) { ... } }
Class B extends A { A f; void m2() { ... } }
Class C extends B { void m3() { ... } }
Declarations

Figure 3.11: μJVM execution snapshot.

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

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 conclude from the conformance of σ together with the *app* part of *wt-jvm-prog* that all assumptions of the operational semantics are met (like ”the stack is not empty”). Then we execute the instruction and observe that the new state τ conforms to the corresponding t in *eff pc s*.

The full Isabelle proof of the invariance lemma is about 1,600 lines (33 pages) long, including lemmas about the conformance relation. This is too large to show here, but since the structure of the proof is similar for each instruction, a typical example suffices: below, I show the Isabelle/Isar proof for the *New* instruction.

We are in the step case of the induction, we have to show that conformance is invariant if we make one single execution step, and we have made the case distinction on the instruction set to get to the *New* instruction. The global assumption *wt-jvm-prog* Γ is already decomposed into *wf-prog wt* Γ and *wt-instr pc* (Φ C' *sig*) (see Section 3.3.3). The method context is determined by *method* (Γ,C') *sig* = *Some* (C',*rt*,(*mxs*,*mxl*,*ins*,*et*)), the current state is (*None*, *hp*, (*stk*,*regs*,C',*sig*,*pc*)#*frs*), and assumption *conf* says that it conforms to Φ. Assumption *no-x* says that in this one-step execution no exception occurs (exceptions are handled in a separate lemma for all instructions at once).

lemma *New-correct*:
assumes *wf*: *wf-prog wt* Γ
assumes *meth*: *method* (Γ,C') *sig* = *Some* (C',*rt*,(*mxs*,*mxl*,*ins*,*et*))
assumes *ins*: *ins*!*pc* = *New C*
assumes *wt*: *wt-instr pc* (Φ C' *sig*)
assumes *exec*: *Some* τ = *exec* (Γ, *None*, *hp*, (*stk*,*regs*,C',*sig*,*pc*)#*frs*)
assumes *conf*: $\Phi \vdash$ (*None*, *hp*, (*stk*,*regs*,C',*sig*,*pc*)#*frs*)$\surd$
assumes *no-x*: *fst* (*exec-instr* (*ins*!*pc*) Γ *hp stk regs* C' *sig pc frs*) = *None*
shows $\Phi \vdash \tau\surd$

proof –

We begin the proof by decomposing the conformance relation $\Phi \vdash \ldots \surd$ of assumption *conf* into its parts. Since we know the method context from *meth*, the only new variables we get are *st* and *lt*, the static type information at Φ C' *sig*!*pc*:

from *conf meth*
obtain *st lt* **where**
heap-ok: *hp*$\surd$ **and** *prealloc*: *preallocated hp* **and**
phi-pc: Φ C' *sig*!*pc* = *Some* (*st*,*lt*) **and** *is-class-*C': *is-class* Γ C' **and**
frame: *correct-frame hp* (*st*,*lt*) *mxl ins* (*stk*, *regs*, C', *sig*, *pc*) **and**

frames: *correct-frames hp* Φ *rt sig frs*
by (*auto simp add*: *correct-state-def iff del*: *not-None-eq*)

With *wt-instr*, the BV gives us information about the effect of the instruction: *New* pushes an object of class C onto the stack. The BV also ensures the applicability of *New* in the current state: the stack has enough space, the new class C is declared in the program and the program counter *pc+1* is still safely within the method. Keep in mind that $\leq_s$ is the semilattice order on *state-type* (without the option layer):

from *phi-pc ins wt*
obtain *st′ lt′* **where**
is-class-C: *is-class* Γ C **and** *mxs*: *size st* $<$ *mxs* **and**
suc-pc: *pc+1* $<$ *size ins* **and** *phi-suc*: Φ C' *sig* ! (*pc+1*) = *Some* (*st′*, *lt′*) **and**
less: (*Class C* # *st*, *lt*) $\leq_s$ (*st′*, *lt′*)
by (*auto simp add*: *wt-instr-def eff-def norm-eff-def*)

The next lines turn to the operational semantics, the dynamic side. They exploit the fact that the execution step does not throw an exception, hence the *new-Addr* function in the *New* rule of *exec-instr* must have returned an unused location in the heap.

obtain *r xp′* **where** *new-Addr*: *new-Addr hp* = (*r*,*xp′*) **by** (*cases new-Addr hp*)
with *ins no-x* **obtain** *hp*: *hp r* = *None* **and** *xp′* = *None*
by (*auto dest*: *new-AddrD simp add*: *raise-system-xcpt-def*)

With this, we can write down the state after execution of *New*:

with *exec ins meth new-Addr*
have τ = (*None*, *hp*(*r*$\mapsto$*blank* Γ *C*), (*Addr r* # *stk*, *regs*, C', *sig*, *pc+1*) # *frs*)
(**is** τ = (*None*, *?hp′*, *?f* # *frs*)) **by** *simp*

Now that we know what τ looks like, we can begin a **moreover** chain that collects the parts we need to show $\Phi \vdash \tau\surd$ in the end.

moreover

Heap conformance is easy: a blank object only has default values in its fields that by construction all conform to their declared types. Since the rest of the heap has not changed, the new heap *?hp′* conforms.

from *wf hp heap-ok is-class-C* **have** *hp′*: *?hp′* $\surd$
by (*auto intro*!: *hconf-newref simp add*: *oconf-def dest*: *fields-is-type*)
moreover

Adding a new object of course also leaves the heap preallocated with system exception objects, if it was before.

from *hp prealloc* **have** *preallocated ?hp′* **by** (*rule preallocated-newref*)
moreover {

The *correct-frame* predicate is more involved. We mainly need to show *approx-stk* and *approx-loc* for the new state, the other two components we either already have ($pc+1 < size\ ins$) or they follow directly from *correct-frame* for the current state ($size\ regs=1+size\ (snd\ sig)+mxl$). We begin by decomposing *correct-frame* for the current state:

from *frame* **obtain**
stk: *approx-stk hp stk st* **and** *regs*: *approx-loc hp regs lt* **and**
len: $size\ regs = 1+size\ (snd\ sig)+mxl$
by (*clarsimp simp add*: *correct-frame-def*)

We then note that extending the heap with a new object cannot influence the conformance of objects in the old heap. I write $hp \leq| hp'$ if all addresses in *hp* point to objects of the same type as in hp'. Formally:

$$hp{\leq}|hp' \equiv \forall\, a\ C\ fs.\ hp\ a = Some(C,fs) \longrightarrow (\exists fs'.\ hp'\ a = Some(C,fs'))$$

Since the registers have not changed, and lt' is even more general than *lt*, we can conclude that *approx-loc* still holds:

from *hp* **have** *sup*: $hp \leq|\ ?hp'$ **by** (*rule hext-new*)
with *regs less wf* **have** *approx-loc ?hp′ regs lt′* **by** (*auto intro*: *approx-loc-sup-heap*)
moreover

The same line of thought applies to *approx-stk*. Here we first have to show that the new value on the stack (the address pointing to the new object) conforms to the value the BV predicted. Since it is a blank object of class *C*, this is easy.

have $?hp' \vdash Addr\ r\ ::\preceq\ Class\ C$ **by** (*simp add*: *conf-def*)

First lifting *approx-stk* to the new heap and then pushing the new values on both the dynamic and static side leaves us with:

with *sup stk* **have** *approx-stk ?hp′* (*Addr r* # *stk*) (*Class C* # *st*)
by (*auto intro*: *approx-stk-sup-heap*)

The widening step to st' brings us

with *less wf* **have** *approx-stk ?hp′* (*Addr r* # *stk*) *st′* **by** *auto*

and together with *approx-loc* we can conclude *correct-frame*:

  ultimately
  have *correct-frame ?hp′* (*st′*,*lt′*) *mxl ins ?f* **using** *suc-pc len*
    by (*clarsimp simp add*: *correct-frame-def*)
}

Up to this point, we have collected information about τ, about the new heap, and about *correct-frame*. We merely have *correct-frames* left, for which we repeat the reasoning from above: adding a new object to the heap does not change conformance of old values. Lemma *correct-frames-newref* lifts this by induction on the frame stack to *correct-frames*:

moreover
from *hp frames wf heap-ok is-class-C*
have *correct-frames ?hp′* Φ *rt sig frs* **by** (*auto intro*: *correct-frames-newref*)

All these taken together show $\Phi \vdash \tau\surd$:

ultimately
show *?thesis* **using** *meth is-class-C′ phi-suc* **by** (*simp add*: *correct-state-def*)
qed

This concludes the proof of Lemma *New-correct*.

The full invariance proof has a lemma like this for each instruction. The most involved ones are those for the *Invoke* and *Return* instructions, because these concern more than one frame stack. As mentioned above, the exception case is handled in a separate lemma for all instructions at once. It is subdivided into one case where the exception handler is in the same method, and into another case where the frame stack is searched recursively.

The uppermost lemma collects the results of all lemmas for single instructions and for the exception case, and performs the (trivial) induction on the execution sequence to conclude Lemma 3.2: conformance is invariant during execution.

3.5 Conclusion

In Chapter 3, I have shown how to instantiate the framework of Chapter 2 with a first simple type system for the μJVM. I have described in detail the formalization of the μJVM, a small, but representative subset of the JVM with objects, inheritance, virtual methods, and exception handling.

The type system is comparable to the one of Pusch [60, 68]. In addition, it supports exception handling, is formulated in a more modular and intuitive way, and, due to the abstract framework, gives rise not only to a description of welltyped programs, but to two different executable and verified bytecode verifiers (an iterative standard BV and a lightweight BV).

Specifications of operational semantics and algorithms should be executable. For validating the μJVM specification and its executability, I have generated ML code (using [9]) for the operational semantics of the μJVM, and for both bytecode verifiers.

In comparison to Pusch's theorem [68], the statement of the type safety theorem is clearer. The defensive μJVM makes it easier to see what exactly type safety guarantees. The proof internally still builds on an invariant argument. Because large parts of it are written in Isabelle/Isar, it is now possible to read, understand, and reproduce the argumentation in detail.

The complete μJava formalization with the type system for exception handling consists of about 11,100 lines of Isabelle code (245 pages). This does not include the source language.

The full specification and proofs are available as part of the Isabelle distribution [35] and from the VerifiCard project web site [89].

4 Object Initialization

Constructors, a simple concept in the source language, initialize objects by side effect. Java's security architecture relies on constructors following a certain protocol in that process. In this chapter, I formalize and prove correct a type system in the BV for object initialization. As the state of the initialization protocol is not directly observable from types and values alone, this complicates the JVM formalization moderately and the proof of type safety significantly.

4.1 Introduction

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

Why care at all about object initialization? Object initialization is not necessary for the type safety of the language, the default values of fields do nicely. After all, I have just proved in Chapter 3 that μJava is type safe and there was no mention of object initialization anywhere. The feature is interesting because large parts of Java's security mechanism depend on constructors being called before objects are used and on superclass constructors being called before intialization commences in a constructor. In this way, and together with access modifiers, secure, consistent object states can be guaranteed.

To make object initialization accessible to the type safety proof, I extend the operational semantics by a safety automaton [79] that stores and checks the initialization status of objects. If, for instance, a *Getfield* instruction accesses an uninitialized object, this safety automaton will raise an alarm. The type safety theorem states that no such alarms will be raised if the bytecode verifier accepts. The JVM specification is vague on what exactly object initialization means for each particular instruction at runtime. The defensive VM in this thesis makes this precise. If in doubt, I have used Sun's

pc	*instruction*	*opstack* (at runtime)
	...	
3	*New A*	[]
4	*Dup*	[*Addr 0*]
5	*New A*	[*Addr 0*, *Addr 0*]
6	*Dup*	[*Addr 1*, *Addr 0*, *Addr 0*]
7	*LitPush Null*	[*Addr 1*, *Addr 1*, *Addr 0*, *Addr 0*]
8	*Invoke-spcl A* (*init*, [*Class A*])	[*Null*, *Addr 1*, *Addr 1*, *Addr 0*, *Addr 0*]
9	*Invoke-spcl A* (*init*, [*Class A*])	[*Addr 1*, *Addr 0*, *Addr 0*]
	...	[*Addr 0*]

Figure 4.1: Object creation and constructor call for `new A(new A(null))`.

implementation as reference or chosen to be more strict in the safety automaton or more general in the BV than the JVM specification demands. The resulting BV accepts more programs than the one in the JVM specification, and it guarantees properties as least as strong as those required.

The JVM specification [51, pages 131–133] gives a short description of how to check for proper object initialization in the BV. Figure 4.1 is a typical piece of bytecode for the object creation/constructor call cycle as produced by common Java compilers for the source language expression `new A(new A(null))`.

The instruction sequence first allocates space for the still uninitialized object, and then duplicates the reference to that object (address *0*) for the constructor call. This process is repeated for the inner expression. After the constructors have been called, the result of the expression—a reference to the newly created and initialized object—is on top of the stack. In the process, two different uninitialized references to objects of class *A* have been on the stack.

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

As the name suggests, *UnInit* stands for uninitialized, freshly created objects. The reference on top of the stack after the first *New A* instruction would receive the type *UnInit A*.

After the constructor has been called on that reference, the object is initialized and we can replace *UnInit A* by the usual type *Class A*. However, as the JVM specification [51, pages 132–133] points out, this is not sufficient:

The instruction number needs to be stored as part of the special type, as

pc	*instruction*	*stack*
	…	
3	*New A*	[]
4	*Dup*	[*UnInit A 3*]
5	*New A*	[*UnInit A 3*,*UnInit A 3*]
6	*Dup*	[*UnInit A 5*,*UnInit A 3*,*UnInit A 3*]
7	*LitPush Null*	[*UnInit A 5*,*UnInit A 5*,*UnInit A 3*,*UnInit A 3*]
8	*Invoke-spcl A* (*init*, [*Class A*])	[*NT*,*UnInit A 5*,*UnInit A 5*,*UnInit A 3*,*UnInit A 3*]
9	*Invoke-spcl A* (*init*, [*Class A*])	[*Init A*,*UnInit A 3*,*UnInit A 3*]
	…	[*Init A*]

Figure 4.2: A welltyping with object initialization.

> there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. [...] When an instance initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the local variable array that are the same object as the class instance are replaced.

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

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

Figure 4.2 shows the stack part of a welltyping for the instructions of Figure 4.1 in the extended type system. Note how the types *UnInit A 3* and *UnInit A 5* keep track of the references *Addr 0* and *Addr 1* created in lines *3* and *5*. If there was only a simple *UnInit A* type, we would not be able to decide which reference the first *Invoke-spcl* (at $pc = 8$) initializes—all of them would have type *UnInit A*. We would mark all references as initialized and would thus miss that the reference *Addr 0* is not yet initialized at program counter *9*.

In order to include object initialization in the formalization, I extend the model of Chapter 3 in four steps: Section 4.2 begins by introducing formally the new types described above. Section 4.3 deals with the changes to the VM and the operational semantics. Section 4.4 instantiates the BV for object initialization, and Section 4.5 relates VM and type system in the type safety theorem.

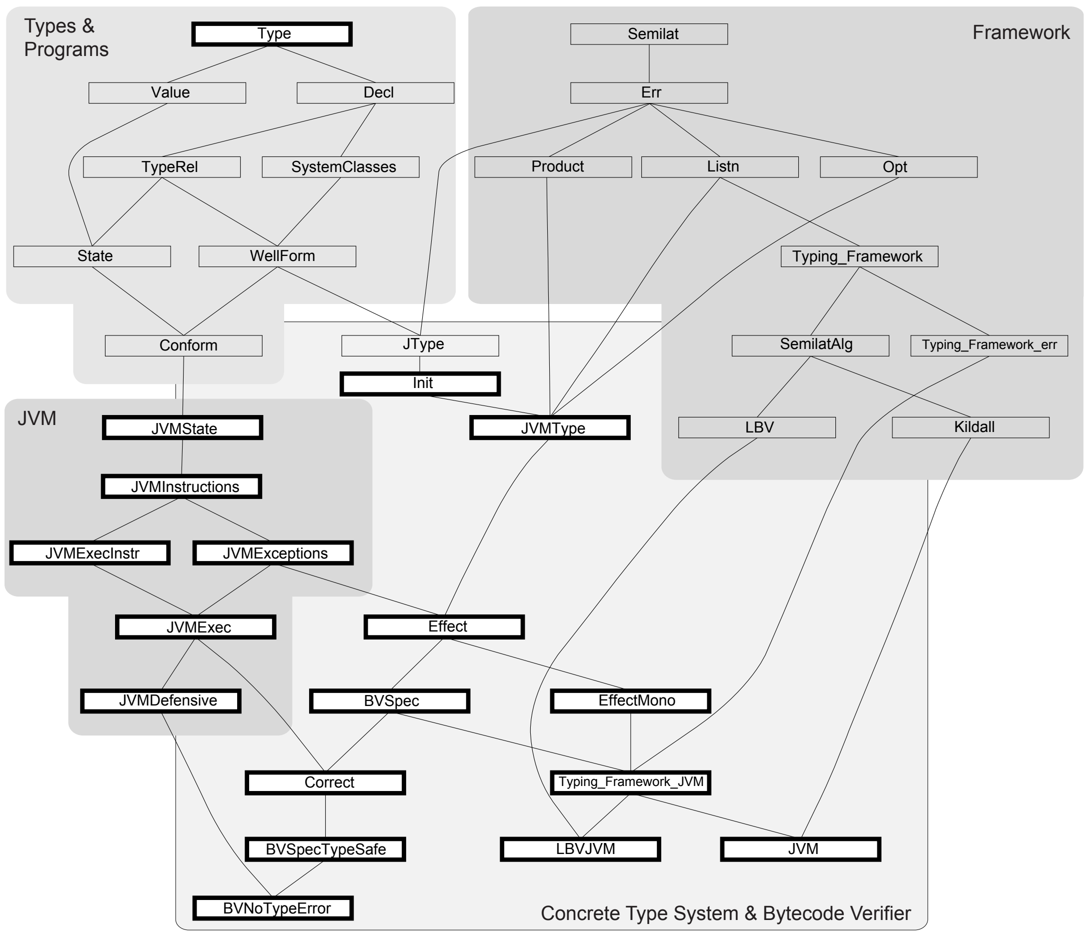

Figure 4.3: Including object initialization: overview.

Figure 4.3 gives an overview of which Isabelle theories are affected by the move to object initialization. The framework and program structure are unchanged. Although there is merely one new theory (*Init*), most modules in the JVM and the BV instantiation have undergone major changes (indicated by bold border).

4.2 Types

The following data type definition captures the intuition of Section 4.1 formally; the constructor *Init ty* stands for the normal, initialized μJava types.

datatype *init-ty* = *Init ty* | *UnInit cname nat* | *PartInit cname*

These initialization types are just an additional layer on top of the types *ty* of Chapter 3. Everything below *ty*—reference types and primitive types—remains as defined in Chapter 3.

The type system for initialization in the BV has *init-ty* entries on the stack and beneath the *OK*/*Err* structure in the register set:

types *state-type* = *init-ty list* × *init-ty err list*

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

types *state-bool* = *state-type* × *bool*

4.3 Operational Semantics

This section covers the changes to the μJVM that are needed to model constructor calls and their handling in the BV. The organization of this section is canonical: it starts out with the structure and state space of the new μJVM in Section 4.3.1; after that, it extends first the aggressive machine in Section 4.3.2 and then the defensive machine in Section 4.3.3.

4.3.1 Structure and State Space of the VM

What the μJVM of Chapter 3 lacks for object initialization are constructors. In the real JVM, constructors are methods with a special name `<init>` and no return value. Similarly, in μJava, constructors are ordinary methods with a special name *init*, but, like any other method, they may have a return value (which I have ignored in Figure 4.2).

We also need a bytecode instruction to call constructors: *Invoke-spcl*. Similar to the *Invoke* instruction, it has a class and a method signature as parameters. The method

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* (*ty list*)	invoke constructor
|	*Return*	return from method
|	*Dup*	duplicate top element
|	*Dup-x1*	duplicate top element and push 2 values down
|	*IAdd*	integer addition
|	*Goto int*	goto relative address
|	*Ifcmpeq int*	branch if equal
|	*Throw*	throw exception

Figure 4.4: The bytecode instruction set with constructors.

name is fixed to *init*.[1] In contrast to *Invoke*, it does not execute a virtual method call, but a static one. Figure 4.4 shows the new instruction set.

Since we want to prove something about if and how far objects are initialized, we need to observe the initialization status of individual objects. The current μJVM state does not allow this: from heap, stack, and registers, we cannot get any information on whether an object is initialized or not.

The standard solution for this kind of problem is a *safety automaton* [79]. If we view the standard operational semantics as an automaton, the safety automaton is an automaton that runs in parallel to the operational semantics. With each transition, it checks and carries with it the safety critical information that cannot be observed directly from the operational semantics alone.

For this safety automaton, I introduce a new, artificial component into the state: a second heap, which mirrors the real one in structure, that stores the initialization status using values of HOL type *init-ty*. Formally, the new component *iheap* is simply a function from locations to *init-ty*:

$$\textbf{types}\ \textit{iheap} = \textit{loc} \Rightarrow \textit{init-ty}$$

At this point, the class name parameter of *PartInit* comes into play: if an object gets the

[1] In the real JVM, *Invoke-spcl* is also used for calls to `private` and `super` methods and therefore has the method name as an additional parameter.

tag *PartInit C*, it means that the object is initialized up to class C in the class hierarchy, or, more precisely, that the constructor of C's superclass must be called before the fields of the object can be accessed.

As it is a safety automaton only, this new component of the μJVM state merely plays the role of an auxiliary variable for the type safety proof and does not alter the observable behaviour of the μJVM other than possibly raising an alarm.

Unfortunately, recording the initialization status of objects induces another problem: in the type safety proof, we relied on a large invariant which requires that objects, once allocated, do not change their type on the heap. All changes to the heap from hp to hp' had to satisfy $hp \le| hp'$ for the invariant to be preserved (see also the example for the *New* instruction in the type safety proof in Section 3.4.3, p. 74). Values may change, of course, and new objects may be created, but the type information of existing objects must not change. This will also apply to the new *iheap* component. On the other hand, observing the changes during the lifetime of objects is the very purpose of *iheap*, so we need to accommodate it somehow. I use the solution Freund proposes [26]: the invariant may not allow type changes, but it does allow the creation of new objects. For each constructor call I therefore create a new blank object that is a copy of the object before, only the initialization status tag gets updated. When a constructor is finished, it replaces the uninitialized reference in the calling method by the new, now initialized object.

This sounds like a significant modification, but on closer inspection there is no observational difference in executions between creating new objects (and then discarding the superfluous ones) and the standard semantics: the initialization process is basically a chain of constructor calls along the class hierarchy. It is finished when the constructor of class *Object* has been called. Before that, in between constructor calls, the object is inaccessible. During the whole process, all fields retain their default value, and no method other than the constructor can be invoked on it. Once the end of the chain is reached with the constructor of class *Object*, the program can only work on the last allocated object. Because each constructor replaces the object of the calling constructor, all intermediate objects are discarded.

The solution outlined above does not impress by its beauty, but it seems to be the only practical one for an invariant proof. Below, I finish the formal definition of the new μJVM's state space, give an example of a constructor call chain in Figure 4.5, and discuss some alternatives to the solution presented here.

In order to replace the correct reference when a constructor is finished, we extend the definition of frames from Section 3.2.2 by a pair of references. In this pair, we store the reference the current constructor has been called to initialize and the reference to the initialized object. Since there will be no initialized object in the beginning, we will store

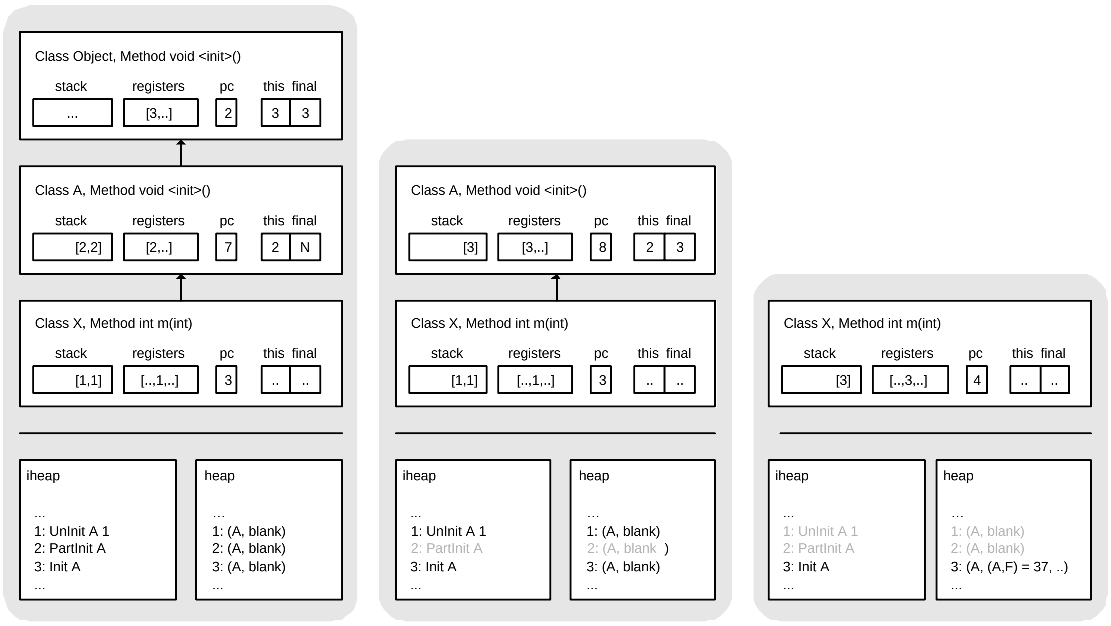

Figure 4.5: Constructor call chain, three stages.

Null in that case. To avoid annoying HOL type conversions, we work with values *val* instead of references (although we will only be using addresses and *Null*). So *frame* is now:

$$\textbf{types}\ \mathit{frame} = \mathit{opstack} \times \mathit{registers} \times \mathit{cname} \times \mathit{sig} \times \mathit{nat} \times (\mathit{val} \times \mathit{val})$$

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

$$\textbf{types}\ \mathit{jvm\text{-}state} = \mathit{xcpt\ option} \times \mathit{aheap} \times \mathit{iheap} \times \mathit{frame\ list}$$

Figure 4.5 illustrates the constructor call chain in three stages.

In stage one, on the left, method *m* in class *X* has created a new object of class *A* at address *1* which the *iheap* tags as uninitialized. The constructors of class *A* and class *Object* have been called, but they have not returned yet. Note that the reference update pair *this*/*final* is not used in method *m*. For the constructor of class *A*, a new copy of the object has been created at address *2*. The *iheap* tags it as *PartInit A*, because we are in the *A*-constructor. The *this* component stores *Addr 2*. Even though this information is usually also present in register *0*, it is necessary to store it separately: the program might overwrite register *0* and try to return without having called the superclass constructor. The *final* component is not known yet and still set to *Null*. The chain ends with the constructor of *Object*: the copy of the object that has been created for the *Object* constructor (at address *3*) is tagged as initialized.

In stage two, in the middle of Figure 4.5, the constructor of class *Object* has just returned. The program counter of the *A*-constructor was incremented, the reference *Addr 2* replaced by *Addr 3*, and *Addr 3* was stored in the *final* component to indicate that this is the initialized object. The *A*-constructor is now allowed to return normally. There now exist no more references that point to the object at *Addr 2*. Note that the object at *Addr 1* cannot be changed either, because *Addr 1* does not occur in the top frame.

In stage three, on the right of Figure 4.5, the constructor of class *A* has returned and the initialization process is complete. The program counter of the frame for method *m* has been increased by one, and the reference *Addr 1* has been replaced by *Addr 3* (pointing to the now fully initialized object). Note that the *A*-constructor might already have accessed fields and methods of the object between stages two and three. All references to the objects at *Addr 1* and *Addr 2* have been deleted.

The new-objects-semantics of [26] which I use here deviates from the standard formulation of the JVM semantics (although it is equivalent in observable behaviour), therefore I explore some alternatives in the following.

We want the *iheap* to contain precise information about the initialization status. This is its purpose in the safety automaton, so there is not much we can hope to change in this respect.

An alternative would be for the invariant to allow the *iheap* to change in a controlled way: a subtyping scheme in the invariant that would allow the same object to slowly change its type from uninitialized to initialized. That, however, would not be enough. If we only know that this subtyping invariant holds (and the BV accepted), we could not conclude that the object has a specific initialization status at a certain position in the program (which we need in the type safety proof to show that the safety automaton does not raise an alarm). Depending on how we define this subtyping, the information would either say the object is at most initialized up to some point, or it is at least initialized up to some point. Neither is sufficient: for the *Invoke-spcl* instruction, we need to know that the object is at most initialized up to some class *A* (because it is not allowed to call a constructor on an initialized object), and for the *Return* instruction, it is necessary that the object is at least initialized up to some *A* (because before that, a constructor must not return normally).

Since the new-objects-semantics of [26] is equivalent to the normal semantics, there must exist an invariant that can cope with a changing *iheap*. And indeed, if the invariant varies the constraints on the *iheap* relative to the current position in the frame stack, it might be possible to succeed with it. For each frame, the invariant would count how many frames above it initialize the same object (including a distinction if the frame already has called the superclass constructor, waits for the superclass constructor to return, or if the superclass constructor already has returned). Call that number of currently active initializers n. Then the dynamic tag of the objects must be exactly n steps further up in the initialization hierarchy than the statically predicted type. This would mean that the conformance of each single frame depends on the state and conformance of a number of other frames above it in a nontrivial manner. The relatively local statement that each frame is halted at the *Invoke* instruction which created the frame directly above (see also Section 3.4.2, p. 69) already lead to an involved definition. A global dependency like that would complicate the invariant and the proof of type safety significantly.

The new-objects-semantics shifts some of the complexity of the invariant and the proof to the operational semantics. The change in the semantics is not negligible, but the standard semantics can easily be seen as an optimization of the new-objects-semantics. As the invariant and the type safety proof are already quite involved in this simplified version, the new-object-semantics seems to be the more practical approach. A formal proof of equivalence between standard and new semantics would also involve an invariant, but this invariant should be simpler than the one outlined above, because it does not need to involve a static type system.

4.3.2 Aggressive Machine

This section describes the operational semantics of the μJVM with object initialization.

The only interesting changes (compared to Chapter 3) occur in the definition of *exec-instr* in Figure 4.6, and the only interesting instructions there are *New*, *Invoke-spcl*, and *Return*.

The definitions of *exec* and $\xrightarrow{\text{jvm}}$ remain the same, the only change is that *exec* passes the new *ihp* and z parameters on to *exec-instr*.

The rules for the other instructions (apart from the three mentioned above) get two new parameters, *ihp* for initialization status and z for the reference update in constructors, but they simply pass them on. The rule for *Load* for instance looks like this:

$$\begin{aligned}&\textit{exec-instr}\ (\textit{Load idx})\ \textit{hp ihp stk regs}\ C'\ \textit{sig pc z frs} =\\ &\qquad\qquad (\textit{None},\ \textit{hp},\ \textit{ihp},\ ((\textit{regs}\ !\ \textit{idx})\ \#\ \textit{stk},\ \textit{regs},\ C',\ \textit{sig},\ \textit{pc}{+}1,\ z)\#\textit{frs})\end{aligned}$$

```
exec-instr :: instr ⇒ aheap ⇒ iheap ⇒ opstack ⇒ registers ⇒ cname ⇒ sig ⇒ nat ⇒
               val × val ⇒ frame list ⇒ jvm-state

exec-instr (Load idx) hp ihp stk regs C′ sig pc z frs =
           (None, hp, ihp, ((regs ! idx) # stk, regs, C′, sig, pc+1, z)#frs)
exec-instr (Store idx) hp ihp stk regs C′ sig pc z frs =
           (None, hp, ihp, (tl stk, regs[idx:=hd stk], C′, sig, pc+1, z)#frs)
exec-instr (LitPush v) hp ihp stk regs C′ sig pc z frs =
           (None, hp, ihp, (v # stk, regs, C′, sig, pc+1, z)#frs)
exec-instr (New C) hp ihp stk regs C′ sig pc z frs =
           let (ref,xp′) = new-Addr hp;
               hp′ = if xp′=None then hp(ref ↦ blank Γ C) else hp;
               ihp′ = if xp′=None then ihp(ref := UnInit C pc) else ihp′;
               pc′ = if xp′=None then pc+1 else pc
           in (xp′, hp′, ihp′, ((Addr ref) # stk, regs, C′, sig, pc′, z)#frs)
exec-instr (Getfield F C) hp ihp stk regs C′ sig pc z frs =
           let r = hd stk;
               xp′ = raise-system-xcpt (r=Null) NullPointer;
               (c,fs) = the(hp(the-Addr r));
               pc′ = if xp′=None then pc+1 else pc
           in (xp′, hp, ihp, (the(fs(F,C))#(tl stk), regs, C′, sig, pc′, z)#frs)
exec-instr (Putfield F C) hp ihp stk regs C′ sig pc z frs =
           let (v,r)= (hd stk, hd(tl stk));
               xp′ = raise-system-xcpt (r=Null) NullPointer;
               a = the-Addr r;
               (c,fs) = the (hp a);
               hp′ = if xp′=None then hp(a↦(c,fs((F,C)↦v))) else hp;
               pc′ = if xp′=None then pc+1 else pc
           in (xp′, hp′, (tl (tl stk), regs, C′, sig, pc′, z)#frs)
exec-instr (Checkcast C) hp ihp stk regs C′ sig pc z frs =
           let r = hd stk;
               xp′ = raise-system-xcpt (¬ cast-ok Γ C hp r) ClassCast;
               stk′ = if xp′=None then stk else tl stk;
               pc′ = if xp′=None then pc+1 else pc
           in (xp′, hp, ihp, (stk′, regs, C′, sig, pc′, z)#frs)
exec-instr (Invoke C mn ps) hp ihp stk regs C′ sig pc z frs =
           let n = size ps; args = take n stk; r = stk!n;
               xp′ = raise-system-xcpt (r=Null) NullPointer;
               dt = fst(the(hp(the-Addr r)));
               (dc,-,-,mxl,-)= the (method (Γ,dt) (mn,ps));
               frs′ = if xp′≠None then [] else
                      [([],(rev args)@[r]@replicate mxl arbitrary,dc,(mn,ps),0,arbitrary)]
           in (xp′, hp, ihp, frs′@(stk, regs, C′, sig, pc, z)#frs)
```

Figure 4.6: Single step execution (part 1).

```
exec-instr (Invoke-spcl C ps) hp ihp stk regs C' sig pc z frs =
        let n = size ps; args = take n stk; ref = stk!n;
            $x_1$ = raise-xcpt (ref=Null) NullPointer;
            D = fst(the(hp (the-Addr ref)));
            (dc,_,_, mxl,_) = the (method (Γ,C) (init,ps));
            (a',$x_2$) = new-Addr hp;
            xp'= if $x_1$ = None then $x_2$ else $x_1$;
            hp' = hp(a' ↦ blank Γ D);
            T = if C = Object then Init (Class D) else PartInit C;
            z' = if C = Object then (Addr a', Addr a') else (Addr a', Null);
            frs' = if xp'≠None then [] else
                  [([],(Addr a')#(rev args)@(replicate mxl arbitrary),dc,(init,ps),0,z')]
        in (xp', hp', ihp(a':= T), frs'@(stk, regs, C', sig, pc, z)#frs)
exec-instr Return hp ihp $stk_0$ regs C' $sig_0$ pc $z_0$ frs =
        if frs=[] then (None, hp, ihp, []) else
        let (stk,regs,C,sig,pc,z) = hd frs;
            v = hd $stk_0$; (mn,ps) = $sig_0$; (a,b) = $z_0$;
            n = size ps; addr = stk!n;
            drpstk = drop (n+1) stk;
            stk' = if mn=init then v#(replace addr b drpstk) else v#drpstk;
            regs' = if mn=init then replace addr b regs else regs;
            z' = if mn=init ∧ z = (addr,Null) then (addr,b) else z
        in (None, hp, ihp, (stk',regs',C,sig,pc+1,z')#tl frs)
exec-instr Pop hp ihp stk regs C' sig pc z frs =
        (None, hp, ihp, (tl stk, regs, C', sig, pc+1, z)#frs)
exec-instr Dup hp ihp stk regs C' sig pc z frs =
        (None, hp, ihp, (hd stk # stk, regs, C', sig, pc+1, z)#frs)
exec-instr Dup-x1 hp ihp stk regs C' sig pc z frs =
      (None, hp, ihp, (hd stk#hd (tl stk)#hd stk#(tl (tl stk)), regs, C', sig, pc+1, z)#frs)
exec-instr IAdd hp ihp stk regs C' sig pc z frs =
        let (i1,i2) = (the-Intg (hd stk), the-Intg (hd (tl stk)))
        in (None, hp, ihp, (Intg (i1+i2)#(tl (tl stk)), regs, C', sig, pc+1, z)#frs)
exec-instr (Ifcmpeq b) hp ihp stk regs C' sig pc z frs =
        let ($v_1$,$v_2$) = (hd stk, hd (tl stk));
            pc' = if $v_1$ = $v_2$ then nat(int pc+b) else pc+1
        in (None, hp, ihp, (tl (tl stk), regs, C', sig, pc', z)#frs)
exec-instr (Goto b) hp ihp stk regs C' sig pc z frs =
        (None, hp, ihp, (stk, regs, C', sig, nat(int pc+b), z)#frs)
exec-instr Throw hp ihp stk regs C' sig pc z frs =
        let xp = raise-system-xcpt (hd stk = Null) NullPointer;
            xp' = if xp = None then Some (hd stk) else xp
        in (xp', hp, ihp, (stk, regs, C', sig, pc, z)#frs)
```

Figure 4.7: Single step execution (part 2).

For the *New* instruction, we have to record that freshly created objects are uninitialized. Apart from that, everything remains the same (see also Section 3.2.3, p. 51):

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

The definition for *Invoke-spcl* is new:

```
exec-instr (Invoke-spcl C mn ps) hp ihp stk regs C' sig pc z frs =
let         n = size ps;
         args = take n stk;
          ref = stk!n;
          x_1 = raise-xcpt (ref=Null) NullPointer;
            D = fst(the(hp the-Addr ref));
  (dc, _, _, mxl, _) = the (method (Γ,C) (mn,ps));
     (a',x_2) = new-Addr hp;
          xp' = if x_1 = None then x_2 else x_1;
          hp' = hp(a' ↦ blank Γ D);
            T = if C = Object then Init (Class D) else PartInit C;
           z' = if C = Object then (Addr a', Addr a') else (Addr a', Null);
         frs' = if xp'≠None then [] else
                  [([],(Addr a')#(rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,z')]
in  (xp', hp', ihp(a':= T), frs'@(stk, regs, C', sig, pc, z)#frs)
```

The beginning is the same as in the normal *Invoke*: in *args*, we store the actual parameters of the constructor call. The reference on which to invoke the constructor is the next element on the stack after the parameters. If it is *Null*, a *NullPointer* exception is thrown. The rest is different: we retrieve the dynamic type D of the object the reference *ref* points to and do a static method lookup with the parameters C (the class) and *ps* (the list of parameter types) of the *Invoke-spcl* instruction. We then create a new object: as in the *New* instruction, we request a free location, handle the *OutOfMemory* exception (should one occur), and assign a blank object to the new address with the same dynamic type as the one at *ref* (in effect, we copy the object at *ref* to the new address *Addr a'*).

Now to the new initialization status *ihp*$(a' := T)$: the new object gets the tag *PartInit* C, because the constructor for class C has just been invoked. The next constructor must be one in the superclass of C. If C has no superclass (if $C = Object$), we have reached the end of the constructor chain and can tag the new object as initialized. After this, its fields and methods are accessible.

The reference update pair z' in the constructor frame is new. It stores two components: the *this* pointer of the new constructor, and the reference of the initialized object. The initialized object will replace *ref* in the current frame when the constructor is finished. The *this* pointer is the new *Addr a'*. The initialized object we do not know yet, so we set it to *Null*. Only if we have reached *Object*, we know that the newly created object is also the final one that will replace all intermediate objects in the call chain. With these values, we construct the call frame of the new constructor the same way the normal *Invoke* does.

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

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

Let's first take another look at the parameters: stk_0, sig_0, and z_0 are the stack, the signature, and the reference update pair of the current call frame. As in Section 3.2.3, we extract the return value *val*, the name *mn*, and the list of formal parameter types *ps* of the method we are currently executing. We then drop the actual parameters from the stack in the caller frame. If the current frame does not belong to a constructor (if $mn \neq init$), we simply put the return value on top of the stack in the caller frame and are done. If we are returning from a constructor, however, we use *replace* to substitute the now initialized object b (the second component of the reference update pair) for the address of the original object *addr* (see also the example in Figure 4.5, p. 84). This replacement must occur everywhere on the stack and in the register set of the caller frame to delete all references to the original object. If the caller frame belongs to a constructor that is initializing the same object as we are, we also have to modify the second component of its reference update pair z to the now initialized object b.

4.3.3 Defensive Machine

As in the aggressive machine, *check*, *exec-d*, and $\xrightarrow{\text{djvm}}$ remain the same; they merely pass on the new components *ihp* and z that are used in *check-instr*.

Figure 4.8 shows the new definition of *check-instr*. It implements the checking part of the safety automaton: for each object access, apart from constructor calls, it requires that the object is tagged as initialized. This concerns the *Getfield*, *Putfield*, *Checkcast*, *Invoke*, and *Throw* instructions. For *Invoke*, it additionally checks if all parameters are initialized.

As the *Invoke-spcl* instruction is new, we will take a closer look at it:

```
$\mathit{check\text{-}instr}\ (\mathit{Invoke\text{-}spcl}\ C\ ps)\ hp\ ihp\ stk\ regs\ Cl\ sig\ pc\ z\ frs =$
  $\mathit{size}\ ps < \mathit{size}\ stk\ \wedge$
  $(\mathit{let}\ n = \mathit{size}\ ps;\ v = stk!n\ \mathit{in}\ \mathit{is\text{-}Ref}\ v\ \wedge$
  $(v \neq \mathit{Null} \longrightarrow hp\ (\mathit{the\text{-}Addr}\ v) \neq \mathit{None} \wedge \mathit{method}\ (\Gamma,C)\ (\mathit{init},ps) \neq \mathit{None}\ \wedge$
                        $\mathit{fst}\ (\mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{init},ps))) = C\ \wedge$
                        $\mathit{list\text{-}all2}\ (\lambda v\ T.\ hp \vdash v ::\preceq T \wedge \mathit{is\text{-}init}\ hp\ ihp\ v)\ (\mathit{rev}\ (\mathit{take}\ n\ stk))\ ps\ \wedge$
                        $(\mathit{case}\ ihp\ (\mathit{the\text{-}Addr}\ v)\ \mathit{of}\ \mathit{Init}\ T \Rightarrow \mathit{False}$
                                                  $|\ \mathit{UnInit}\ C'\ pc' \Rightarrow C' = C$
                                                  $|\ \mathit{PartInit}\ C' \Rightarrow C' = Cl \wedge (C',C) \in \mathit{subcls}\ \Gamma))$
```

Most of it is again the same as *Invoke*: the stack must contain initialized parameters of the correct type (the *list-all2* line) and it must contain the object reference on which to invoke the constructor. Contrary to *Invoke*, the constructor that is called must be defined exactly in the class the instruction statically assumes, and not in a subclass. Finally, the object itself must not be tagged as initialized. For uninitialized tags, the tag must match the constructor to be called: if the tag is *UnInit* C' *pc*, we must call the constructor of class C'; for classes that are tagged as partly initialized up to C', we must call the superclass constructor of C'.

For the *Return* instruction, *check-instr* in Figure 4.9 now requires that the return value be initialized. The last line in the *Return* rule is new:

$$(\mathit{fst}\ \mathit{sig}_0 = \mathit{init} \longrightarrow \mathit{snd}\ z_0 \neq \mathit{Null} \wedge \mathit{is\text{-}Ref}\ (\mathit{snd}\ z_0) \wedge \mathit{is\text{-}init}\ hp\ ihp\ (\mathit{snd}\ z_0))$$

It means that, if *Return* is leaving a constructor, the reference that is passed up the constructor chain ($\mathit{snd}\ z_0$) points to an initialized object.

$$\begin{array}{l}
\textit{check-instr} :: \textit{instr} \Rightarrow \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{opstack} \Rightarrow \textit{registers} \Rightarrow \textit{cname} \Rightarrow \textit{sig} \Rightarrow \textit{nat} \Rightarrow \\
\qquad\qquad\quad \textit{val} \times \textit{val} \Rightarrow \textit{frame list} \Rightarrow \textit{bool}
\end{array}$$

$$\begin{array}{ll}
\textit{check-instr}\ (\textit{Load idx})\ \textit{hp ihp stk regs Cl sig pc z frs} & = \textit{idx} < \textit{size regs} \\
\textit{check-instr}\ (\textit{Store idx})\ \textit{hp ihp stk regs Cl sig pc z frs} & = 0 < \textit{size stk} \wedge \textit{idx} < \textit{size regs} \\
\textit{check-instr}\ (\textit{LitPush}\ v)\ \textit{hp ihp stk regs Cl sig pc z frs} & = \neg \textit{is-Addr}\ v \\
\textit{check-instr}\ (\textit{New}\ C)\ \textit{hp ihp stk regs Cl sig pc z frs} & = \textit{is-class}\ \Gamma\ C
\end{array}$$

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Getfield}\ F\ C)\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\quad 0 < \textit{size stk} \wedge \textit{is-class}\ \Gamma\ C \wedge \textit{field}\ (\Gamma,C)\ F \neq \textit{None} \wedge \\
\quad (\textit{let}\ (C',T) = \textit{the}\ (\textit{field}\ (\Gamma,C)\ F);\ v = \textit{hd stk} \\
\quad \textit{in}\ C' = C \wedge \textit{is-Ref}\ v \wedge \\
\qquad\quad (v \neq \textit{Null} \longrightarrow\ \textit{hp}\ (\textit{the-Addr}\ v) \neq \textit{None} \wedge \textit{is-init hp ihp}\ v \wedge \\
\qquad\qquad\qquad\qquad (\textit{let}\ (D,\textit{fs}) = \textit{the}\ (\textit{hp}\ (\textit{the-Addr}\ v)) \\
\qquad\qquad\qquad\qquad\ \ \textit{in}\ (D,C) \in (\textit{subcls}\ \Gamma)^* \wedge \textit{fs}\ (F,C) \neq \textit{None} \wedge \textit{hp} \vdash \textit{the}\ (\textit{fs}\ (F,C)) ::\preceq T)))
\end{array}$$

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Putfield}\ F\ C)\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\quad 1 < \textit{size stk} \wedge \textit{is-class}\ \Gamma\ C \wedge \textit{field}\ (\Gamma,C)\ F \neq \textit{None} \wedge \\
\quad (\textit{let}\ (C',\ T) = \textit{the}\ (\textit{field}\ (\Gamma,C)\ F);\ v = \textit{hd stk};\ v = \textit{hd}\ (\textit{tl stk})\ \textit{in} \\
\quad C' = C \wedge \textit{is-init hp ihp}\ v \wedge \textit{is-Ref}\ v \wedge \\
\quad (v \neq \textit{Null} \longrightarrow \textit{hp}\ (\textit{the-Addr}\ v) \neq \textit{None} \wedge \textit{is-init hp ihp}\ v \wedge \\
\qquad\qquad\qquad (\textit{let}\ (D,\textit{fs}) = \textit{the}\ (\textit{hp}\ (\textit{the-Addr}\ v))\ \textit{in}\ (D,C) \in (\textit{subcls}\ \Gamma)^* \wedge \textit{hp} \vdash v ::\preceq T)))
\end{array}$$

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Checkcast}\ C)\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\quad 0 < \textit{size stk} \wedge \textit{is-class}\ \Gamma\ C \wedge \textit{is-Ref}\ (\textit{hd stk}) \wedge \textit{is-init hp ihp}\ (\textit{hd stk})
\end{array}$$

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Invoke}\ C\ \textit{mn ps})\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\quad \textit{size ps} < \textit{size stk} \wedge \textit{mn} \neq \textit{init} \wedge \\
\quad (\textit{let}\ n = \textit{size ps};\ v = \textit{stk}!n\ \textit{in is-Ref}\ v \wedge \\
\quad (v \neq \textit{Null} \longrightarrow \textit{hp}\ (\textit{the-Addr}\ v) \neq \textit{None} \wedge \textit{is-init hp ihp}\ v \wedge \\
\qquad\qquad\qquad\quad \textit{method}\ (\Gamma,\textit{cname-of hp}\ v)\ (\textit{mn},\textit{ps}) \neq \textit{None} \wedge \\
\qquad\qquad\qquad\quad \textit{list-all2}\ (\lambda v\ T.\ \textit{hp} \vdash v ::\preceq T \wedge \textit{is-init hp ihp}\ v)\ (\textit{rev}\ (\textit{take}\ n\ \textit{stk}))\ \textit{ps}))
\end{array}$$

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Invoke-spcl}\ C\ \textit{ps})\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\quad \textit{size ps} < \textit{size stk} \wedge \\
\quad (\textit{let}\ n = \textit{size ps};\ v = \textit{stk}!n\ \textit{in is-Ref}\ v \wedge \\
\quad (v \neq \textit{Null} \longrightarrow \textit{hp}\ (\textit{the-Addr}\ v) \neq \textit{None} \wedge \textit{method}\ (\Gamma,C)\ (\textit{init},\textit{ps}) \neq \textit{None} \wedge \\
\qquad\qquad\qquad\quad \textit{fst}\ (\textit{the}\ (\textit{method}\ (\Gamma,C)\ (\textit{init},\textit{ps}))) = C \wedge \\
\qquad\qquad\qquad\quad \textit{list-all2}\ (\lambda v\ T.\ \textit{hp} \vdash v ::\preceq T \wedge \textit{is-init hp ihp}\ v)\ (\textit{rev}\ (\textit{take}\ n\ \textit{stk}))\ \textit{ps} \wedge \\
\qquad\qquad\qquad\quad (\textit{case ihp}\ (\textit{the-Addr}\ v)\ \textit{of Init}\ T \Rightarrow \textit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{UnInit}\ C'\ \textit{pc}' \Rightarrow C' = C \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{PartInit}\ C' \Rightarrow C' = \textit{Cl} \wedge (C',C) \in \textit{subcls}\ \Gamma))
\end{array}$$

Figure 4.8: The defensive μJVM with initialization checks (part 1).

$$
\begin{array}{l}
\textit{check-instr Return hp ihp stk}_0 \textit{ regs Cl sig}_0 \textit{ pc } z_0 \textit{ frs} = \\
\quad 0 < \textit{size stk}_0 \wedge \\
\quad (0 < \textit{size frs} \longrightarrow \textit{method } (\Gamma,\textit{Cl}) \textit{ sig}_0 \neq \textit{None} \wedge \\
\qquad (\textit{let } v = \textit{hd stk}_0;\ (C,\textit{rt},b) = \textit{the } (\textit{method } (\Gamma,\textit{Cl}) \textit{ sig}_0) \\
\qquad\ \textit{in Cl} = C \wedge \textit{hp} \vdash v ::\preceq \textit{rt} \wedge \textit{is-init hp ihp } v) \wedge \\
\qquad (\textit{fst sig}_0 = \textit{init} \longrightarrow \textit{snd } z_0 \neq \textit{Null} \wedge \textit{is-Ref } (\textit{snd } z_0) \wedge \textit{is-init hp ihp } (\textit{snd } z_0)))
\end{array}
$$

$$
\begin{array}{ll}
\textit{check-instr Pop hp ihp stk regs Cl sig pc z frs} & = 0 < \textit{size stk} \\
\textit{check-instr Dup hp ihp stk regs Cl sig pc z frs} & = 0 < \textit{size stk} \\
\textit{check-instr Dup-x1 hp ihp stk regs Cl sig pc z frs} & = 1 < \textit{size stk} \\
\textit{check-instr IAdd hp ihp stk regs Cl sig pc z frs} & = 1 < \textit{size stk} \wedge \textit{is-Intg } (\textit{hd stk}) \wedge \\
 & \quad \textit{is-Intg } (\textit{hd } (\textit{tl stk})) \\
\textit{check-instr } (\textit{Ifcmpeq } b) \textit{ hp ihp stk regs Cl sig pc z frs} & = 1 < \textit{size stk} \wedge 0 \leq \textit{int pc}+b \\
\textit{check-instr } (\textit{Goto } b) \textit{ hp ihp stk regs Cl sig pc z frs} & = 0 \leq \textit{int pc}+b \\
\textit{check-instr Throw hp ihp stk regs Cl sig pc z frs} & = 0 < \textit{size stk} \wedge \textit{is-Ref } (\textit{hd stk}) \wedge \\
 & \quad \textit{is-init hp ihp } (\textit{hd stk})
\end{array}
$$

Figure 4.9: The defensive μJVM with initialization checks (part 2).

The canonical start state now includes an otherwise undefined *iheap* which marks the preallocated system exception objects as initialized. The information on which addresses are allocated for objects is already encoded in the normal heap. The reference update pair in the call frame is only used in constructors. Since the main method is a normal method, we can use a default value *arbitrary*:

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

Theorem 3.1, which states that defensive and aggressive VM commute if there are no type errors, also holds for the new VM:

Theorem 4.1 One-step execution in aggressive and defensive machines commutes if there are no type errors.

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

This concludes the formalization of the μJVM with object initialization.

4.4 Bytecode Verification

The new types for object initialization are already defined in Section 4.2, so we now only have to turn them into a semilattice (Section 4.4.1) and change the transfer function accordingly to instantiate the BV framework (Section 4.4.2). The instantiation again results in two executable bytecode verifiers in Section 4.4.3.

4.4.1 The Semilattice

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

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

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

$$\begin{array}{l} \textit{init-tys} \equiv \{\textit{Init } x \mid x.\ x \in (\textit{types } \Gamma)\} \cup \{x.\ \exists\, C\ n.\ x = \textit{UnInit } C\ n\} \cup \\ \qquad\qquad \{x.\ \exists\, C.\ x = \textit{PartInit } C\} \end{array}$$

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

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

Lemma 4.1 If Γ is well structured, then *Init.esl* is an err-semilattice and its order satisfies the ascending chain condition.

The proof builds on Theorem 3.2. The new initialiaztion layer adds case distinctions for *init-ty* on top.

If we repeat the construction from Section 3.3, we get:

$$\begin{array}{ll} \textit{stk-esl} :: \textit{nat} \Rightarrow \textit{ty list esl} & \textit{reg-sl} :: \textit{nat} \Rightarrow \textit{ty err list sl} \\ \textit{stk-esl mxs} \equiv \textit{upto-esl mxs } (\textit{Init.esl}) & \textit{reg-sl mxr} \equiv \textit{Listn.sl mxr } (\textit{Err.sl } (\textit{Init.esl})) \end{array}$$

$$\begin{array}{l} \textit{sl} :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool option err sl} \\ \textit{sl mxs mxr} \equiv \textit{Err.sl}(\textit{Opt.esl}(\textit{Product.esl } (\textit{Product.esl} \\ \qquad\qquad (\textit{stk-esl mxs})\ (\textit{Err.esl}(\textit{reg-sl mxr})))\ (\textit{Triv.esl}::\textit{bool esl}))) \end{array}$$

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

Lemma 4.2 If Γ is well structured, then *sl* is a semilattice and its order satisfies the ascending chain condition.

The proof uses Lemmas 2.1 to 2.9 from the framework as in Section 3.3.1.

With this, the semilattice construction is complete.

4.4.2 Applicability and Effect

The second ingredient to the BV is the transfer function. In Figures 4.10 and 4.12, I define *app'* and *eff'* for the new type system. As before, they do not involve the *bool* and *option* component yet, and they are defined in the same method context as the one in Chapter 3:

Γ	:: *program*	the program,
C'	:: *cname*	the class the method we are verifying is declared in,
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.

Compared to the original version in Figures 3.8 (p. 61) and 3.10 (p. 63), both definitions have become a bit larger, but remained the same in structure. Both are again subdivided into one case for exceptional and one case for normal execution.

I begin with the normal, non-exception case of applicability. The definition of *app'* in Figure 4.10 is large, but compared to Figure 3.8 the changes are few: *Load*, *Store*, *LitPush*, *Dup* and *Goto* are the same. *Getfield*, *Putfield*, *Checkcast*, *IAdd*, *Ifcmpeq*, *Throw*, and *Return* restrict their arguments to initialized types.

For the *New* instruction, *app'* now additionally checks that *UnInit C pc* is not part of the stack to make sure that the type *UnInit C pc* is not assigned to two different objects:

$$\mathit{app}'\ (\mathit{New}\ C,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) = \mathit{is\text{-}class}\ \Gamma\ C \land \mathit{size}\ \mathit{st} < \mathit{mxs} \land \mathit{UnInit}\ C\ \mathit{pc} \notin \mathit{set}\ \mathit{st}$$

Note that *app'* does not test the register set for *UnInit C pc* as Freund's type system [26, 27] does. If it did, the transfer function would not be monotone: raising the type in a

$$
\begin{array}{ll}
\mathit{app}' :: \mathit{instr} \times \mathit{nat} \times \mathit{state\text{-}type} \Rightarrow \mathit{bool} & \\
\mathit{app}'\ (\mathit{Load}\ \mathit{idx},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{lt} \wedge \mathit{lt}!\mathit{idx} \neq \mathit{Err} \wedge \\
 & \quad \mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Store}\ \mathit{idx},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size}\ \mathit{lt} \\
\mathit{app}'\ (\mathit{LitPush}\ v,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{st} < \mathit{mxs} \wedge \mathit{typeof}\ v \neq \mathit{None} \\
\mathit{app}'\ (\mathit{Getfield}\ F\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge\ t \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,\ t')) \\
\mathit{app}'\ (\mathit{Putfield}\ F\ C,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,t') \wedge \\
 & \quad t_2 \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge t_1 \preceq_i \mathit{Init}\ t') \\
\mathit{app}'\ (\mathit{New}\ C,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{size}\ \mathit{st} < \mathit{mxs} \wedge \\
 & \quad \mathit{UnInit}\ C\ \mathit{pc} \notin \mathit{set}\ \mathit{st} \\
\mathit{app}'\ (\mathit{Checkcast}\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge (\exists\, r.\ t = \mathit{Init}\ (\mathit{RefT}\ r)) \\
\mathit{app}'\ (\mathit{Dup},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{1}{+}\mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Dup\text{-}x1},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{2}{+}\mathit{size}\ \mathit{st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{IAdd},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \wedge t_1 = \mathit{Init}\ (\mathit{PrimT}\ \mathit{Integer}) \\
\mathit{app}'\ (\mathit{Ifcmpeq}\ b,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \vee (\exists\, r\ r'.\ t_1 = \mathit{Init}\ (\mathit{RefT}\ r) \wedge \\
 & \quad t_2 = \mathit{Init}\ (\mathit{RefT}\ r')) \\
\mathit{app}'\ (\mathit{Goto}\ b,\ \mathit{pc},\ s) & = \mathit{True} \\
\mathit{app}'\ (\mathit{Return},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = t \preceq_i \mathit{Init}\ \mathit{rt} \\
\mathit{app}'\ (\mathit{Throw},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \exists\, r.\ t = \mathit{Init}\ (\mathit{RefT}\ r) \\
\mathit{app}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st} \wedge \mathit{mn} \neq \mathit{init} \wedge \\
 & \quad \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \wedge \\
 & \quad \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st}); \\
 & \quad \quad t\ \ = \mathit{st}!\mathit{size}\ \mathit{ps} \\
 & \quad \mathit{in}\ t \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \quad \quad (\forall\, (a,f) {\in} \mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq_i \mathit{Init}\ f) \\
\mathit{app}'\ (\mathit{Invoke\text{-}spcl}\ C\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st} \wedge \\
 & \quad (\exists\, r.\ \mathit{method}\ (\Gamma,C)\ (\mathit{init},\mathit{ps}) = \mathit{Some}\ (C,r)) \wedge \\
 & \quad \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st}); \\
 & \quad \quad t\ \ = \mathit{st}!\mathit{size}\ \mathit{ps} \\
 & \quad \mathit{in}\ \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \quad \quad ((\exists\, \mathit{pc}.\ t = \mathit{UnInit}\ C\ \mathit{pc}) \vee \\
 & \quad \quad t = \mathit{PartInit}\ C' \wedge (C',C) \in \mathit{subcls}\ \Gamma) \wedge \\
 & \quad \quad (\forall\, (a,f) {\in} \mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq_i (\mathit{Init}\ f)) \\
\mathit{app}'\ (i,\ \mathit{pc},\ s) & = \mathit{False}
\end{array}
$$

Figure 4.10: Applicability of instructions with object initialization.

register from *UnInit C pc* to *Err* might make the instruction applicable and would lower the type of the result of *step* from *Err* to something different. Here, the *eff′* function below will take care of the register set. Sun's JVM specification solves the problem by disallowing all backwards jumps as long as there is any *UnInit* type anywhere on the stack or in the register set. This is correct, but somewhat drastic: although the restriction is not severe for programming in Java, it rejects an unnecessarily large number of type safe programs. At the same time it is hard to reason about, because it makes assumptions about the data flow analysis itself, and not only about properties of the resulting welltyping.

Normal method invocation in *app′* is restricted to initialized objects (for the parameters as well as for the object on which to invoke the method). Additionally, the method must not be a constructor.

Finally, there is the new rule for *Invoke-spcl*:

$$\begin{aligned}
&\mathit{app}'\ (\mathit{Invoke\text{-}spcl}\ C\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) = \mathit{size}\ \mathit{ps} < \mathit{size}\ \mathit{st}\ \wedge\\
&\quad (\exists\, r.\ \mathit{method}\ (\Gamma,C)\ (\mathit{init},\mathit{ps}) = \mathit{Some}\ (C,r))\ \wedge\\
&\quad \mathit{let}\ \mathit{as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size}\ \mathit{ps})\ \mathit{st});\\
&\qquad t = \mathit{st}!\mathit{size}\ \mathit{ps}\\
&\quad \mathit{in}\ \mathit{is\text{-}class}\ \Gamma\ C\ \wedge\\
&\qquad ((\exists\, \mathit{pc}.\ t = \mathit{UnInit}\ C\ \mathit{pc})\ \vee\\
&\qquad t = \mathit{PartInit}\ C'\ \wedge\ (C',C) \in \mathit{subcls}\ \Gamma)\ \wedge\\
&\qquad (\forall\, (a,f){\in}\mathit{set}(\mathit{zip}\ \mathit{as}\ \mathit{ps}).\ a \preceq_i (\mathit{Init}\ f))
\end{aligned}$$

It is similar to *Invoke*. Since it is supposed to be a static method invocation, the method dictionary *method* must yield an entry telling us that the method is defined in class C (and not in a superclass of C). The rule ensures that the type t of the object on which to invoke the constructor is either completely uninitialized, or partly initialized. If it is uninitialized, it must be of type *UnInit C pc* (for some *pc*)—only then are we allowed to invoke the C constructor. If it is partly initialized, it must be of type *PartInit* C' (C' is the class we are currently verifying). This is because, if it is partly initialized, we are verifying a constructor (only there may partly initialized objects occur), and it must be initialized up to exactly C', because for any class D the type *PartInit* D may only occur in D constructors. The only thing a constructor may do with partly initialized objects is invoke the superclass constructor on them—so C must be the direct superclass of C'. The JVM specification also allows to call another constructor of the same class, not only one of the superclass. In practice, this is convenient, in the formalization it would just add one more uninteresting case (where $C{=}C'$) to all proofs about *Invoke-spcl*.

This concludes the normal, non-exception case of *app*. The exception case is canonical, in *xcpt-names* (Figure 4.11) *Invoke-spcl* is treated like *Invoke*:

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

Figure 4.11: Exception names with *Invoke-spcl*.

$$\begin{array}{l}
\textit{xcpt-app} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{bool} \\
\textit{xcpt-app}\ i\ pc \equiv \forall\, C{\in}\textit{set}(\textit{xcpt-names}\ (i,pc,et)).\ \textit{is-class}\ \Gamma\ C
\end{array}$$

In order to build the final applicability function *app*, we have to lift *app′* to the *bool* component and to the *option* type: for *None*, we again get *True*, for *Some* first *app′*, *xcpt-app*, and the code boundary check must be satisfied, then we have two additional conditions on the superclass-constructor-has-been-called marker z:

$$\begin{array}{l}
\textit{app} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool option} \Rightarrow \textit{bool} \\
\textit{app}\ i\ pc\ s \equiv \textit{case}\ s\ \textit{of None} \Rightarrow \textit{True} \mid \textit{Some}\ t \Rightarrow \\
\textit{let}\ ((st,lt),z) = t\ \textit{in} \\
\quad \textit{app}'\ (i,pc,(st,lt)) \wedge \textit{xcpt-app}\ i\ pc \wedge (\forall\, (pc',s'){\in}\textit{set}\ (\textit{eff}\ i\ pc\ s).\ pc'{<}\textit{mpc}) \wedge \\
\quad (\textit{mn} = \textit{init} \longrightarrow \\
\qquad (i = \textit{Return} \longrightarrow z) \wedge (\forall\, C\ p.\ i = \textit{Invoke-spcl}\ C\ p \wedge st!\textit{size}\ p = \textit{PartInit}\ C' \longrightarrow \neg z))
\end{array}$$

If we are verifying a constructor ($mn = init$), then at each *Return* instruction the marker must be *True*, and at each *Invoke-spcl* for partly initialized objects the marker must be *False* (because we must call the superclass constructor only once).

In *eff′* (Figure 4.12), the instructions *Load*, *Store*, *Putfield*, *Ifcmpeq*, *Goto*, *Return*, and *Dup* remain unchanged; the instructions *LitPush*, *Getfield*, *Checkcast*, *IAdd*, and *Invoke* now explicitly yield initialized values. The instructions *Invoke-spcl* and *New* are more interesting.

Invoke-spcl is similar to *Invoke*, but it can only be used on uninitialized references (which is checked in *app*). After the constructor returns normally, the reference will be fully initialized, so the *Invoke-spcl* rule replaces the uninitialized type t with an initialized one of the same class (*theClass* :: *ini-ty* ⇒ *ty* satisfies *theClass* (*PartInit* C) = *Class* C and

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{eff}\,' :: \textit{instr} \times \textit{nat} \times \textit{state-type} \Rightarrow \textit{state-type}} \\
\textit{eff}\,' \ (\textit{Load idx},\ \textit{pc},\ (\textit{st},\ \textit{lt})) & = (\textit{ok-val}\ (\textit{lt}!\textit{idx})\#\textit{st},\ \textit{lt}) \\
\textit{eff}\,' \ (\textit{Store idx},\ \textit{pc},\ (t\#\textit{st},\ \textit{lt})) & = (\textit{st},\ \textit{lt}[\textit{idx}{:=}\ \textit{OK}\ t]) \\
\textit{eff}\,' \ (\textit{LitPush}\ v,\ \textit{pc},\ (\textit{st},\ \textit{lt})) & = (\textit{Init}\ (\textit{the}\ (\textit{typeof}\ v))\#\textit{st},\ \textit{lt}) \\
\textit{eff}\,' \ (\textit{Getfield}\ F\ C,\ \textit{pc},\ (t\#\textit{st},\ \textit{lt})) & = (\textit{Init}\ (\textit{snd}\ (\textit{the}\ (\textit{field}\ (\Gamma,C)\ F)))\#\textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{Putfield}\ F\ C,\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (\textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{New}\ C,\ \textit{pc},\ (\textit{st},\textit{lt})) & = (\textit{UnInit}\ C\ \textit{pc}\#\textit{st},\ \textit{replace}\ (\textit{OK}\ (\textit{UnInit}\ C\ \textit{pc}))\ \textit{Err}\ \textit{lt}) \\
\textit{eff}\,' \ (\textit{Checkcast}\ C,\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = (\textit{Init}\ (\textit{Class}\ C)\ \#\ \textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{Dup},\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = (t\#t\#\textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{Dup-x1},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (t_1\#t_2\#t_1\#\textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{IAdd},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (\textit{Init}\ (\textit{PrimT Integer})\#\textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{Ifcmpeq}\ b,\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (\textit{st},\textit{lt}) \\
\textit{eff}\,' \ (\textit{Goto}\ b,\ \textit{pc},\ s) & = s \\
\textit{eff}\,' \ (\textit{Invoke}\ C\ \textit{mn}\ \textit{ps},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{let}\ \textit{st}' = \textit{drop}\ (1{+}\textit{size ps})\ \textit{st}; \\
& \qquad (\_,\textit{rt},\_,\_,\_) = \textit{the}\ (\textit{method}\ (\Gamma,C)\ (\textit{mn},\textit{ps})) \\
& \quad\ \textit{in}\ (\textit{Init}\ \textit{rt}\#\textit{st}',\ \textit{lt}) \\
\textit{eff}\,' \ (\textit{Invoke-spcl}\ C\ \textit{ps},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{let}\ t = \textit{st}!\textit{size ps};\ i = \textit{Init}\ (\textit{theClass}\ t); \\
& \qquad \textit{st}'' = \textit{drop}\ (1{+}\textit{size ps})\ \textit{st}; \\
& \qquad \textit{st}' = \textit{replace}\ t\ i\ \textit{st}''; \\
& \qquad \textit{lt}' = \textit{replace}\ (\textit{OK}\ t)\ (\textit{OK}\ i)\ \textit{lt}; \\
& \qquad (\_,\textit{rt},\_,\_,\_) = \textit{the}\ (\textit{method}\ (\Gamma,C)\ (\textit{init},\textit{ps})) \\
& \quad\ \textit{in}\ (\textit{Init}\ \textit{rt}\#\textit{st}',\ \textit{lt}')
\end{array}
$$

Figure 4.12: Effect of instructions on the state type with object initialization.

theClass (*UnInit C pc*) = *Class C*). As in the operational semantics, the replacement happens everywhere in the stack and registers.

The *New* instruction seems easy at first: if a *New C* is at position *pc*, it produces the type *UnInit C pc*.

The rule in Figure 4.12 does more: for the alias analysis to be correct, there must not be any former instances of the type *UnInit C pc* with the same *pc* on the stack or in the registers. The *replace* in *eff'* takes care of the register set while the check for *UnInit C pc* in *app'* takes care of the stack.

This is also the solution used by Stärk et al. [83]; it yields a monotone transfer function, admits more programs than Freund's rule, and is still safe. In fact, here *replace* is the identity function, because the first path to *New* in the data flow analysis cannot contain *UnInit C pc* (because only the instruction at *pc* creates the type *UnInit C pc*). A merge with subsequent paths that might contain *UnInit C pc* would already yield *Err* without *replace*. This fact is difficult to prove formally, because it again involves reasoning about the data flow analysis (*paths*); just looking at the welltyping is not enough.

It remains to lift *eff'* to *state-bool* and then further to *state-bool option*. The purpose of the *bool* part in the state type is to mark whether a constructor has already called its superclass constructor. We set it to *True* when we call *Invoke-spcl* for a partly initialized type (*app* checks that it is the superclass constructor), otherwise we leave it untouched:

$$
\begin{array}{l}
\textit{eff-bool} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool} \Rightarrow \textit{state-bool}\\
\textit{eff-bool i pc} \ ((\textit{st},\textit{lt}),z) \equiv\\
(\textit{eff}' \ (i,\textit{pc},(\textit{st},\textit{lt})),\ \textit{if} \ \exists\, C\ p\ D.\ i = \textit{Invoke-spcl}\ C\ p \wedge \textit{st}!\textit{size}\ p = \textit{PartInit}\ D\ \textit{then True else}\ z)
\end{array}
$$

$$
\begin{array}{l}
\textit{norm-eff} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool option} \Rightarrow \textit{state-bool option}\\
\textit{norm-eff i pc} \equiv \textit{option-map}\ (\textit{eff-bool i pc})
\end{array}
$$

This concludes the case for normal execution. Exceptional execution may assume that the exception object is initialized; the superclass-marker z is merely passed through. Other than that, it remains the same:

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

Note that constructors that do not return normally (but by an exception) are accounted for: the (due to the exception) not fully initialized object will not be used, because the stack is cleared and the registers remain unchanged. They still contain the uninitialized type, because there is no replacement with *Init*.

The definition of *succs* is the same as in Section 3.3.2; *Invoke-spcl* is caught by the fall back clause *pc+1*. If we apply *norm-eff* to every successor instruction pc' and append the effect for the exception case, we arrive at the final effect function *eff*:

$$
\begin{array}{l}
\textit{eff} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool option} \Rightarrow (\textit{nat} \times \textit{state-bool option})\ \textit{list}\\
\textit{eff i pc s} \equiv (\textit{map}\ (\lambda \textit{pc}'.\ (\textit{pc}',\ \textit{norm-eff i pc s}))\ (\textit{succs i pc}))\ @\ (\textit{xcpt-eff i pc s})
\end{array}
$$

4.4.3 Executable Bytecode Verifiers

The definition of welltypings *wt-method*, which serves as the basis for the type safety proof and the executable bytecode verifiers, remains unchanged, only *wt-start* has to be adjusted. The changes to *wt-method* take place in the underlying semilattice and transfer function.

$$\begin{aligned}
\mathit{wt\text{-}start}\ \varphi \equiv\ & \mathit{let}\ t = \mathit{OK}\ (\mathit{if}\ mn = \mathit{init} \wedge C' \neq \mathit{Object}\ \mathit{then}\ \mathit{PartInit}\ C'\ \mathit{else}\ \mathit{Init}\ (\mathit{Class}\ C'));\\
& \qquad s_0 = \mathit{Some}\ (([],t\#(\mathit{map}\ (\mathit{OK} \circ \mathit{Init})\ ps)@(\mathit{replicate}\ mxl\ \mathit{Err})),C'{=}\mathit{Object});\\
& \mathit{in}\ s_0 \leq' \varphi!0
\end{aligned}$$

$$\mathit{wt\text{-}method}\ \varphi \equiv 0 < mpc \wedge \mathit{map}\ \mathit{OK}\ \varphi \in \mathit{states} \wedge \mathit{wt\text{-}start}\ \varphi \wedge \mathit{wt\text{-}app\text{-}eff}\ \varphi$$

In the definition of *wt-start*, the *this* pointer t (register 0) is more complicated than before: if we verify a constructor (and if it is not the constructor of class *Object*), then the *this* pointer is only partly initialized yet—the superclass constructor has to be called before it can be used. Otherwise, if it is a normal method or if it is class *Object*, we may assume that *this* points to an initialized object. There is only one new thing in the rest of the start value: we may also assume that the parameters only contain initialized values. As in Chapter 3, the $\leq' :: \mathit{state\text{-}bool\ option\ ord}$ is the semilattice order on the *option* level, and *states* is the carrier set of the semilattice.

Lifting to programs and filling in the method context is unchanged:

$$\begin{aligned}
&\mathit{wt\text{-}jvm\text{-}prog}\ \Gamma\ \Phi \equiv \mathit{wf\text{-}prog}\ (\lambda\Gamma\ C'\ ((mn,ps),rt,(mxs,mxl,ins,et)).\\
&\qquad\qquad \mathit{wt\text{-}method}\ \Gamma\ C'\ mn\ ps\ rt\ mxs\ mxl\ ins\ et\ (\Phi\ C'\ (mn,ps)))\ \Gamma
\end{aligned}$$

The instantiation of Kildall's algorithm is standard; we just use the new start value instead of the old one:

$$\begin{aligned}
&\mathit{wt\text{-}kil} \equiv 0 < \mathit{size}\ \mathit{ins}\ \wedge\\
&\mathit{let}\ t = \mathit{OK}\ (\mathit{if}\ mn = \mathit{init} \wedge C' \neq \mathit{Object}\ \mathit{then}\ \mathit{PartInit}\ C'\ \mathit{else}\ \mathit{Init}\ (\mathit{Class}\ C'));\\
&\quad s_0 = \mathit{Some}\ (([],t\#(\mathit{map}\ (\mathit{OK} \circ \mathit{Init})\ ps)@(\mathit{replicate}\ mxl\ \mathit{Err})),C'{=}\mathit{Object});\\
&\quad \varphi_0 = (\mathit{OK}\ S_0)\#(\mathit{replicate}\ (\mathit{size}\ \mathit{ins}{-}1)\ (\mathit{OK}\ \mathit{None}))\\
&\mathit{in}\ \forall\, n < \mathit{size}\ \mathit{ins}.\ (\mathit{kiljvm}\ \varphi_0)!n \neq \mathit{Err}
\end{aligned}$$

$$\begin{aligned}
&\mathit{wt\text{-}jvm\text{-}prog}_k\ \Gamma \equiv\\
&\mathit{wf\text{-}prog}\ (\lambda\Gamma\ C'\ ((mn,ps),rt,(mxs,mxl,ins,et)).\ \mathit{wt\text{-}kil}\ \Gamma\ C'\ mn\ ps\ rt\ mxs\ mxl\ ins\ et)\ \Gamma
\end{aligned}$$

By case distinction over the instruction set we get:

Lemma 4.3 The transfer function *step*, built from *app* and *eff* as described in Section 2.3.3 and Section 4.4.2, is monotone, bounded, and type preserving (w.r.t. *states* and *size ins*).

With Theorem 2.1 follows soundness and completeness.

Theorem 4.2 The executable BV is sound and recognizes all welltyped programs:

$$\textit{wt-jvm-prog}_k\ \Gamma = (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

The lightweight bytecode verifier is straightforward, too:

```
$\textit{wt-lbv} :: \textit{state-type option err cert} \Rightarrow \textit{bool}$
$\textit{wt-lbv}\ c \equiv \textit{check-cert}\ (\textit{size ins})\ c \wedge 0 < \textit{size ins} \wedge$
        $\textit{let}\ t = \textit{OK}\ (\textit{if}\ mn = \textit{init} \wedge C' \neq \textit{Object then PartInit}\ C'\ \textit{else Init}\ (\textit{Class}\ C'));$
            $s_0 = \textit{OK}\ (\textit{Some}\ (([],t\#(\textit{map}\ (\textit{OK}\circ\textit{Init})\ ps)@(\textit{replicate mxl Err})),C'{=}\textit{Object}));$
        $\textit{in wtl}\ c\ s_0 \neq \textit{Err})$
```

```
$\textit{wt-jvm-prog}_l :: \textit{jvm-prog} \Rightarrow \textit{prog-cert} \Rightarrow \textit{bool}$
$\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \equiv \textit{wf-prog}\ (\lambda\Gamma\ C'\ ((mn,ps),rt,(mxs,mxl,ins,et)).$
                    $\textit{wt-lbv}\ \Gamma\ C'\ mn\ ps\ rt\ mxs\ mxl\ ins\ et\ (\textit{Cert}\ C'\ (mn,ps)))\ \Gamma$
```

As before, the *check-cert* predicate ensures that the certificate is wellformed.

Theorems 2.3 and 2.4, together with Lemma 4.3 and the semilattice construction in Section 4.4.1, give us that the LBV is sound and complete for this type system.

Theorem 4.3 If the LBV accepts a program, it is welltyped:

$$\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \longrightarrow (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

Theorem 4.4 The LBV accepts every welltyped program:

$$\textit{wt-jvm-prog}\ \Gamma\ \Phi \longrightarrow \textit{wt-jvm-prog}_l\ \Gamma\ (\textit{mk-cert}\ \Phi)$$

The function *mk-cert* :: *prog-type* ⇒ *prog-cert* is the certificate as defined in Section 2.5.4 lifted to programs.

For both $\textit{wt-jvm-prog}_k$ and $\textit{wt-jvm-prog}_l$, I have generated ML code, showing that they are fully executable.

4.5 Type Safety

This section relates the type system discussed above with the operational semantics and its safety automaton for object initialization: Section 4.5.1 states the correctness theorem for the type system with object initialization. Section 4.5.2 shows the conformance relation on which the proof builds.

4.5.1 The Theorem

This section presents the type safety theorem. Because the defensive machine now includes the safety automaton for object initialization, it not only implies that the bytecode verifier guarantees type safe execution, but also that all objects are properly initialized. If the bytecode verifier succeeds and we start the program Γ in its canonical start state (see Section 4.3.3), the defensive μJVM will never return a type error. With Theorem 4.1, this again implies that the checks of the defensive machine are redundant and the aggressive machine can be used safely instead.

The theorem itself is the same as before:

Theorem 4.5 If C is a class in Γ with a *main* method, then

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

The proof remains the same in structure; it again uses an invariant argument with a new conformance relation $\Phi \vdash \sigma\surd$ (to be defined in Section 4.5.2):

Lemma 4.4 Conformance is invariant during execution in welltyped programs.

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

Together with conformance of the start state

Lemma 4.5 If C is a class in Γ with a *main* method, then

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

and the absence of type errors in conformant states

Lemma 4.6 An execution step started in a conformant state cannot produce a type error in welltyped programs.

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

we can conclude Theorem 4.5: there will be no type errors in welltyped programs and all objects are initialized before they are used.

The Isabelle proof of this theorem is about 4,400 lines[2] long (not counting lemmas about the type system) and consists mainly of a large case distinction over the instruction set

[2]The numbers differ from [39] because the type system here additionally includes exceptions and a defensive machine.

together with a wealth of lemmas about the invariant. The same proof without object initialization only took about 2,000 lines. Of these 2,000 lines about 1,500 could be replayed after slight adjustments. The reusable part consisted of lemmas about the invariant (more specifically about its individual parts) and of the general structure of the type safety proof. The detailed reasoning for individual instructions had to be changed due to the stronger properties to be proved. The additional work is mostly due to the new parts of the invariant, adding not only to size but also to complexity. Especially the alias analysis predicate *consistent-init* (see below) causes an increase in proof size: since it is designed to keep track of single values, each kind of instruction required its own set of lemmas (which was not necessary for the rest of the invariant).

Freund presents a similar proof of type safety in his PhD thesis [26]. While Freund's model as a whole is sound and at least the theorems I have looked at more closely all hold, it still contains some subtle problems (like the non-monotonicity of the typing rule for *New*) as well as small errors in the proofs (for instance, an incomplete case distinction in Lemma D.16.16). These are precisely the human errors that are to be expected in a large and complex formal development and that are addressed by theorem provers like Isabelle.

4.5.2 Conformance

This section shows the full definition of the conformance relation $\Phi \vdash \sigma\surd$ used above in Lemma 4.4:

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

To conclude Lemma 4.6 (that there will be no type error in conformant states), the invariant should ensure that runtime types are approximated correctly by the static welltyping. With object initialization, we also want the new *iheap* component of the μJVM state to agree with the initialization status the BV predicts.

As in Section 3.4, we need to strengthen these goals for the proof to succeed. In the previous sections, we extended the μJVM model by three things: the new types *UnInit* and *PartInit*, the *iheap*, and the reference updates at constructor returns (because we create a new object at each constructor call). The conformance relation describes the purpose of these extensions formally. Apart from the relation of the *iheap* to the statically predicted initialization status, this is captured by the following two properties:

- The alias analysis that the BV does on uninitialized values must be correct. We created the type *UnInit C pc* to keep track of a single value: the reference to the object that was freshly created by the instruction *New C* at address *pc*. The type *PartInit C*, too, is intended to keep track of a single value, namely the *this* pointer in constructors. The predicate *consistent-init stk regs s ihp* (see below) states that each type *UnInit C pc* and *PartInit C* in a state type *s* refers to at most one value in stack and registers.

- The BV and the operational semantics must agree on the new objects that are created for constructor calls and on the reference update that we perform at constructor returns. The former is part of *correct-frame*, the latter is described by *constructor-ok* below.

The formal definitions of these additional properties are hard to read and understand, but it is not necessary to understand them in detail in order to trust the proof. The conformance relation is an intermediate proof device, the correctness theorem itself only states the absence of type errors, which is easy to grasp. 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.

Even though the conformance relation is not important for the final result, it is of course essential to carry out the proof. The challenge lies in finding a conformance relation that is invariant during execution, that is strong enough to imply the absence of type errors, and that holds in the start state. Below, I show its formal definition.

The basic building block is still the single value conformance $hp \vdash v ::\preceq T$ of Section 3.2.4 (p. 54). We extend it to take the new *iheap* and *init-ty* into account by declaring single value conformance with initialization $hp,ih \vdash v ::\preceq_i T$:

$$\begin{array}{lll}
\multicolumn{3}{l}{hp,ih \vdash v ::\preceq_i T \equiv} \\
\textit{case } T \textit{ of } \textit{Init } t & \Rightarrow & hp \vdash v ::\preceq t \wedge \textit{is-init } hp\ ih\ v \\
\quad\mid \textit{UnInit } C\ pc & \Rightarrow & hp \vdash v ::\preceq \textit{Class } C \wedge \\
 & & \textit{typeof } hp\ v = \textit{Some } C \wedge \textit{tag } ih\ v = \textit{Some } T \\
\quad\mid \textit{PartInit } C & \Rightarrow & hp \vdash v ::\preceq \textit{Class } C \wedge \textit{tag } ih\ v = \textit{Some } T
\end{array}$$

For initialized types, we just use the existing $::\preceq$, and require with *is-init* that the value is initialized. For *UnInit C pc* and *PartInit C*, we require that v approximates the type *Class C*, and that the *iheap* agrees with the predicted type (*tag ih v* returns *Some T* if the value v is an address tagged with type T in *ih*). In the *UnInit* case, we can be more precise: since *UnInit C pc* is only used for freshly created objects, we know that the static type is exact. The definition of *is-init hp ih v* is a case distinction: either the value is an address, or it is not an address. If it is not an address, it is initialized. If it is an address that does not point to an object, we also treat it as initialized. Note that the definition of $::\preceq$ ensures that this case does not occur. Finally, if the value is an address that points to an object, the *iheap ih* must contain a tag *Init t* for it.

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

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

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

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

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

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

In Section 3.4.2, the invariant *correct-state* implied that the heap is consistent, that all objects on the heap only have fields according to their declared type. Now we also need *h-init hp ihp* to say that the fields of all objects contain fully initialized values. It has the same structure as *hp* $\surd$ in Section 3.4.2: in *h-init*, we say that for each object that is defined in the heap *hp* and each field *f* of these objects, *is-init hp ihp f* must hold:

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

$$\begin{aligned}&\textit{o-init} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{obj} \Rightarrow \textit{bool}\\ &\textit{o-init hp ih } (C,\textit{fs}) \equiv \textit{l-init hp ih fs } (\textit{map-of } (\textit{fields } (\Gamma,C)))\end{aligned}$$

$$\begin{aligned}&\textit{h-init} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{bool}\\ &\textit{h-init hp ih} \equiv \forall\, a\ \textit{obj}.\ \textit{hp } a = \textit{Some obj} \longrightarrow \textit{o-init hp ih obj}\end{aligned}$$

The next component of the conformance relation concerns the alias analysis on uninitialized objects. This part closely follows the description in [26].

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

$$\begin{array}{l}
\textit{corr-regs} :: \textit{registers} \Rightarrow \textit{ty err list} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{init-ty} \Rightarrow \textit{bool}\\
\textit{corr-regs regs lt ihp v T} \equiv\\
\qquad\qquad \textit{list-all2}\ (\lambda l\ t.\ t = \textit{OK T} \longrightarrow l = v \land \textit{tag ihp v} = \textit{Some T})\ \textit{regs lt}
\end{array}$$

$$\begin{array}{l}
\textit{corr-stk} :: \textit{opstack} \Rightarrow \textit{ty list} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{init-ty} \Rightarrow \textit{bool}\\
\textit{corr-stk stk st ihp v T} \equiv \textit{corr-regs stk}\ (\textit{map OK st})\ \textit{ihp v T}
\end{array}$$

$$\begin{array}{l}
\textit{corresponds} :: \textit{opstack} \Rightarrow \textit{registers} \Rightarrow \textit{state-type} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{init-ty} \Rightarrow \textit{bool}\\
\textit{corresponds stk regs s ihp v T} \equiv\\
\qquad\qquad \textit{corr-stk stk}\ (\textit{fst s})\ \textit{ihp v T} \land \textit{corr-regs regs}\ (\textit{snd s})\ \textit{ihp v T}
\end{array}$$

$$\begin{array}{l}
\textit{consistent-init} :: \textit{opstack} \Rightarrow \textit{registers} \Rightarrow \textit{state-type} \Rightarrow \textit{iheap} \Rightarrow \textit{bool}\\
\textit{consistent-init stk regs s ihp} \equiv\\
\qquad\qquad (\forall C\ pc.\ \exists v.\ \textit{corresponds stk regs s ihp v}\ (\textit{UnInit C pc})\ ) \land\\
\qquad\qquad (\forall C.\ \exists v.\ \textit{corresponds stk regs s ihp v}\ (\textit{PartInit C})\ )
\end{array}$$

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

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

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

- The *this* pointer a of the calling constructor (of frame n). This is the reference originally to be initialized. Since at that point the initialization process cannot be complete, it must have the *iheap* tag *UnInit* or *PartInit*.

- The *this* pointer b of the current constructor (of frame $n+1$). This is the reference to the object that was artificially created for the constructor call. It is one step further in the initialization chain and must therefore be tagged with *PartInit* or *Init* (*Init* only if we have reached the end of the chain and arrived at *Object*). We require that if the tag is *PartInit* then it must be *PartInit* C (where C is the current class), and if it is some *Init* (*Class* D), then D from the *iheap* must be equal to the dynamic type C' from the heap.

  The fact that b is exactly one step further in the initialization chain and that it is only fully initialized if the current class is *Object* can be inferred from *app* and the start value of the BV.

- The reference c to the fully initialized object. It will be passed up along the initialization chain by the *Return* instructions in constructors (see also the semantics of the *Return* instruction in Section 4.3). The reference can be *Null* when the superclass constructor has not been called yet. More specifically, it will be *Null* precisely as long as the superclass-constructor-has-been-called marker z of the BV is false. If it is not *Null* then it must point to an initialized object of type C'.

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

$$
\begin{array}{l}
\textit{constructor-ok} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{val} \Rightarrow \textit{cname} \Rightarrow \textit{bool} \Rightarrow \textit{val} \times \textit{val} \Rightarrow \textit{bool} \\
\textit{constructor-ok}\ \textit{hp}\ \textit{ih}\ a\ C\ z\ (b,\ c) \equiv \\
\exists\, C'\ D\ \textit{pc}.\ z = (c \neq \textit{Null}) \wedge \\
\qquad\qquad \textit{typeof}\ \textit{hp}\ a = \textit{Some}\ C' \wedge \textit{typeof}\ \textit{hp}\ b = \textit{Some}\ C' \wedge \\
\qquad\qquad (c \neq \textit{Null} \longrightarrow \textit{typeof}\ \textit{hp}\ c = \textit{Some}\ C') \wedge \\
\qquad\qquad \textit{tag}\ \textit{ih}\ a \in \{\textit{Some}\ (\textit{UnInit}\ C'\ \textit{pc}),\ \textit{Some}\ (\textit{PartInit}\ D)\} \wedge \\
\qquad\qquad \textit{tag}\ \textit{ih}\ b \in \{\textit{Some}\ (\textit{PartInit}\ C),\ \textit{Some}\ (\textit{Init}\ (\textit{Class}\ C'))\} \wedge \\
\qquad\qquad (c \neq \textit{Null} \longrightarrow \textit{tag}\ \textit{ih}\ c = \textit{Some}\ (\textit{Init}\ (\textit{Class}\ C')))
\end{array}
$$

The rest of the invariant proceeds as in Section 3.4.2, we merely have to insert the predicate *constructor-ok* and to adjust for *Invoke-spcl*. The parameters rt_0, sig_0, z_0, and r_0 stem from the call frame above the current f:

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

To the toplevel conformance predicate we add the new *h-init* and change *preallocated* to additionally check that the system exception objects are initialized; the rest of the definition is the same as in Section 3.4.2:

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

4.6 Conclusion

I have presented and proved correct a type system for object initialization in the μJVM. It is important to note that the the type system here also includes safe exception handling. Other formalizations [24, 27] suggest that this is not a trivial matter, as there might be unintended interactions between exception handling and object initialization. In this type system, there was no such problem. Chapter 5 will show that this remains the case even if we include bytecode subroutines which are used to implement the `finally` clause of exception handlers in the source language.

A simpler version of a type system for object initialization in μJava, without exception handling, has appeared in [39].

Although the structure of the formalization and the instantiation process remain the same (compared to Chapter 3), the type safety proof is considerably larger and more complex. Due to the modularity and consequently high reusability of the framework approach, the amount of work to define the type system and derive both executable bytecode verifiers from it was small. The proof of type safety constituted the main workload.

The complete μJava formalization with the type system for object initialization and exception handling consists of about 14,300 lines of Isabelle code (303 pages). Again, as in Chapter 3, this does not include the source language.

The full specification and Isabelle proofs for this type system are available from the VerifiCard project web site [89].

5 Subroutines

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. Hence, subroutines are left out or are restricted in most formalizations of the bytecode verifier. In this chapter, I extend the JVM by subroutines and formalize an expressive, set based type system for the BV that supports subroutines in a simple and elegant way.

5.1 Introduction

The relatively simple concept of procedures in the bytecode language does not seem to fit nicely into the standard data flow analysis approach to bytecode verification. Bytecode subroutines are the center of numerous publications, the cause of bugs in the bytecode verifier, and they even have been banished completely from the bytecode language by Sun in the KVM [88], a JVM for embedded devices.

Publications about subroutines range from describing them as a pain that is best gotten rid of [25] to proposing complex solutions of the problem [26]. Many formalizations of the JVM ignore the *Jsr*/*Ret* instructions altogether [10, 27, 68], offer only restricted versions of it [6, 26, 47, 83], or do not take exception handling and object initialization into account [16, 84].

Subroutines can be seen as procedures on the bytecode level. If the same sequence of instructions occurs more than once in 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 `finally` construct of Java: the `finally` code must be executed on every possible way out of the block protected by `try` (see Figure 5.1 for an example). In contrast to method calls, subroutines share the frame with their caller and hence manipulate the same register set and operand stack.

Two bytecode instructions, namely *Jsr* b and *Ret* x, handle subroutine calls and returns. The *Jsr* b instruction pushes the return address (the current program counter

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

source

```
int m() {
 int i=5;
 try {
  if (i!=0) {
    return i;
  }
  int j=7;
  return j;
 }
 finally {
  if (i!=1) i=3;
 }
}
```

Figure 5.1: Bytecode with subroutine.

incremented by *1*) onto the stack and branches control to address *pc*+*b*. For example, the program in Figure 5.1 contains a subroutine which starts at address *13* and is called from addresses *5* and *10*. Address *13* is called the **entry point**, addresses *5* and *10* the **call points**, and *6* and *11* the **return points** of the subroutine. The *Ret x* instruction returns from a subroutine: 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 usually is a *Store x*.

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

Successors of Ret After the BV has analyzed an instruction, it has to compute its successors in order to propagate the resulting state type and to continue analysis. The successors of *Ret x* instructions are hard to determine, because return addresses are values and not accessible on the type level. For example, in Figure 5.1 at address *19*, the BV must find out that the return address *RA* stored in register *2* refers to the return points *6* and *11*.

Polymorphism on registers Subroutines usually have multiple call points, whose registers may each carry values of different types. One should expect that registers not used inside the subroutine have the same type before and after the subroutine's execution. Figure 5.1 shows that at the entry point, address *13*, the BV merges the types *Int* and *Err* to their least common supertype (*Err*). However, if we do that we lose information about the register's original types. If the BV propagates the merged type back to each return point, some programs will not be accepted, because they expect the original, more specific type. For example, bytecode verification would fail at address *11* in Figure 5.1, because the instruction there expects the original *Int* from address *10* in register *1*. These subroutines are called *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 of references irrespective of its element types.

Subroutine boundaries In Java, bytecode subroutines are not syntactically delimited from their surrounding code. The compilation of the Java source language construct `break`, for instance, is an ordinary jump instruction that can also 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 may be nested: a subroutine may call a further subroutine, and so on. This nesting contributes to the difficulty of determining return points statically. When the BV encounters a *Ret x* instruction, it must find out which of the currently active subroutines is returning. Furthermore, it may be a *multilevel return*: a return not to the subroutine's caller, but to its caller's caller or even further up in the subroutine call stack. The restrictions on nested subroutines differ widely in the literature. The JVM specification [51] only forbids recursive subroutines. Other publications [84] demand a strict LIFO order for nested subroutines. Yet other publications [26] are less restrictive and allow multilevel returns, or they do not constrain the call order at all [16].

Of the approaches in the literature, Freund's [26] seems to be the closest approximation of the JVM specification [51]. Type checking and type inference, however, are rather complicated in this three-step approach that determines the subroutine structure explicitly, searches for unused registers, and finally performs the data flow analysis. The type checking rules in the polyvariant approach of Leroy [47] are simpler and accept more type safe programs, but they, too, require the data flow analysis to be modified.

The polyvariant analysis in [47] is equivalent to first expanding (and thus eliminating) subroutines and then performing standard bytecode verification. Wildmoser's survey of the literature on bytecode subroutines [93] concludes that the type system of Coglio [16] is the most general one (it accepts the largest possible number of programs). It also fits nicely into the data flow framework: Coglio's idea is to use type sets and set union instead of single types and type merges. As Leroy points out in a later article [50], the data flow analysis in this approach in fact implements a model checker on state types. This is not surprising: data flow analysis in general can be seen as model checking of abstract interpretations [78]. The difference between this and using an off-the-shelf model checker, as in [8, 67], is that the explicit data flow analysis is much more lightweight (because there is no translation of the typing rules into temporal logic involved), and it is easier to optimize, as Leroy [50] shows. Whereas the polyvariant approach [47] analyzes each subroutine call history separately, the set approach [16] analyzes all subroutine call histories at the same time.

In this set approach [16], state types are not single types, but rather whole sets of what in the other approaches is the state type. If a program address i is reachable under two different type configurations (st,lt) and (st',lt'), the BV assigns the state type $\{(st,lt),(st',lt')\}$, a set, to i rather than the single, merged $(st \sqcup st',\ lt \sqcup lt')$.

Joining type sets instead of merging types is more precise. Since the original type information is not lost, polymorphism on registers is not a problem anymore. Due to the absence of type merges, we can lift return addresses into the type system: *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 these successors.

Figure 5.2 shows a welltyping for the program of Figure 5.1. For instance, the *Ret 2* instruction at address *19* with state type $\{([], [\mathit{Int}, \mathit{Int}, \mathit{RA}\ 11]), ([], [\mathit{Int}, \mathit{Err}, \mathit{RA}\ 6])\}$ propagates $\{([], [\mathit{Int}, \mathit{Int}, \mathit{RA}\ 11])\}$ to address *11* and $\{([], [\mathit{Int}, \mathit{Err}, \mathit{RA}\ 6])\}$ to address *6*. This models the machine behaviour precisely.

The bytecode language on which Coglio defines the set approach [15, 16] is tailored to subroutines alone, and therefore extremely simple. It does not even contain classes and objects, and types are restricted to integers, floats, and return addresses. In the following, I will show that this approach scales to the representative subset of Java bytecode that I have built up in Chapters 3 and 4.

Section 5.2 introduces the values and type of return addresses into μJava. Section 5.3 shows the operational semantics of the new instructions *Jsr* and *Ret*. Section 5.4 defines a new semilattice and transfer function for the new type system, and instantiates the BV framework once more. Section 5.5 proves that the type system with exception handling, object initialization, and subroutines is sound.

	instruction	state type
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*])}
4	*Ifcmpeq +4*	{([*Int*, *Int*], [*Int*, *Err*, *Err*])}
5	*Jsr +8*	{([], [*Int*, *Err*, *Err*])}
6	*Load 0*	{([], [*Int*, *Err*, *RA 6*])}
7	*Return*	{([*Int*], [*Int*, *Err*, *RA 6*])}
8	*LitPush 7*	{([], [*Int*, *Err*, *Err*])}
9	*Store 1*	{([*Int*], [*Int*, *Err*, *Err*])}
10	*Jsr +3*	{([], [*Int*, *Int*, *Err*])}
11	*Load 1*	{([], [*Int*, *Int*, *RA 11*])}
12	*Return*	{([*Int*], [*Int*, *Int*, *RA 11*])}
13	*Store 2*	{([*RA 11*], [*Int*, *Int*, *Err*]), ([*RA 6*], [*Err*, *Int*, *Err*])}
14	*Load 0*	{([], [*Int*, *Int*, *RA 11*]), ([], [*Int*, *Err*, *RA 6*])}
15	*LitPush 1*	{([*Int*], [*Int*, *Int*, *RA 11*]), ([*Int*], [*Int*, *Err*, *RA 6*])}
16	*Ifcmpeq +3*	{([*Int*, *Int*], [*Int*, *Int*, *RA 11*]), ([*Int*, *Int*], [*Int*, *Err*, *RA 6*])}
17	*LitPush 3*	{([], [*Int*, *Int*, *RA 11*]), ([], [*Int*, *Err*, *RA 6*])}
18	*Store 0*	{([*Int*], [*Int*, *Int*, *RA 11*]), ([*Int*], [*Int*, *Err*, *RA 6*])}
19	*Ret 2*	{([], [*Int*, *Int*, *RA 11*]), ([], [*Int*, *Err*, *RA 6*])}

Figure 5.2: Welltyping for a subroutine.

Figure 5.3 gives an overview of which theories changed. Contrary to object initialization in Chapter 4, most of these changes make the formalization simpler.

5.2 Types and Values

The following data type definition replaces *ty* of Chapters 3 and 4. It introduces the return address *RetA nat* as a primitive type into the μJava type system.

datatype *prim-ty* = *Void* | *Boolean* | *Integer* | *RetA nat*
datatype *ref-ty* = *NullT* | *ClassT cname*
datatype *ty* = *PrimT prim-ty* | *RefT ref-ty*

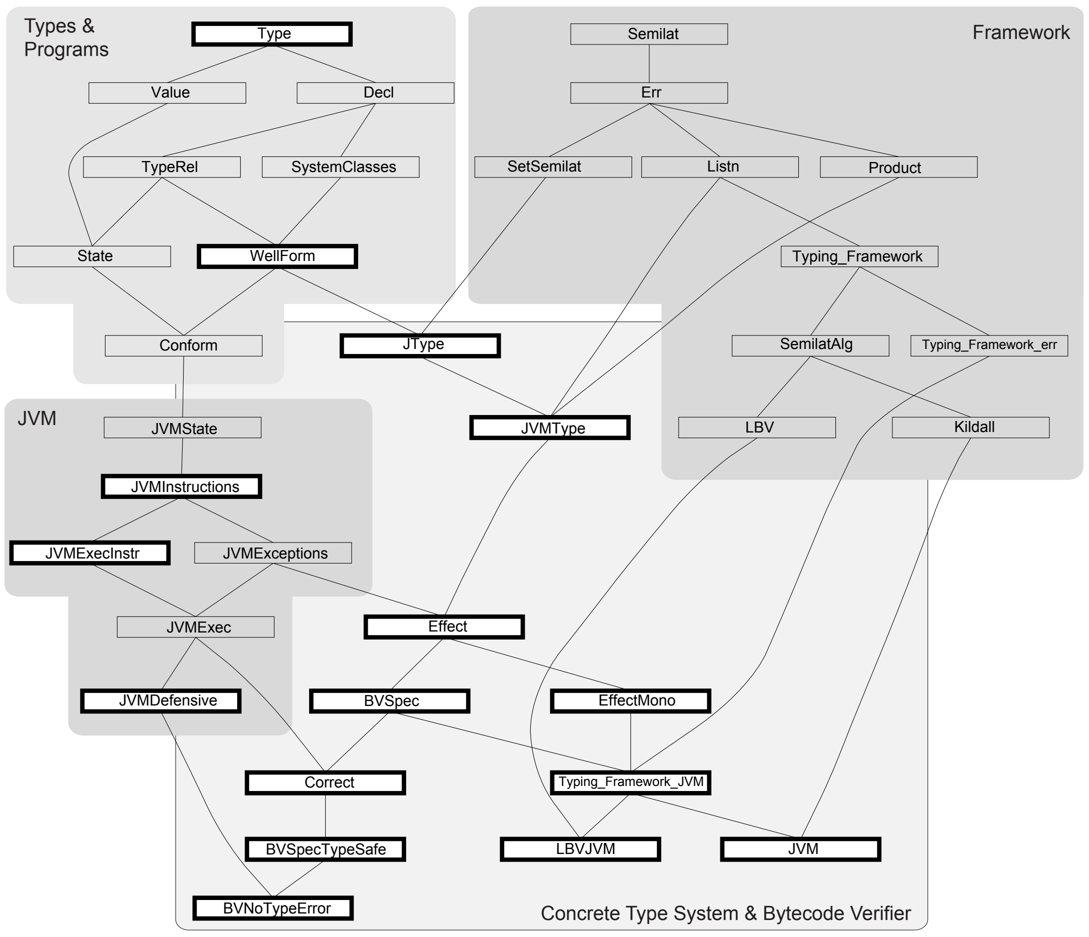

Figure 5.3: Including subroutines: overview.

In addition to the abbreviations *NT* and *Class C*, I introduce *RA pc* to denote the return address *pc*:

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

The function *is-RA T* used below is true iff *T* is a return address.

The types for object initialization on top remain the same:

datatype *init-ty* = *Init ty* | *UnInit cname nat* | *PartInit cname*

Return addresses are only to be used inside methods, not across method boundaries, so we also modify the definition of wellformed programs of Section 3.2.1: field and method declarations should not mention return addresses:

$wf\text{-}fdecl :: \gamma\ prog \Rightarrow fdecl \Rightarrow bool$
$wf\text{-}fdecl\ \Gamma\ (fn,ft) \equiv is\text{-}type\ \Gamma\ ft \wedge \neg is\text{-}RA\ ft$

$wf\text{-}mhead :: \gamma\ prog \Rightarrow sig \Rightarrow ty \Rightarrow bool$
$wf\text{-}mhead\ \Gamma\ (mn,ps)\ rt \equiv is\text{-}type\ \Gamma\ rt \wedge \neg is\text{-}RA\ ft \wedge (\forall T{\in}set\ ps.\ is\text{-}type\ \Gamma\ T \wedge \neg is\text{-}RA\ T)$

The rest of wellformedness remains the same as in Section 3.2.1, pp. 45–46.

Return addresses also have to be handled as values at runtime. The type *val* of values is now defined as:

$$\textbf{datatype}\ val = Unit \mid Null \mid Bool\ bool \mid Intg\ int \mid Addr\ loc \mid RetAddr\ nat$$

The destructor *the-RetAddr*, defined by $the\text{-}RetAddr\ (RetAddr\ r) = r$, is useful below.

5.3 Operational Semantics

This section presents the operational semantics of the *Jsr* and *Ret* instructions. Contrary to the static level in bytecode verification, subroutines constitute a clear and easy concept on the dynamic side.

Introducing the new instructions *Jsr b* and *Ret*, Figure 5.4 shows the new complete instruction set of the μJVM.

The operational semantics of bytecode subroutines is very simple. The definition is the same as the one for object initialization in Section 4.3.2, Figure 4.6. We just have to add two new rules, one for *Jsr* and one for *Ret*:

$exec\text{-}instr\ (Jsr\ b)\ hp\ ihp\ stk\ regs\ Cl\ sig\ pc\ z\ frs =$
$\quad (None, hp, ihp, (RetAddr\ (pc{+}1)\#stk, regs, Cl, sig, nat\ ((int\ pc){+}b), z)\#frs)$
$exec\text{-}instr\ (Ret\ x)\ hp\ ihp\ stk\ regs\ Cl\ sig\ pc\ z\ frs =$
$\quad (None, hp, ihp, (stk, regs, Cl, sig, the\text{-}RetAddr\ (regs\ !\ x), z)\ \#\ frs)$

The *Jsr* instruction puts the return address *pc+1* on the operand stack and performs a relative jump to the subroutine (remember that *nat* and *int* are Isabelle type conversion functions that convert the HOL type *int* to *nat* and vice versa).

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* (*ty list*)	invoke constructor
|	*Return*	return from method
|	*Dup*	duplicate top element
|	*Dup-x1*	duplicate top element and push 2 values down
|	*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 (relative jump)

Figure 5.4: The μJava instruction set with *Jsr* and *Ret*.

The *Ret* x instruction affects only the program counter. It fetches the return address from register x and, with *the-RetAddr*, converts it to *nat*.

The defensive machine does not contain any surprises either. The definition is the same as in Section 4.3.3, Figure 4.8, and the additional rules for *Jsr* and *Ret* are equally short.

In fact, for *Jsr* we only check that the branch did not try to jump outside the method.[1] Note that $pc < mpc$ is handled for all instructions at once in the definition of *check* (see also Section 3.2.4, p. 53).

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

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

$$\begin{aligned}&\textit{check-instr}\ (\textit{Ret}\ x)\ \textit{ihp hp stk regs Cl sig pc z frs} =\\ &\qquad x < \textit{length regs} \wedge \textit{is-RetAddr}\ (\textit{regs}!x)\end{aligned}$$

where *is-RetAddr v* is true iff the value v is a return address.

[1]Not even this check is necessary for type safety, because the aggressive machine converts the result to *nat* anyway. It is useful for the compiler [40, 85, 86], though.

The defensive and aggressive VMs still have the same operational one-step semantics if there are no type errors:

Theorem 5.1 One-step execution in aggressive and defensive machines commutes if there are no type errors.

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

The proof is the same as for Theorem 3.1 in Section 3.2.4, because the definition of *exec-d* is unchanged and because there is no need to unfold the definition of *exec-instr* or *check-instr*.

The canonical start state remains the same. This concludes the operational semantics of bytecode subroutines.

5.4 The Bytecode Verifier

5.4.1 The Semilattice

The abstract framework of Chapter 2 requires state types to form a semilattice that satisfies the ascending chain condition. In this section, I will build up a semilattice suitable for the treatment of the *Jsr* and *Ret* instructions.

Following Coglio [15, 16], state types are sets. Section 2.2.7 already showed that finite sets give rise to a semilattice. The first goal in this section must therefore be to make the set of possible basic types finite, and then to build up a structure which preserves this finiteness and which can describe the μJVM's operand stack and register set.

The basic types *ty* are those defined in Section 5.2 above. Although there is ample opportunity for infinity in these data type definitions, the set of basic types can easily be restricted to a finite subset without excluding any type safe programs: 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, in the context of a fixed program Γ and a method with *mpc* instructions, the carrier set *init-tys* is defined as follows:

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

With $boundedRA\ (mpc, T)$, we check that the program counter is not greater than *mpc* if T is a return address.

We now only need to lift this set to the stack and register structure of the bytecode verifier. As before, the register set is just 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 Section 2.2. The operand stack is a list of maximum length *mxs*. Apart from operand stack and registers, we also need the boolean flag for verifying constructors. Using *list n A* as in Section 2.2.6 for the set of lists over A with length n, we arrive at:

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

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

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

Because *state-types* is finite, it is easy to show the following lemma:

Lemma 5.1 $(\textit{states}, \subseteq, \cup)$ is a semilattice and $\subseteq$ satisfies the ascending chain condition on *states*.

Strictly speaking, the artificial error element is not necessary: the power set semilattice already has a top element (the full set *state-types*). It is, however, impractical to use this full set as an error indicator in the algorithm, and it is also convenient to use the distinction into applicability and effect from Section 2.3.3 which requires *Err* on the top level. Note that there is no *option* layer anymore in the type system. The bottom element of the semilattice, which the BV uses for unreachable code, is $OK\ \{\}$.

Because the state types are finite sets, we can replace them by a list implementation in a real BV. In the ML code generated from the Isabelle specification, I have done so; in the formalization itself, it is more convenient to continue with sets.

5.4.2 Applicability and Effect

In this section, I will instantiate *app* and *eff* for the instruction set of the μJVM with subroutines. Both definitions are again subdivided into a part for normal and a part for exceptional execution.

In order to reuse the transfer function of Chapter 4, I put up with a slight inaccuracy in terminology: I leave the definition of the type *state-type* intact, even though strictly speaking it does not model a proper state type any more. The state types here are sets, the type *state-type* still models one single type configuration for stack and register sets. With the *state-bool* definition of Chapter 4, a proper state type in the sense of the framework is a set of *state-bool* entries. A welltyping is a list of such sets:

$$\begin{array}{ll} \textbf{types} & \textit{state-type} \;\;= \textit{init-ty list} \times \textit{init-ty err list} \\ & \textit{state-bool} \;\;= \textit{state-type} \times \textit{bool} \\ & \textit{method-type} = \textit{state-bool set list} \end{array}$$

With these type definitions, we can leave *app'* and *eff'* almost unchanged; only the lifting step to *app* and *eff* will take the new set level into account.

The method context for the definitions below is still the same as in Chapters 3 and 4:

Γ	:: *program*	the program,
C'	:: *cname*	the class the method we are verifying is declared in,
mn	:: *mname*	the name of the method,
mxs	:: *nat*	maximum stack size of the method,
mxr	:: *nat*	size of the register set,
mpc	:: *nat*	maximum program counter,
rt	:: *ty*	return type of the method,
et	:: *ex-table*	exception handler table of the method.

The exception handling part *xcpt-app* is unaffected by the new instructions *Jsr* and *Ret*. They cannot cause an exception. The definition is the same as in Section 4.4.2 on page 98.

The applicability of instructions in the normal, non-exception case still builds on the old *app'* :: *instr* × *nat* × *state-type* ⇒ *bool*. The two new instructions are easily added (see below). Figure 5.5 shows the full definition.

The *Jsr b* instruction puts the return address on the stack, so we have to make sure that there is enough space for it. The test whether *pc'* is within the code boundaries is again done once for all instructions in *app* below.

$$\textit{app}'\ (\textit{Jsr}\ b,\ pc,\ (st,lt)) = \textit{length}\ st < \textit{mxs}$$

The *Ret x* instruction is equally simple: the index *x* must be inside the register set, and the value in register *x* must be a return address:

$$\textit{app}'\ (\textit{Ret}\ x,\ pc,\ (st,lt)) = x < \textit{length}\ lt \wedge (\exists\, r.\ lt!x{=}\textit{OK}\ (\textit{Init}\ (\textit{RA}\ r)))$$

$$
\begin{array}{ll}
\textit{app}' :: \textit{instr} \times \textit{nat} \times \textit{state-type} \Rightarrow \textit{bool} & \\
\textit{app}'\ (\textit{Load idx},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{idx} < \textit{lt} \wedge \textit{lt}!\textit{idx} \neq \textit{Err} \wedge \textit{size st} < \textit{mxs} \\
\textit{app}'\ (\textit{Store idx},\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = \textit{idx} < \textit{size lt} \\
\textit{app}'\ (\textit{LitPush}\ v,\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{size st} < \textit{mxs} \wedge \\
 & \quad (\textit{typeof}\ v = \textit{Some NT} \vee \\
 & \quad \textit{typeof}\ v = \textit{Some}\ (\textit{PrimT Boolean}) \vee \\
 & \quad \textit{typeof}\ v = \textit{Some}\ (\textit{PrimT Integer})) \\
\textit{app}'\ (\textit{Getfield F C},\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = \textit{is-class}\ \Gamma\ C \wedge\ t \preceq_i \textit{Init}\ (\textit{Class}\ C) \wedge \\
 & \quad (\exists\, t'.\ \textit{field}\ (\Gamma,C)\ F = \textit{Some}\ (C,\ t')) \\
\textit{app}'\ (\textit{Putfield F C},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = \textit{is-class}\ \Gamma\ C \wedge \\
 & \quad (\exists\, t'.\ \textit{field}\ (\Gamma,C)\ F = \textit{Some}\ (C,t') \wedge \\
 & \quad t_2 \preceq_i \textit{Init}\ (\textit{Class}\ C) \wedge t_1 \preceq_i \textit{Init}\ t') \\
\textit{app}'\ (\textit{New}\ C,\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{is-class}\ \Gamma\ C \wedge \textit{size st} < \textit{mxs} \wedge \\
 & \quad \textit{UnInit C pc} \notin \textit{set st} \\
\textit{app}'\ (\textit{Checkcast}\ C,\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = \textit{is-class}\ \Gamma\ C \wedge (\exists\, r.\ t = \textit{Init}\ (\textit{RefT}\ r)) \\
\textit{app}'\ (\textit{Dup},\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = \textit{1+size st} < \textit{mxs} \\
\textit{app}'\ (\textit{Dup-x1},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = \textit{2+size st} < \textit{mxs} \\
\textit{app}'\ (\textit{IAdd},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = t_1 = t_2 \wedge t_1 = \textit{Init}\ (\textit{PrimT Integer}) \\
\textit{app}'\ (\textit{Ifcmpeq}\ b,\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (t_1 = t_2 \vee (\exists\, r\ r'.\ t_1 = \textit{Init}\ (\textit{RefT}\ r) \wedge \\
 & \quad t_2 = \textit{Init}\ (\textit{RefT}\ r'))) \\
\textit{app}'\ (\textit{Goto}\ b,\ \textit{pc},\ s) & = \textit{True} \\
\textit{app}'\ (\textit{Return},\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = t \preceq_i \textit{Init rt} \\
\textit{app}'\ (\textit{Throw},\ \textit{pc},\ (\textit{Init}\ t\#\textit{st},\textit{lt})) & = \textit{is-RefT}\ t \\
\textit{app}'\ (\textit{Jsr}\ b,\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{length st} < \textit{mxs} \\
\textit{app}'\ (\textit{Ret}\ x,\ \textit{pc},\ (\textit{st},\textit{lt})) & = x < \textit{length lt} \wedge (\exists\, r.\ \textit{lt}!x{=}\textit{OK}\ (\textit{Init}\ (\textit{RA}\ r))) \\
\textit{app}'\ (\textit{Invoke C mn ps},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{size ps} < \textit{size st} \wedge \textit{mn} \neq \textit{init} \wedge \\
 & \quad \textit{method}\ (\Gamma,C)\ (\textit{mn},\textit{ps}) \neq \textit{None} \wedge \\
 & \quad \textit{let as} = \textit{rev}\ (\textit{take}\ (\textit{size ps})\ \textit{st}); \\
 & \qquad t\ \ = \textit{st}!\textit{size ps} \\
 & \quad \textit{in}\ t \preceq_i \textit{Init}\ (\textit{Class}\ C) \wedge \textit{is-class}\ \Gamma\ C \wedge \\
 & \qquad (\forall\, (a,f) {\in} \textit{set}(\textit{zip as ps}).\ a \preceq_i \textit{Init}\ f) \\
\textit{app}'\ (\textit{Invoke-spcl C ps},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{size ps} < \textit{size st} \wedge \\
 & \quad (\exists\, r.\ \textit{method}\ (\Gamma,C)\ (\textit{init},\textit{ps}) = \textit{Some}\ (C,r)) \wedge \\
 & \quad \textit{let as} = \textit{rev}\ (\textit{take}\ (\textit{size ps})\ \textit{st}); \\
 & \qquad t\ \ = \textit{st}!\textit{size ps} \\
 & \quad \textit{in is-class}\ \Gamma\ C \wedge \\
 & \qquad ((\exists\, \textit{pc}.\ t = \textit{UnInit C pc}) \vee \\
 & \qquad t = \textit{PartInit}\ C' \wedge (C',C) {\in} \textit{subcls}\ \Gamma) \wedge \\
 & \qquad (\forall\, (a,f) {\in} \textit{set}(\textit{zip as ps}).\ a \preceq_i (\textit{Init}\ f)) \\
\textit{app}'\ (i,\ \textit{pc},\ s) & = \textit{False}
\end{array}
$$

Figure 5.5: Applicability of instructions.

The only change in the rest of *app′* is *LitPush*: we specifically exclude return addresses to be pushed as literals onto the stack.

With *app′*, we can now build the full applicability function *app*: an instruction is applicable when it is applicable in every type configuration in the state type set; 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 within the method:

$$\begin{array}{l}
\mathit{app} :: \mathit{instr} \Rightarrow \mathit{nat} \Rightarrow \mathit{state\text{-}bool\ set} \Rightarrow \mathit{bool}\\
\mathit{app}\ i\ \mathit{pc}\ s \equiv (\forall ((\mathit{st},\mathit{lt}),z) \in s.\ \mathit{xcpt\text{-}app}\ i\ \mathit{pc} \wedge \mathit{app}'\ (i,\mathit{pc},(\mathit{st},\mathit{lt})) \wedge\\
\qquad\qquad\qquad\qquad (\mathit{mn} = \mathit{init} \wedge i = \mathit{Return} \longrightarrow z) \wedge\\
\qquad\qquad\qquad\qquad (\forall\, C\ p.\ i = \mathit{Invoke\text{-}spcl}\ C\ p \wedge \mathit{st}!\mathit{size}\ p = \mathit{PartInit}\ C' \longrightarrow \neg z)) \wedge\\
\qquad\qquad (\forall\, (\mathit{pc}',s') \in \mathit{set}\ (\mathit{eff}\ i\ \mathit{pc}\ s).\ \mathit{pc}' < \mathit{mpc})
\end{array}$$

This concludes applicability.

The exception case *xcpt-eff* of the effect of instructions produces one edge in the flow graph for each exception. The resulting state type of each edge is the input set s where the stack is replaced by a stack that only contains the exception:

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

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 again define them separately. Figure 5.6 shows the successors.

The old instructions in Figure 5.6 are unchanged, and *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:

$$\begin{array}{ll}
\mathit{theRA}\ x\ ((\mathit{st},\mathit{lt}),z) & = \mathit{the\text{-}RA}\ (\mathit{lt}!x)\\
\mathit{the\text{-}RA}\ (\mathit{OK}\ (\mathit{Init}\ (\mathit{RA}\ \mathit{pc}))) & = \mathit{pc}
\end{array}$$

It works on elements $((\mathit{st},\mathit{lt}),z)$ 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

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

Figure 5.6: Successor program counters for the non-exception case.

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 some list that converts to this set.

Remember that in the implementation we will use lists for state types instead of sets, so this *SOME* construction is just the identity function applied to $(\mathit{theRA}\ x)\ `\ s$. In the proofs, *SOME* is not a problem, because we know that s is finite and therefore a suitable list l always exists.

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

As with *app*, we first define the effect eff' on single stack and register sets (Figure 5.7). While eff' postpones the treatment of the *Ret* instruction (by just returning s), the effect of *Jsr b* is defined there: we put $\mathit{pc}{+}1$ as the return address on top of the stack.

Before we turn our attention to *Ret*, we note that the object initialization layer *eff-bool* that concerns the initialization flag z remains unchanged:

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

If it were not for *Ret*, we could apply this *eff-bool* to every element of the state type. For all other instructions we do just that, but 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 originate from different calls to the subroutine (see also the example in Section 5.1, Figure 5.2). The elements of the state type that can return to pc' are those elements that contain the return address

$$
\begin{array}{ll}
\textit{eff}' :: \textit{instr} \times \textit{nat} \times \textit{state-type} \Rightarrow \textit{state-type} & \\
\textit{eff}'\ (\textit{Load idx},\ \textit{pc},\ (\textit{st},\ \textit{lt})) & = (\textit{ok-val}\ (\textit{lt}!\textit{idx})\#\textit{st},\ \textit{lt}) \\
\textit{eff}'\ (\textit{Store idx},\ \textit{pc},\ (t\#\textit{st},\ \textit{lt})) & = (\textit{st},\ \textit{lt}[\textit{idx}{:=}\ \textit{OK}\ t]) \\
\textit{eff}'\ (\textit{LitPush}\ v,\ \textit{pc},\ (\textit{st},\ \textit{lt})) & = (\textit{Init}\ (\textit{the}\ (\textit{typeof}\ v))\#\textit{st},\ \textit{lt}) \\
\textit{eff}'\ (\textit{Getfield}\ F\ C,\ \textit{pc},\ (t\#\textit{st},\ \textit{lt})) & = (\textit{Init}\ (\textit{snd}\ (\textit{the}\ (\textit{field}\ (\Gamma,C)\ F)))\#\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{Putfield}\ F\ C,\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{New}\ C,\ \textit{pc},\ (\textit{st},\textit{lt})) & = (\textit{UnInit}\ C\ \textit{pc}\#\textit{st},\ \textit{replace}\ (\textit{OK}\ (\textit{UnInit}\ C\ \textit{pc}))\ \textit{Err}\ \textit{lt}) \\
\textit{eff}'\ (\textit{Checkcast}\ C,\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = (\textit{Init}\ (\textit{Class}\ C)\ \#\ \textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{Dup},\ \textit{pc},\ (t\#\textit{st},\textit{lt})) & = (t\#t\#\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{Dup-x1},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (t_1\#t_2\#t_1\#\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{IAdd},\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (\textit{Init}\ (\textit{PrimT Integer})\#\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{Ifcmpeq}\ b,\ \textit{pc},\ (t_1\#t_2\#\textit{st},\textit{lt})) & = (\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{Goto}\ b,\ \textit{pc},\ s) & = s \\
\textit{eff}'\ (\textit{Jsr}\ t,\ \textit{pc},\ (\textit{st},\textit{lt})) & = ((\textit{Init}\ (\textit{RA}\ (\textit{pc}{+}1)))\#\textit{st},\textit{lt}) \\
\textit{eff}'\ (\textit{Ret}\ x,\ \textit{pc},\ s) & = s \\
\textit{eff}'\ (\textit{Invoke}\ C\ \textit{mn}\ \textit{ps},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{let}\ \textit{st}' = \textit{drop}\ (1{+}\textit{size ps})\ \textit{st}; \\
& \qquad (\_,\textit{rt},\_,\_,\_) = \textit{the}\ (\textit{method}\ (\Gamma,C)\ (\textit{mn},\textit{ps})) \\
& \quad\ \textit{in}\ (\textit{Init}\ \textit{rt}\#\textit{st}',\ \textit{lt}) \\
\textit{eff}'\ (\textit{Invoke-spcl}\ C\ \textit{ps},\ \textit{pc},\ (\textit{st},\textit{lt})) & = \textit{let}\ t = \textit{st}!\textit{size ps};\ i = \textit{Init}\ (\textit{theClass}\ t); \\
& \qquad \textit{st}'' = \textit{drop}\ (1{+}\textit{size ps})\ \textit{st}; \\
& \qquad \textit{st}' = \textit{replace}\ t\ i\ \textit{st}''; \\
& \qquad \textit{lt}' = \textit{replace}\ (\textit{OK}\ t)\ (\textit{OK}\ i)\ \textit{lt}; \\
& \qquad (\_,\textit{rt},\_,\_,\_) = \textit{the}\ (\textit{method}\ (\Gamma,C)\ (\textit{init},\textit{ps})) \\
& \quad\ \textit{in}\ (\textit{Init}\ \textit{rt}\#\textit{st}',\ \textit{lt}')
\end{array}
$$

Figure 5.7: Effect of instructions on the state type.

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.

$$
\begin{array}{l}
\textit{norm-eff} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool set} \Rightarrow \textit{state-bool set} \\
\textit{norm-eff}\ i\ \textit{pc}\ \textit{pc}'\ s \equiv \\
(\textit{eff-bool}\ i\ \textit{pc})\ \text{`}\ (\textit{if isRet}\ i\ \textit{then}\ \{s'.\ s' \in s \wedge \textit{theRA}\ (\textit{theIdx}\ i)\ s' = \textit{pc}'\}\ \textit{else}\ s)
\end{array}
$$

This is the effect of instructions in the non-exception case. The final effect function is canonical:

$$
\begin{array}{l}
\textit{eff} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool set} \Rightarrow (\textit{nat} \times \textit{state-bool set})\ \textit{list} \\
\textit{eff}\ i\ \textit{pc}\ s \equiv (\textit{map}\ (\lambda \textit{pc}'.\ (\textit{pc}',\ \textit{norm-eff}\ i\ \textit{pc}\ \textit{pc}'\ s))\ (\textit{succs}\ i\ \textit{pc}\ s))\ @\ (\textit{xcpt-eff}\ i\ \textit{pc}\ s)
\end{array}
$$

This concludes the definition of the transfer function. Note that the monotonicity problem of *New* (see also Section 4.4.2) disappears in this type system, because the ordering ($\subseteq$) is now different from the type ordering. On the other hand, the *replace* in the *New* rule is not the identity anymore because there is no merging.

5.4.3 Executable Bytecode Verifiers

Having defined the semilattice and the transfer function in Section 5.4.1 and Section 5.4.2, I show in this section how the parts are put together to form the two executable bytecode verifiers the framework provides.

As in Chapter 4, the definition of welltypings in *wt-method*, which serves as the basis for the type safety proof and the executable bytecode verifiers, remains unchanged—only *wt-start* has to be adjusted. The changes to *wt-method* occur in the underlying semilattice and transfer function.

$$\begin{aligned}
&\textit{wt-start}\ \varphi \equiv \textit{let}\ t = \textit{OK}\ (\textit{if}\ mn = \textit{init} \wedge C' \neq \textit{Object then PartInit}\ C'\ \textit{else Init}\ (\textit{Class}\ C'));\\
&\qquad\qquad\qquad s_0 = (([],t\#(\textit{map}\ (\textit{OK} \circ \textit{Init})\ ps)@(\textit{replicate mxl Err})),C'{=}\textit{Object});\\
&\qquad\qquad\quad \textit{in}\ \{s_0\} \subseteq \varphi!0
\end{aligned}$$

$$\textit{wt-method}\ \varphi \equiv 0 < \textit{mpc} \wedge \textit{map OK}\ \varphi \in \textit{states} \wedge \textit{wt-start}\ \varphi \wedge \textit{wt-app-eff}\ \varphi$$

The start value is almost the same as in Section 4.4.3; we merely leave out the *Some* of the *option* layer, and write $\{s_0\} \subseteq \varphi!0$ instead of $s_0 \leq' \varphi!0$. The expression $\{s_0\} \subseteq \varphi!0$ is, of course, equivalent to $s_0 \in \varphi!0$, but this way the semilattice order is more visible.

Lifting to programs and filling in the method context remains unchanged:

$$\begin{aligned}
&\textit{wt-jvm-prog}\ \Gamma\ \Phi \equiv \textit{wf-prog}\ (\lambda\Gamma\ C'\ ((mn,ps),rt,(mxs,mxl,ins,et)).\\
&\qquad\qquad\qquad\qquad \textit{wt-method}\ \Gamma\ C'\ mn\ ps\ rt\ mxs\ mxl\ ins\ et\ (\Phi\ C'\ (mn,ps)))\ \Gamma
\end{aligned}$$

The instantiation of Kildall's algorithm is again standard, we just use the new start value and bottom element:

$$\begin{aligned}
&\textit{wt-kil} \equiv 0 < \textit{size ins} \wedge\\
&\textit{let}\ t = \textit{OK}\ (\textit{if}\ mn = \textit{init} \wedge C' \neq \textit{Object then PartInit}\ C'\ \textit{else Init}\ (\textit{Class}\ C'));\\
&\quad s_0 = (([],t\#(\textit{map}\ (\textit{OK} \circ \textit{Init})\ ps)@(\textit{replicate mxl Err})),C'{=}\textit{Object});\\
&\quad \varphi_0 = (\textit{OK}\ \{S_0\})\#(\textit{replicate}\ (\textit{size ins}-1)\ (\textit{OK}\ \{\}))\\
&\textit{in}\ \forall\, n < \textit{size ins}.\ (\textit{kiljvm}\ \varphi_0)!n \neq \textit{Err}
\end{aligned}$$

$$\begin{aligned}
&\textit{wt-jvm-prog}_k\ \Gamma \equiv\\
&\textit{wf-prog}\ (\lambda\Gamma\ C'\ ((mn,ps),rt,(mxs,mxl,ins,et)).\ \textit{wt-kil}\ \Gamma\ C'\ mn\ ps\ rt\ mxs\ mxl\ ins\ et)\ \Gamma
\end{aligned}$$

The following lemma even becomes easier to prove:

Lemma 5.2 The transfer function *step*, built from *app* and *eff* as described in Section 2.3.3 and Section 5.4.2, is monotone, bounded, and type preserving (w.r.t. *states* and *size ins*).

The proof that *step* is bounded remains the same. Monotonicity becomes easier: 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 remains a large case distinction over the instruction set, but again Isabelle handles most cases automatically.

With Theorem 2.1 follows soundness and completeness.

Theorem 5.2 The executable BV is sound and recognizes all welltyped programs:

$$\textit{wt-jvm-prog}_k\ \Gamma = (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

The lightweight bytecode verifier is straightforward, too:

$$\begin{array}{l}
\textit{wt-lbv} :: \textit{state-type option err cert} \Rightarrow \textit{bool}\\
\textit{wt-lbv}\ c \equiv \textit{check-cert}\ (\textit{size ins})\ c \wedge 0 < \textit{size ins} \wedge\\
\qquad\qquad \textit{let}\ t = \textit{OK}\ (\textit{if mn} = \textit{init} \wedge C' \neq \textit{Object then PartInit}\ C'\ \textit{else Init}\ (\textit{Class}\ C'));\\
\qquad\qquad\quad s_0 = \textit{OK}\ \{(([],t\#(\textit{map}\ (\textit{OK} \circ \textit{Init})\ \textit{ps})@(\textit{replicate mxl Err})),C'{=}\textit{Object})\};\\
\qquad\qquad \textit{in wtl}\ c\ s_0 \neq \textit{Err})
\end{array}$$

$$\begin{array}{l}
\textit{wt-jvm-prog}_l :: \textit{jvm-prog} \Rightarrow \textit{prog-cert} \Rightarrow \textit{bool}\\
\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \equiv \textit{wf-prog}\ (\lambda\Gamma\ C'\ ((\textit{mn},\textit{ps}),\textit{rt},(\textit{mxs},\textit{mxl},\textit{ins},\textit{et})).\\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{wt-lbv}\ \Gamma\ C'\ \textit{mn ps rt mxs mxl ins et}\ (\textit{Cert}\ C'\ (\textit{mn},\textit{ps})))\ \Gamma
\end{array}$$

As before, the *check-cert* predicate ensures that the certificate is wellformed.

Theorems 2.3 and 2.4, together with Lemma 5.2 and the semilattice construction in Section 5.4.1, give us that the LBV is sound and complete for this type system.

Theorem 5.3 If the LBV accepts a program, it is welltyped:

$$\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \longrightarrow (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

Theorem 5.4 The LBV accepts every welltyped program:

$$\textit{wt-jvm-prog}\ \Gamma\ \Phi \longrightarrow \textit{wt-jvm-prog}_l\ \Gamma\ (\textit{mk-cert}\ \Phi)$$

The function *mk-cert* :: *prog-type* ⇒ *prog-cert* is the certificate as defined in Section 2.5.4 lifted to programs.

For both $\textit{wt-jvm-prog}_k$ and $\textit{wt-jvm-prog}_l$, I have generated ML code, showing that they are fully executable. The type sets are implemented by lists.

5.5 Type Safety

This section presents the type safety theorem. It implies that the bytecode verifier guarantees type safe execution, and that all objects are properly initialized, both in the presence of bytecode subroutines and exceptions. With Theorem 5.1, this again implies that the checks of the defensive machine are redundant and the aggressive machine can be used safely instead.

The theorem itself is the same as before:

Theorem 5.5 If C is a class in Γ with a *main* method, then

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

The proof structure with the main lemmas and the invariant argument also remains the same:

Lemma 5.3 Conformance is invariant during execution in welltyped programs.

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

Lemma 5.4 If C is a class in Γ with a *main* method, then

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

Lemma 5.5 An execution step started in a conformant state cannot produce a type error in welltyped programs.

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

The trick is again to find the right invariant $\Phi\vdash \sigma\surd$. Contrary to Chapter 4, this is refreshingly easy and elegant for the set based type system.

For $\Phi \vdash \sigma\surd$ to be true, the following must hold: if, in state σ, execution is at position *pc* of method $(C,\textit{sig})$, then there must be an element s of the state type $(\Phi\ C\ \textit{sig})!\textit{pc}$

such that σ conforms to s in the sense of the old invariant of Chapter 4 (see below for the formal definition).

The proof of invariance, Lemma 5.3, then works as follows: for each instruction, we pick an element s of $(\Phi\ C\ sig)!pc$, and we conclude from the conformance of σ together with the *app* part of *wt-jvm-prog* that all assumptions of the aggressive machine are met. We then execute the instruction and observe that the new state τ conforms to $t = \mathit{eff}\ pc\ s$. This t is the element of $(\Phi\ C\ sig)!pc'$ that shows $\Phi \vdash \tau\surd$. The reasoning at that level is the same as for the proof of type safety for the type system in Chapter 4. It even becomes a bit simpler, because before it involved a widening step on t which disappears here (earlier, the BV provided $t \leq' (\Phi\ C\ sig)!pc$, whereas now we directly know $t \in (\Phi\ C\ sig)!pc$).

The formal definition of conformance is the same as in Section 4.5.2. We merely have to replace every $\Phi\ C\ sig!pc = Some\ x$ by $x \in \Phi\ C\ sig!pc$. This affects line 6 of *correct-frames* and line 5 of $\text{-} \vdash \text{-} \surd$ only. The new definitions of *correct-frames* and conformance read:

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

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

Everything else remains as in Section 4.5.2.

5.6 Conclusion

I have presented a formalization of the μJVM with bytecode subroutines. In this type system, subroutines are not artificially restricted for the sake of bytecode verification as they are in other approaches in the literature.

The instantiation of the abstract framework of Chapter 2 resulted in two verified bytecode verifiers for a type system that supports classes, subroutines, object initialization, and exception handling. The bytecode verifier is fully executable and proved correct.

The treatment of subroutines caused the formalization to be reduced from 14,300 lines for the type system for object initialization to 13,600 lines (293 pages) for the type system with subroutines (but still containing object initialization). The reduction is mainly due to the simpler proofs of monotonicity and type safety. The full formalization is available on the web [89].

The type system I use is based on Coglio's idea [16] of using sets to avoid type merges altogether. The formalization presented here is more than a version of [16] in Isabelle/HOL, though: I have shown that the idea scales up to a realistic model of the JVM ([16] did not even have classes), and that subroutines do not necessarily interfere with exception handling or object initialization as it is the case for the type systems by Freund [27] and Stärk et al. [81]. Both propose to restrict subroutines in the BV to work around that.[2]

In theory, the sets that are used as state types in the data flow analysis might become very large (up to the full set of all possible types). The sets could grow at every join operation, that is, 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 [49] 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. My own experience is the same: even for contrived examples (taken from [81]), 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.

Section 5.4.3 presented the first formalization and the first proof of soundness of lightweight bytecode verification that supports subroutines, thus eliminating the need to expand subroutines prior to verification on embedded devices.

[2]The fix that Freund proposes in [27] for the subroutine-related bug in the BV of JDK 1.1.4 (to disallow the type *UnInit C pc* in subroutines), is not monotone, though. It is therefore not usable for the data flow analysis. The fixed rule that Sun actually used is monotone.

6 Arrays

Arrays make the definition of types recursive and complicate the subtype relation. In this chapter, I show how arrays can be integrated cleanly into the JVM and the set based type system of the BV. Although array support requires four new bytecode instructions and a new heap model, the changes in the bytecode verifier are small and isolated.

6.1 Introduction

This chapter introduces the concept of arrays into the μJava language.

Arrays are potentially problematic for the bytecode verifier, because the definition of types becomes recursive: arrays can have any type as element type, and specifically also another array. This in turn makes the subtyping relation more complex: following the JVM specification, each array is a subtype of *Object*, and if T is a subtype of T', then an array of T is a subtype of an array of T'.

At runtime, arrays are treated like objects: their content is stored in the heap, while the operand stack and registers only contain references. Arrays are created dynamically, and the JVM stores their length along with the content to make bounds checking possible. Note that this complicates the heap model: apart from objects, the heap can now also hold arrays.

In the bytecode language, there are separate instructions for storing and retrieving array elements, for creating new arrays, and for retrieving the length of an array. A number of system exceptions handle array-specific error situations: while *ArrayIndexOutOfBounds* and *NegativeArraySize* speak for themselves, the *ArrayStore* exception demands explanation. Due to the subtyping rules for arrays, the following piece of Java source language code cannot be statically type checked:

```
void m(T[] a, T b) { a[0] = b; }
```

Consider two different, unrelated subtypes A and B of T: since an A-array is assignment compatible to a T-array, the value of the a parameter could be of runtime type $A[]$ while the value of b could be of runtime type B. With these types, the assignment `a[0] = b` is not type safe: an A-array can only have values which are of a subtype of A. A and B may have a common supertype T, but B is not necessarily a subtype of A. Hence, array store operations have to be checked at runtime.

Arrays are orthogonal to the other features of the BV. They could be added to a merging type system like the one in Chapter 4 as well as to the set based type system of Chapter 5.

For the merging type systems, the situation is easy: the new subtyping relation still is a partial order, and the new μJava types still form a semilattice. The subtyping order also still satisfies the ascending chain condition, even for unbounded array dimensions. As Knoblock and Rehof [42, 43] point out, the order gives rise to an infinitely *descending* chain:

$$\mathit{Object} \succeq \;\mathit{Object}\;[] \succeq \;\mathit{Object}\;[]\;[] \succeq \mathit{Object}\;[]\;[]\;[] \succeq \ldots$$

In the other direction, however, every chain must be finite: for every n-dimensional array, the end of the chain (Object) is reached after n steps.

Since there are no problems to be expected with a merging type system, and since the set based type system is more expressive and the resulting language with arrays more comprehensive, the choice is obvious: in this chapter I will extend the set based type system of Chapter 5 with arrays.

The set based type system requires that the set of basic types is finite. As indicated above, this is problematic for arrays. There are at least two possible solutions:

- For each program, there must be a maximum dimension of the arrays that are used in the program. In the definition of the set of types, we can take this number as bound for the array dimension and thus are left with a finite set.

- The JVM specification restricts the dimension of arrays to 255 for technical reasons (the JVM uses one byte to express the dimension). This automatically makes the set of all basic types finite.

As it is easier to read and more succinct in the formalization, I will use the second solution in the following.

Section 6.2 introduces array types formally. Section 6.3 shows the operational semantics of the new instructions. Section 6.4 defines semilattice and transfer function for the extended type system, and instantiates the BV framework once more. Section 6.5 shows

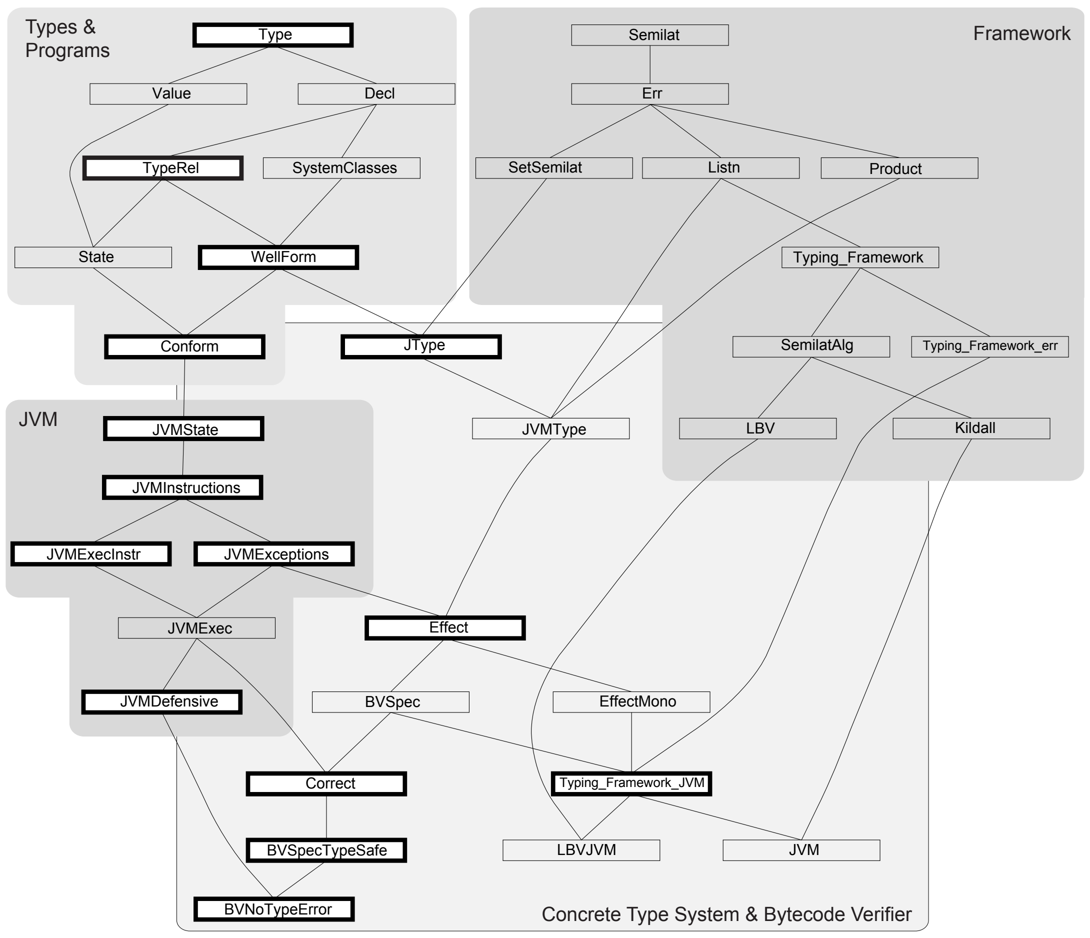

Figure 6.1: Including arrays: overview.

that the type system with exception handling, object initialization, subroutines, and arrays is sound.

Figure 6.1 gives an overview of which theories have changed. While there are significant changes in the virtual machine, the changes in the bytecode verifier are mainly simple additions for the new instructions: they take place in the definition of the semilattice and transfer function, and in the type safety proof.

6.2 Types and Wellformed Programs

This section defines the type of arrays formally and shows how the wellformedness conditions for programs are adjusted accordingly.

Introducing arrays makes the definition of μJava types *ty* mutually recursive with *ref-ty*, because the new reference type *ArrayT* can take any type *ty* as parameter:

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

The types for object initialization on top remain the same:

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

To make the notation more reminiscent of Java, I introduce the following abbreviation:

$$\textbf{translations}\ T.[] \rightleftharpoons \textit{RefT}\ (\textit{ArrayT}\ T)$$

The subtyping relation is extended by the two rules mentioned in Section 6.1: each array is a subtype of *Object*, and two arrays are in the subtype relation if their component types are.

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

With arrays, we also need three more of Java's system exceptions; one for index errors, one for negative sizes in array creation, and one for the subtyping problem outlined in Section 6.1:

$$\begin{array}{ll}
\textbf{datatype}\ \textit{xcpt} = & \textit{NullPointer} \mid \textit{ClassCast} \mid \textit{OutOfMemory} \\
& \mid \textit{ArrayIndexOutOfBounds} \mid \textit{NegativeArraySize} \mid \textit{ArrayStore}
\end{array}$$

For wellformedness of programs below, we will need the dimension *dim* :: *ty* $\Rightarrow$ *nat* of a type. Since the data type definition is mutually recursive, this is defined using a helper function *dim-r* :: *ref-ty* $\Rightarrow$ *nat*:

$$\begin{array}{llll} \textit{dim}\ (\textit{PrimT}\ T) = \textit{0} & & \textit{dim-r}\ \textit{NullT} & = \textit{0} \\ \textit{dim}\ (\textit{RefT}\ T) & = \textit{dim-r}\ T & \textit{dim-r}\ (\textit{ClassT}\ C) & = \textit{0} \\ & & \textit{dim-r}\ (\textit{ArrayT}\ T) & = \textit{1} + \textit{dim}\ T \end{array}$$

In accordance with the JVM specification [51, §4.8.1], there is a maximum dimension of arrays *max-dim*. The actual value of *max-dim* is irrelevant for the formalization. For example programs, I have used *255* as the JVM specification requires.

In wellformed programs, no object field or method parameter can hold a return address. Now we have to make sure that they also do not hold an array of return addresses. To that end, I define *noRA* :: *ty* ⇒ *bool* and the helper *noRA-r* :: *ref-ty* ⇒ *bool* as an extension of the condition ¬*is-RA T* of Section 5.2.

$$\begin{array}{llll} \textit{noRA}\ (\textit{PrimT}\ T) = \neg\textit{is-RA}\ (\textit{PrimT}\ T) & & \textit{noRA-r}\ \textit{NullT} & = \textit{True} \\ \textit{noRA}\ (\textit{RefT}\ T) & = \textit{noRA-r}\ T & \textit{noRA-r}\ (\textit{ClassT}\ C) & = \textit{True} \\ & & \textit{noRA-r}\ (\textit{ArrayT}\ T) & = \textit{noRA}\ T \end{array}$$

With this, we can express that in declarations of wellformed programs no return addresses and only arrays with a dimension up to *max-dim* occur. As in Section 5.2, only the definitions for wellformed fields and method headers need to be adjusted:

$$\textit{wf-fdecl}\ \Gamma\ (\textit{fn},\textit{ft}) \equiv \textit{is-type}\ \Gamma\ \textit{ft} \wedge \textit{noRA}\ \textit{ft} \wedge \textit{dim}\ \textit{ft} \leq \textit{max-dim}$$

$$\begin{array}{ll} \textit{wf-mhead}\ \Gamma\ (\textit{mn},\textit{ps})\ \textit{rt} \equiv & \textit{is-type}\ \Gamma\ \textit{rt} \wedge \textit{noRA}\ \textit{rt} \wedge \textit{dim}\ \textit{rt} \leq \textit{max-dim} \wedge \\ & (\forall\ T{\in}\textit{set ps}.\ \textit{is-type}\ \Gamma\ T \wedge \textit{noRA}\ T \wedge \textit{dim}\ T \leq \textit{max-dim}) \end{array}$$

Array references are assignment compatible to references of class *Object*, so the set of fields of class *Object* must be a subset of the set of fields of arrays (which is empty). The Java Language Specification only mentions implicitly that class *Object* cannot contain fields: in the declaration of *Object* in [32, §4.3.2] there do not occur any. The methods of class *Object* are not a problem: since they cannot manipulate fields, they can be invoked on arrays without harm. Note that the *length* component of arrays is neither a field nor a method. It can be viewed as a special purpose, read only field, but it is not a declared object field in the sense of Section 3.2.1.

If we add the condition *fields* Γ *Object* = [] to *wf-syscls*, the new wellformedness definition of programs is complete. The rest stays as shown in Chapter 3.

$$\textit{wf-syscls} :: \gamma\ \textit{prog} \Rightarrow \textit{bool}$$
$$\textit{wf-syscls}\ \Gamma \equiv \textit{let}\ \textit{cs} = \textit{set}\ \Gamma\ \textit{in}\ \textit{Object} \in \textit{fst`cs} \wedge (\forall x.\ \textit{Xcpt}\ x \in \textit{fst`cs}) \wedge \textit{fields}\ \Gamma\ \textit{Object} = []$$

6.3 Operational Semantics

This section introduces arrays to the μJVM. Section 6.3.1 shows the heap model, Section 6.3.2 the aggressive machine, and Section 6.3.3 the defensive machine with the new array instructions.

6.3.1 State Space

With arrays, entries in the μJVM heap are more complex than before. An entry can be either an object with class name and fields, or an array with component type, length, and entries. Analogously to fields, the array entries are modeled as a partial function from index to values *val*.

$$\begin{aligned} \textbf{datatype}\ \textit{heap-entry} &= \textit{Obj cname}\ (\textit{vname} \times \textit{cname} \Rightarrow \textit{val option}) \\ &\mid\ \textit{Arr ty int}\ (\textit{int} \Rightarrow \textit{val option}) \end{aligned}$$

Heap access in the μJVM is now more than a simple *the* for the option type. For convenience, I introduce the following two destructor functions:

$$\begin{aligned} &\textit{the-obj} :: \textit{heap-entry option} \Rightarrow (\textit{cname}, \textit{vname} \times \textit{cname} \Rightarrow \textit{val option}) \\ &\textit{the-obj}\ (\textit{Some}\ (\textit{Obj C fs})) = (C{,}\textit{fs}) \end{aligned}$$

$$\begin{aligned} &\textit{the-arr} :: \textit{heap-entry option} \Rightarrow (\textit{ty} \times \textit{int} \times \textit{int} \Rightarrow \textit{val option}) \\ &\textit{the-arr}\ (\textit{Some}\ (\textit{Arr T l en})) = (T{,}l{,}\textit{en}) \end{aligned}$$

The conformance relation $::\preceq$ between values and types also has to take the new heap structure into account. The definition of $::\preceq$ is still the same, but *typeof*, on which it builds, distinguishes between objects and arrays on the heap:

$$\begin{aligned} &\text{-} \vdash \text{-} ::\preceq \text{-} :: \textit{aheap} \Rightarrow \textit{val} \Rightarrow \textit{ty} \Rightarrow \textit{bool} \\ &\textit{hp} \vdash v ::\preceq T \equiv \exists T'.\ \textit{typeof hp}\ v = \textit{Some}\ T' \wedge T' \preceq T \end{aligned}$$

$$\begin{aligned} &\textit{typeof} :: \textit{aheap} \Rightarrow \textit{val} \Rightarrow \textit{ty option} \\ &\textit{typeof hp Unit} && = \textit{Some}\ (\textit{PrimT Void}) \\ &\textit{typeof hp}\ (\textit{Intg}\ i) && = \textit{Some}\ (\textit{PrimT Integer}) \\ &\textit{typeof hp}\ (\textit{Bool}\ b) && = \textit{Some}\ (\textit{PrimT Boolean}) \\ &\textit{typeof hp Null} && = \textit{Some NT} \\ &\textit{typeof hp}\ (\textit{Addr}\ a) && = \textit{case hp a of None} \Rightarrow \textit{None} \mid \textit{Some entry} \Rightarrow \textit{Some}\ (\textit{type-of entry}) \end{aligned}$$

$$\begin{aligned} &\textit{type-of} :: \textit{heap-entry} \Rightarrow \textit{ty} \\ &\textit{type-of}\ (\textit{Obj C fs}) && = \textit{Class C} \\ &\textit{type-of}\ (\textit{Arr T l en}) && = T.[] \end{aligned}$$

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* (*ty list*)	invoke constructor
|	*Return*	return from method
|	*Dup*	duplicate top element
|	*Dup-x1*	duplicate top element and push 2 values down
|	*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 (relative jump)
|	*ArrLoad*	load indexed entry from array
|	*ArrStore*	write value to indexed array entry
|	*ArrLength*	retrieve length of array
|	*ArrNew ty*	create new 1-dimensional array

Figure 6.2: The μJava instruction set with arrays.

6.3.2 Aggressive Machine

The μJVM contains four instructions for array handling: *ArrLoad*, *ArrStore*, *ArrLength*, and *ArrNew T*. Figure 6.2 shows the complete instruction set.

The *ArrLoad* and *ArrStore* instructions are polymorphic like *Load* and *Store*: they work on arrays of arbitrary type. As for the register operations, the real JVM contains typed instructions for each primitive type and one for addresses. The *ArrLength* and *ArrNew* instructions behave as described in the JVM specification. The real JVM also contains an instruction for creating multidimensional arrays in one step. This is important for efficiency, but as it can be simulated by a loop, I have left this instruction out of the formalization. For the BV, the instruction is analogous to *ArrNew*: it just puts an array like $A.[].[]$ with more than one dimension on the stack.

The definitions of *exec* and $\xrightarrow{\text{jvm}}$ are still the same as in Section 4.3.2. The operational semantics of the non-array instructions is unchanged, only the object handling instructions now use *the-obj* instead of *the* to access the heap. Those are *Getfield*, *Putfield*, *Invoke*, and *Invoke-spcl*; the *New* instruction is not affected. Note that in the new definitions below, merely line 4 for *Getfield* and line 5 for *Putfield* changed w.r.t. Section 4.3.2.

```
exec-instr (Getfield F C) hp ihp stk regs C' sig pc z frs =
    let r = hd stk;
        xp' = raise-system-xcpt (r=Null) NullPointer;
        (D,fs) = the-obj (hp(the-Addr r));
        pc' = if xp'=None then pc+1 else pc
    in (xp', hp, ihp, (the(fs(F,C))#(tl stk), regs, C', sig, pc', z)#frs)

exec-instr (Putfield F C) hp ihp stk regs C' sig pc z frs =
    let (v,r)= (hd stk, hd(tl stk));
        xp' = raise-system-xcpt (r=Null) NullPointer;
        a = the-Addr r;
        (D,fs) = the-obj (hp a);
        hp' = if xp'=None then hp(a↦(D,fs((F,C)↦v))) else hp;
        pc' = if xp'=None then pc+1 else pc
    in (xp', hp', (tl (tl stk), regs, C', sig, pc', z)#frs)
```

The *Invoke* and *Invoke-spcl* instructions also access the heap, and therefore need to be adjusted. Contrary to *Getfield* and *Putfield*, the method invocation instructions do not care whether the entry is an object or an array. Using

the-obj-ty :: *heap-entry option* ⇒ *cname*
the-obj-ty (*Some* (*Obj C fs*)) = *C*
the-obj-ty (*Some* (*Arr T l en*)) = *Object*

they view arrays as objects. This coincides with the fact that arrays provide exactly the methods of class *Object*. The actual change is again small: only line *4* in the definition of *Invoke* is different from the one in Section 4.3.2:

```
exec-instr (Invoke C mn ps) hp ihp stk regs C' sig pc z frs =
    let n = size ps; args = take n stk; r = stk!n;
        xp' = raise-system-xcpt (r=Null) NullPointer;
        dt = the-obj-ty (hp(the-Addr r));
        (dc,-,-,mxl,-)= the (method (Γ,dt) (mn,ps));
        frs' = if xp'≠None then [] else
                  [([],(rev args)@[r]@replicate mxl arbitrary,dc,(mn,ps),0,arbitrary)]
    in (xp', hp, ihp, frs'@(stk, regs, C', sig, pc, z)#frs)
```

The definition for *Invoke-spcl* is transformed analogously to the one for *Invoke*: we just have to write *the-obj-ty* (*hp* -) instead of *fst* (*the* (*hp* -)).

Now we are set to define one-step execution of the new array instructions. *ArrLoad* is straightforward:

```
$exec\text{-}instr\ (ArrLoad)\ hp\ ihp\ stk\ regs\ C'\ sig\ pc\ z\ frs =$
    $let\ (i,r)\qquad = (the\text{-}Intg\ (stk!0),\ stk!1);$
        $xp''\qquad = raise\text{-}system\text{-}xcpt\ (r{=}Null)\ NullPointer;$
        $(T,l,en) = the\text{-}arr\ (hp(the\text{-}Addr\ r));$
        $xp'\qquad = if\ xp''{=}None$
                  $then\ raise\text{-}system\text{-}xcpt\ (i < 0 \lor l \le i)\ ArrayIndexOutOfBounds\ else\ xp'';$
        $pc'\qquad = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
    $in\ (xp',\ hp,\ ihp,\ (the\ (en\ i)\ \#\ (tl\ (tl\ stk)),\ regs,\ C',\ sig,\ pc',\ z)\#frs)$
```

The definition above fetches the index *i* and the array reference *r* from the operand stack, and retrieves the array entry (*T*,*l*,*en*) from the heap. If *r* is *Null*, it throws a *NullPointer* exception, and if the index is not within the arrays bounds, it throws an *ArrayIndexOutOfBounds* exception. Finally, it increments the program counter and puts the array entry at index *i* on top of the stack.

The definition for *ArrStore* is larger:

```
$exec\text{-}instr\ (ArrStore)\ hp\ ihp\ stk\ regs\ C'\ sig\ pc\ z\ frs =$
    $let\ (v,i,r)\quad = (stk!0,\ the\text{-}Intg\ (stk!1),\ stk!2);$
        $xp_0\qquad = raise\text{-}system\text{-}xcpt\ (r{=}Null)\ NullPointer;$
        $a\qquad\quad = the\text{-}Addr\ r;$
        $(T,l,en) = the\text{-}arr\ (hp\ a);$
        $xp_1\qquad = if\ xp_0{=}None$
                  $then\ raise\text{-}system\text{-}xcpt\ (i < 0 \lor l \le i)\ ArrayIndexOutOfBounds\ else\ xp_0;$
        $xp'\qquad = if\ xp_1{=}None\ then\ raise\text{-}system\text{-}xcpt\ (\neg hp \vdash v ::\preceq T)\ ArrayStore\ else\ xp_1;$
        $hp'\qquad = if\ xp'{=}None\ then\ hp(a \mapsto Arr\ T\ l\ (en(i \mapsto v)))\ else\ hp;$
        $pc'\qquad = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
    $in\ (xp',\ hp',\ ihp,\ (tl\ (tl\ (tl\ stk)),\ regs,\ C',\ sig,\ pc',\ z)\#frs)$
```

The *ArrStore* instruction expects three values on the operand stack: the value *v* to be stored, the array index *i*, and the array reference *r*. After checking for *Null* pointer and array bounds, *ArrStore* also tests if the value *v* is compatible with the dynamic component type *T* of the array (remember the example from Section 6.1 in which it was not possible to guarantee this statically). If all checks are successful, the array entry at position i can be overwritten with the new value, and the resulting array can be stored in the heap.

Fetching the length of an array is easy:

```
$exec\text{-}instr\ (ArrLength)\ hp\ ihp\ stk\ regs\ C'\ sig\ pc\ z\ frs =$
    $let\ r\quad = hd\ stk;$
        $xp' = raise\text{-}system\text{-}xcpt\ (r{=}Null)\ NullPointer;$
        $(T,l,en) = the\text{-}arr\ (hp\ (the\text{-}Addr\ r));$
        $pc' = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
    $in\ (xp',\ hp,\ ihp,\ (Intg\ l\#(tl\ stk),\ regs,\ C',\ sig,\ pc',\ z)\#frs)$
```

ArrLength expects an array reference on the operand stack, tests for *Null*, and puts the length l of the array back on top of the stack.

The last of the new instructions is *ArrNew*:

```
$exec\text{-}instr\ (ArrNew\ T)\ hp\ ihp\ stk\ regs\ C'\ sig\ pc\ z\ frs =$
    $let\ l\quad = the\text{-}Intg\ (hd\ stk);$
        $xp_0 = raise\text{-}system\text{-}xcpt\ (l < 0)\ NegativeArraySize;$
        $(a,xp_1) = new\text{-}Addr\ hp;$
        $xp' = if\ xp_0{=}None\ then\ xp_1\ else\ xp_0;$
        $hp' = if\ xp'{=}None\ then\ hp(a \mapsto blank\text{-}arr\ T\ l)\ else\ hp;$
        $ihp' = if\ xp'{=}None\ then\ ihp(a := Init\ (T.[]))\ else\ ihp;$
        $pc' = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
    $in\ (xp',\ hp',\ ihp',\ ((Addr\ a)\#tl\ stk,\ regs,\ C',\ sig,\ pc',\ z)\#frs)$
```

The *ArrNew* T instruction creates a new array with component type T on the heap. The length of the new array is the top value on the operand stack. If it is negative, the corresponding exception is thrown; otherwise, we request an unused heap entry (as for the *New* instruction, see for example Section 3.2.3). If this did not lead to an *OutOfMemory* exception, we use *blank-arr* $T\ l$ to create a blank array of length l and component type T. The blank array has the default value for type T (*0* or *Null*) in every entry. Arrays do not have constructors and do not need to be initialized, so we can set the corresponding *iheap* entry to *Init* $T.[]$. If there were no exceptions, *ArrNew* leaves the reference to the new array on the stack.

6.3.3 Defensive Machine

This section describes the defensive machine with arrays. Again, the outer levels stay the same: *check*, *exec-d*, and $\xrightarrow{\text{djvm}}$ are unchanged.

As for the aggressive VM, the definitions of *Getfield* and *Putfield* have to be slightly adjusted for the new heap structure. They now use *the-obj* to retrieve the object's contents and *is-obj* to test if the heap entry is indeed an object:

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Getfield}\ F\ C)\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\quad 0 < \textit{size stk} \land \textit{is-class}\ \Gamma\ C \land \textit{field}\ (\Gamma,C)\ F \neq \textit{None} \land \\
\quad (\textit{let}\ (C',T) = \textit{the}\ (\textit{field}\ (\Gamma,C)\ F);\ v = \textit{hd stk} \\
\quad \textit{in}\ C' = C \land \textit{is-Ref}\ v \land \\
\qquad (v \neq \textit{Null} \longrightarrow \textit{is-obj}\ (\textit{hp}\ (\textit{the-Addr}\ v)) \land \textit{is-init hp ihp}\ v \land \\
\qquad\qquad\quad (\textit{let}\ (D,\textit{fs}) = \textit{the-obj}\ (\textit{hp}\ (\textit{the-Addr}\ v)) \\
\qquad\quad\ \ \textit{in}\ (D,C) \in (\textit{subcls}\ \Gamma)^* \land \textit{fs}\ (F,C) \neq \textit{None} \land \textit{hp} \vdash \textit{the}\ (\textit{fs}\ (F,C)) ::\preceq T)))
\end{array}$$

$$\begin{array}{l}
\textit{check-instr}\ (\textit{Putfield}\ F\ C)\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
\ \ 1 < \textit{size stk} \land \textit{is-class}\ \Gamma\ C \land \textit{field}\ (\Gamma,C)\ F \neq \textit{None} \land \\
\ \ (\textit{let}\ (C',\ T) = \textit{the}\ (\textit{field}\ (\Gamma,C)\ F);\ v = \textit{hd stk};\ v = \textit{hd}\ (\textit{tl stk})\ \textit{in} \\
\ \ C' = C \land \textit{is-init hp ihp}\ v \land \textit{is-Ref}\ v \land \\
\ \ (v \neq \textit{Null} \longrightarrow \textit{is-obj}\ (\textit{hp}\ (\textit{the-Addr}\ v)) \land \textit{is-init hp ihp}\ v \land \\
\qquad (\textit{let}\ (D,\textit{fs}) = \textit{the-obj}\ (\textit{hp}\ (\textit{the-Addr}\ v))\ \textit{in}\ (D,C) \in (\textit{subcls}\ \Gamma)^* \land \textit{hp} \vdash v ::\preceq T)))
\end{array}$$

The safety checks in the defensive machine for *Invoke* and *Invoke-spcl* remain the same, because *Invoke* works on both objects and arrays, and *Invoke-spcl* rejects initialized references like arrays in any case.

Below, I show the definition of *check-instr* for the array instructions. They use the function *is-arr* :: *heap-entry option* ⇒ *bool* to test whether a position in the heap is defined and whether the entry is an array.

$$\begin{array}{l}
\textit{check-instr}\ (\textit{ArrLoad})\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
1 < \textit{length stk} \land \\
(\textit{let}\ (i,r) = (\textit{stk}!0,\ \textit{stk}!1)\ \textit{in is-Intg}\ i \land \textit{is-Ref}\ r \land r \neq \textit{Null} \longrightarrow \textit{is-arr}\ (\textit{hp}\ (\textit{the-Addr}\ r)))
\end{array}$$

For *ArrLoad* to be safe, there must be at least two entries on the stack: the first an integer, the second a reference. If the reference is not *Null*, the heap should contain an array at this position.

The *ArrStore* instruction is similar:

$$\begin{array}{l}
\textit{check-instr}\ (\textit{ArrStore})\ \textit{hp ihp stk regs Cl sig pc z frs} = \\
2 < \textit{length stk} \land \\
(\textit{let}\ (i,r) = (\textit{stk}!1,\ \textit{stk}!2)\ \textit{in is-Intg}\ i \land \textit{is-Ref}\ r \land r \neq \textit{Null} \longrightarrow \textit{is-arr}\ (\textit{hp}\ (\textit{the-Addr}\ r)))
\end{array}$$

Here we need three values on the stack. The first is the value to be stored. There are no requirements for this value other than that it exists, because the subtyping problem is handled via exceptions. The second and third stack entry must be an integer and an array reference, respectively.

ArrLength only needs an array reference on the stack:

$$\begin{array}{l} \textit{check-instr}\ (\textit{ArrLength})\ \textit{hp ihp stk regs Cl sig pc z frs} = \\ 0 < \textit{length stk} \wedge (\textit{let } r = \textit{hd stk in is-Ref } r \wedge r \neq \textit{Null} \longrightarrow \textit{is-arr}\ (\textit{hp}\ (\textit{the-Addr } r))) \end{array}$$

For *ArrNew T*, *T* should be a declared type, and the instruction needs an integer (the length) on the operand stack:

$$\begin{array}{l} \textit{check-instr}\ (\textit{ArrNew } T)\ \textit{hp ihp stk regs Cl sig pc z frs} = \\ 0 < \textit{length stk} \wedge \textit{is-Intg}\ (\textit{hd stk}) \wedge \textit{is-type}\ \Gamma\ T \end{array}$$

This concludes one-step execution in the defensive VM.

The defensive and aggressive VMs still have the same operational one-step semantics if there are no type errors:

Theorem 6.1 One-step execution in aggressive and defensive machines commutes if there are no type errors.

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

As the upper levels of the defensive machine are unchanged, the proof is the same as the one for Theorem 3.1 in Section 3.2.4.

The canonical start state remains as it is in Section 4.3.3.

6.4 Bytecode Verification

6.4.1 The Semilattice

The first step in the framework instantiation is the semilattice. The base type system is the same as for subroutines in Section 5.4.1. The task of this section is to make the set of basic types finite. The rest of the semilattice definition will then be canonical.

Section 6.2 already defined a maximum dimension for arrays. The definition below, for a fixed program Γ and maximum program counter *mpc*, additionally expresses that arrays never contain return addresses:

$$\begin{aligned} \textit{init-tys} \equiv\ & \{\textit{Init } T \mid T.\ \textit{is-type } \Gamma\ T \land \textit{boundedRA } (\textit{mpc}, T) \land \\ & \qquad \textit{dim } T \leq \textit{max-dim} \land (\textit{dim } T \neq 0 \longrightarrow \textit{noRA } T)\} \cup \\ & \{\textit{UnInit } C\ \textit{pc} \mid C\ \textit{pc}.\ \textit{is-class } \Gamma\ C \land \textit{pc} < \textit{mpc}\} \cup \\ & \{\textit{PartInit } C \mid C.\ \textit{is-class } \Gamma\ C\} \end{aligned}$$

Keep in mind that *boundedRA* (mpc,T) restricts return addresses to those smaller than *mpc*, and that *noRA* T is true iff the type T does not contain a return address. The precondition $\textit{dim } T \neq 0$ restricts this check to arrays.

It remains to lift *init-tys* to the stack and register structure of the bytecode verifier. We can repeat the construction of Section 5.4.1:

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

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

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

By induction on the maximum array dimension, I have shown that *ini-tys* remains finite, even though it now contains recursive types. Hence, we can use Lemma 2.10 and get:

Lemma 6.1 *(states, ⊆, ∪) is a semilattice and ⊆ satisfies the ascending chain condition on states.*

6.4.2 Applicability and Effect

This section will instantiate *app* and *eff* for the full instruction set of the μJVM with object initialization, exceptions, subroutines, and arrays. Both definitions are again subdivided into a part for normal and a part for exceptional execution.

The definitions of the types *state-type*, *state-bool*, and *method-type* are the same as in Section 5.4.2:

$$\begin{aligned} \textbf{types}\ \ \textit{state-type}\ \ &= \textit{init-ty list} \times \textit{init-ty err list} \\ \textit{state-bool}\ \ &= \textit{state-type} \times \textit{bool} \\ \textit{method-type} &= \textit{state-bool set list} \end{aligned}$$

The method context, too, is still the same as in the chapters before:

Γ	:: *program*	the program,
C'	:: *cname*	the class the method we are verifying is declared in,
mn	:: *mname*	the name of the method,
mxs	:: *nat*	maximum stack size of the method,
mxr	:: *nat*	size of the register set,
mpc	:: *nat*	maximum program counter,
rt	:: *ty*	return type of the method,
et	:: *ex-table*	exception handler table of the method.

The definition of applicability in the exception case *xcpt-app* still only requires all exceptions to be declared in the program, but the definition of *xcpt-names*, on which *xcpt-app* builds, now lists the new instructions in Figure 6.3.

Figure 6.3 indicates by the number of exceptions that arrays require a high amount of runtime checking compared to the rest of the instruction set. The most common of these checks are *ArrayIndexOutOfBounds* and *ArrayStore*. The former could be partially eliminated by an extended static analysis that for simple cases asserts that the index never leaves a certain interval (see for instance [57]). As outlined in Section 6.1, the latter is difficult to treat statically without changing the subtyping rules for arrays.

Applicability in the normal case still builds on $app' :: instr \times nat \times state\text{-}type \Rightarrow bool$. The new array instructions are discussed below; Figure 6.4 shows the full definition.

The *ArrLoad* instruction expects an integer and an array on the stack:

$$app'\ (ArrLoad,\ pc,\ (Init\ (PrimT\ Integer)\#Init\ (t.[])\#st,lt)) = True$$

The *ArrStore* instruction expects an initialized value, an integer, and an array on the stack:

$$app'\ (ArrStore,\ pc,\ (Init\ t\#Init\ (PrimT\ Integer)\#Init\ (t'.[])\#st,lt)) = True$$

A test $t \preceq t'$ is not necessary for *ArrStore* since this is checked at runtime. One might argue that programs failing this test statically should be rejected by the BV. On the other hand, a program might rely on an exception being thrown—however dubious such a programming style might be. Since bytecode verification is necessarily incomplete, $t \preceq t'$ could also be false at verification time, but true at runtime. The JVM specification is not clear on this issue, but the standard BV implementation does not seem to perform the test. In any case, it is of no consequence for type safety.

The *ArrLength* instruction only needs an array on top of the stack:

$$app'\ (ArrLength,\ pc,\ (Init\ (t.[])\#st,\ lt)) = True$$

$$\begin{array}{ll}
\textit{xcpt-names} :: \textit{instr} \times \textit{nat} \times \textit{ex-table} \Rightarrow \textit{cname list} & \\
\textit{xcpt-names}\ (\textit{Getfield}\ F\ C,\ pc,\ et) & = \textit{match NullPointer pc et} \\
\textit{xcpt-names}\ (\textit{Putfield}\ F\ C,\ pc,\ et) & = \textit{match NullPointer pc et} \\
\textit{xcpt-names}\ (\textit{New}\ C,\ pc,\ et) & = \textit{match OutOfMemory pc et} \\
\textit{xcpt-names}\ (\textit{Checkcast}\ C,\ pc,\ et) & = \textit{match ClassCast pc et} \\
\textit{xcpt-names}\ (\textit{Throw},\ pc,\ et) & = \textit{match-any pc et} \\
\textit{xcpt-names}\ (\textit{Invoke}\ C\ m\ p,\ pc,\ et) & = \textit{match-any pc et} \\
\textit{xcpt-names}\ (\textit{Invoke-spcl}\ C\ p,\ pc,\ et) & = \textit{match-any pc et} \\
\textit{xcpt-names}\ (\textit{ArrLoad},\ pc,\ et) & = (\textit{match G ArrayIndexOutOfBounds pc et})\ @ \\
 & \quad (\textit{match G NullPointer pc et}) \\
\textit{xcpt-names}\ (\textit{ArrStore},\ pc,\ et) & = (\textit{match G ArrayIndexOutOfBounds pc et})\ @ \\
 & \quad (\textit{match G NullPointer pc et})\ @ \\
 & \quad (\textit{match G ArrayStore pc et}) \\
\textit{xcpt-names}\ (\textit{ArrLength},\ pc,\ et) & = \textit{match G NullPointer pc et} \\
\textit{xcpt-names}\ (\textit{ArrNew}\ C,\ pc,\ et) & = (\textit{match G OutOfMemory pc et})\ @ \\
 & \quad (\textit{match G NegativeArraySize pc et}) \\
\textit{xcpt-names}\ (i,\ pc,\ et) & = []
\end{array}$$

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

Figure 6.3: Applicability in the exception case.

The *ArrNew T* instruction expects an integer on the stack. The type *T* must be declared in the program and must not contain return addresses. The dimension of the resulting array must not exceed the maximum dimension:

$$\begin{array}{ll}
\textit{app}'\ (\textit{ArrNew}\ T,\ pc,\ (\textit{Init}\ (\textit{PrimT Integer})\#st,lt)) = & \textit{is-type}\ \Gamma\ T \wedge \textit{noRA}\ T\ \wedge \\
 & \textit{dim}\ T{+}1 \le \textit{max-dim}
\end{array}$$

The final *app* function that lifts *app′* to *state-bool set* and combines it with *xcpt-app* remains unchanged. The definition is the same as in Section 5.4.2, p. 123.

This concludes applicability.

The exception case of the effect function *xcpt-eff* is also still the same as in Section 5.4.2. It just builds on the new *xcpt-names*:

$$
\begin{array}{ll}
\mathit{app}' :: \mathit{instr} \times \mathit{nat} \times \mathit{state\text{-}type} \Rightarrow \mathit{bool} & \\
\mathit{app}'\ (\mathit{Load\ idx},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{lt} \wedge \mathit{lt}!\mathit{idx} \neq \mathit{Err} \wedge \mathit{size\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Store\ idx},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{idx} < \mathit{size\ lt} \\
\mathit{app}'\ (\mathit{LitPush}\ v,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size\ st} < \mathit{mxs} \wedge \mathit{typeof}\ v \in \\
 & \quad \mathit{Some}\ \text{`}\ \{\mathit{NT},\ \mathit{PrimT\ Boolean},\ \mathit{PrimT\ Integer}\} \\
\mathit{app}'\ (\mathit{Getfield}\ F\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge\ \ t \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,\ t')) \\
\mathit{app}'\ (\mathit{Putfield}\ F\ C,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \quad (\exists\, t'.\ \mathit{field}\ (\Gamma,C)\ F = \mathit{Some}\ (C,t') \wedge \\
 & \quad t_2 \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge t_1 \preceq_i \mathit{Init}\ t') \\
\mathit{app}'\ (\mathit{New}\ C,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge \mathit{size\ st} < \mathit{mxs} \wedge \\
 & \quad \mathit{UnInit}\ C\ \mathit{pc} \notin \mathit{set\ st} \\
\mathit{app}'\ (\mathit{Checkcast}\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}class}\ \Gamma\ C \wedge (\exists\, r.\ t = \mathit{Init}\ (\mathit{RefT}\ r)) \\
\mathit{app}'\ (\mathit{Dup},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = \mathit{1}{+}\mathit{size\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Dup\text{-}x1},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = \mathit{2}{+}\mathit{size\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{IAdd},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = t_1 = t_2 \wedge t_1 = \mathit{Init}\ (\mathit{PrimT\ Integer}) \\
\mathit{app}'\ (\mathit{Ifcmpeq}\ b,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) & = (t_1 = t_2 \vee (\exists\, r\ r'.\ t_1 = \mathit{Init}\ (\mathit{RefT}\ r) \wedge \\
 & \quad t_2 = \mathit{Init}\ (\mathit{RefT}\ r'))) \\
\mathit{app}'\ (\mathit{Goto}\ b,\ \mathit{pc},\ s) & = \mathit{True} \\
\mathit{app}'\ (\mathit{Return},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) & = t \preceq_i \mathit{Init\ rt} \\
\mathit{app}'\ (\mathit{Throw},\ \mathit{pc},\ (\mathit{Init}\ t\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}RefT}\ t \\
\mathit{app}'\ (\mathit{Jsr}\ b,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{length\ st} < \mathit{mxs} \\
\mathit{app}'\ (\mathit{Ret}\ x,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = x < \mathit{length\ lt} \wedge (\exists\, r.\ \mathit{lt}!x{=}\mathit{OK}\ (\mathit{Init}\ (\mathit{RA}\ r))) \\
\mathit{app}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size\ ps} < \mathit{size\ st} \wedge \mathit{mn} \neq \mathit{init} \wedge \\
 & \quad \mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}) \neq \mathit{None} \wedge \\
 & \quad \mathit{let\ as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size\ ps})\ \mathit{st});\ t = \mathit{st}!\mathit{size\ ps} \\
 & \quad \mathit{in}\ t \preceq_i \mathit{Init}\ (\mathit{Class}\ C) \wedge \mathit{is\text{-}class}\ \Gamma\ C \wedge \\
 & \qquad (\forall\, (a,f){\in}\mathit{set}(\mathit{zip\ as\ ps}).\ a \preceq_i \mathit{Init}\ f) \\
\mathit{app}'\ (\mathit{Invoke\text{-}spcl}\ C\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) & = \mathit{size\ ps} < \mathit{size\ st} \wedge \\
 & \quad (\exists\, r.\ \mathit{method}\ (\Gamma,C)\ (\mathit{init},\mathit{ps}) = \mathit{Some}\ (C,r)) \wedge \\
 & \quad \mathit{let\ as} = \mathit{rev}\ (\mathit{take}\ (\mathit{size\ ps})\ \mathit{st});\ t = \mathit{st}!\mathit{size\ ps} \\
 & \quad \mathit{in\ is\text{-}class}\ \Gamma\ C \wedge \\
 & \qquad ((\exists\, \mathit{pc}.\ t = \mathit{UnInit}\ C\ \mathit{pc}) \vee \\
 & \qquad t = \mathit{PartInit}\ C' \wedge (C',C){\in}\mathit{subcls}\ \Gamma) \wedge \\
 & \qquad (\forall\, (a,f){\in}\mathit{set}(\mathit{zip\ as\ ps}).\ a \preceq_i (\mathit{Init}\ f)) \\
\mathit{app}'\ (\mathit{ArrLoad},\mathit{pc},(i\#\mathit{Init}\ (t.[])\#\mathit{st},\mathit{lt})) & = (i = \mathit{Init}\ (\mathit{PrimT\ Integer})) \\
\mathit{app}'\ (\mathit{ArrStore},\ \mathit{pc},\ (\mathit{Init}\ t\#i\#\mathit{Init}\ (t'.[])\#\mathit{st},\mathit{lt})) & = (i = \mathit{Init}\ (\mathit{PrimT\ Integer})) \\
\mathit{app}'\ (\mathit{ArrLength},\ \mathit{pc},\ (\mathit{Init}\ (t.[])\#\mathit{st},\ \mathit{lt})) & = \mathit{True} \\
\mathit{app}'\ (\mathit{ArrNew}\ T,\ \mathit{pc},\ (l\#\mathit{st},\mathit{lt})) & = \mathit{is\text{-}type}\ \Gamma\ T \wedge l = \mathit{Init}\ (\mathit{PrimT\ Integer}) \wedge \\
 & \quad \mathit{noRA}\ T \wedge \mathit{dim}\ T{+}\mathit{1} \leq \mathit{max\text{-}dim} \\
\mathit{app}'\ (i,\ \mathit{pc},\ s) & = \mathit{False}
\end{array}
$$

Figure 6.4: Applicability of instructions with arrays.

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

It remains to define the normal, non-exception case for *eff* and to combine the two cases into the final effect function. The successors definition *succs* is the same as in Section 5.4.2, Figure 5.6. The new instructions are caught by the default case [*pc+1*]. Figure 6.5 shows the full definition of the effect *eff'* on single type configurations.

The old instructions are unchanged, for the array instructions, *eff'* is defined as follows:

$$\begin{array}{lcl}
\textit{eff}'\ (\textit{ArrLoad},\ pc,\ (i\#\textit{Init}\ (t.[])\#st,lt)) & = & (\textit{Init}\ t\#st,lt) \\
\textit{eff}'\ (\textit{ArrStore},\ pc,\ (v\#i\#a\#st,lt)) & = & (st,lt) \\
\textit{eff}'\ (\textit{ArrLength},\ pc,\ (a\#st,lt)) & = & (\textit{Init}\ (\textit{PrimT Integer})\#st,lt) \\
\textit{eff}'\ (\textit{ArrNew}\ T,\ pc,\ (l\#st,lt)) & = & (\textit{Init}\ (T.[])\#st,lt)
\end{array}$$

ArrLoad removes index and array reference, and puts a value of the component type *t* on the stack. *ArrStore* only pops the value, the index, and the array reference from the stack; the heap access is statically not visible. *ArrLength* replaces the array reference by an integer, and *ArrNew T* pushes an array with component type *T* onto the stack.

The function *norm-eff* that lifts *eff'* to *state-bool set* is the same as in Section 5.4.2, and the final effect function is canonical:

$$\begin{array}{l}
\textit{eff} :: \textit{instr} \Rightarrow \textit{nat} \Rightarrow \textit{state-bool set} \Rightarrow (\textit{nat} \times \textit{state-bool set})\ \textit{list} \\
\textit{eff i pc s} \equiv (\textit{map}\ (\lambda pc'.\ (pc',\ \textit{norm-eff i pc pc}'\ s))\ (\textit{succs i pc s}))\ @\ (\textit{xcpt-eff i pc s})
\end{array}$$

This concludes the definition of the transfer function.

6.4.3 Executable Bytecode Verifiers

With the semilattice and the transfer function of Section 6.4.1 and Section 6.4.2, this section again instantiates the framework and yields two executable bytecode verifiers.

Contrary to the chapters before, all the definitions of *wt-start*, *wt-method*, *wt-jvm-prog*, *wt-kil*, $\textit{wt-jvm-prog}_k$, *wt-lbv*, and $\textit{wt-jvm-prog}_l$ remain as defined in Section 5.4.3. The changes only take place in the underlying semilattice and transfer function.

The following main lemma still holds:

$$\mathit{eff}' :: \mathit{instr} \times \mathit{nat} \times \mathit{state\text{-}type} \Rightarrow \mathit{state\text{-}type}$$

$$\begin{aligned}
&\mathit{eff}'\ (\mathit{Load\ idx},\ \mathit{pc},\ (\mathit{st},\ \mathit{lt})) &&= (\mathit{ok\text{-}val}\ (\mathit{lt}!\mathit{idx})\#\mathit{st},\ \mathit{lt})\\
&\mathit{eff}'\ (\mathit{Store\ idx},\ \mathit{pc},\ (t\#\mathit{st},\ \mathit{lt})) &&= (\mathit{st},\ \mathit{lt}[\mathit{idx}{:=}\ \mathit{OK}\ t])\\
&\mathit{eff}'\ (\mathit{LitPush}\ v,\ \mathit{pc},\ (\mathit{st},\ \mathit{lt})) &&= (\mathit{Init}\ (\mathit{the}\ (\mathit{typeof}\ v))\#\mathit{st},\ \mathit{lt})\\
&\mathit{eff}'\ (\mathit{Getfield}\ F\ C,\ \mathit{pc},\ (t\#\mathit{st},\ \mathit{lt})) &&= (\mathit{Init}\ (\mathit{snd}\ (\mathit{the}\ (\mathit{field}\ (\Gamma,C)\ F)))\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{Putfield}\ F\ C,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) &&= (\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{New}\ C,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) &&= (\mathit{UnInit}\ C\ \mathit{pc}\#\mathit{st},\ \mathit{replace}\ (\mathit{OK}\ (\mathit{UnInit}\ C\ \mathit{pc}))\ \mathit{Err}\ \mathit{lt})\\
&\mathit{eff}'\ (\mathit{Checkcast}\ C,\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) &&= (\mathit{Init}\ (\mathit{Class}\ C)\ \#\ \mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{Dup},\ \mathit{pc},\ (t\#\mathit{st},\mathit{lt})) &&= (t\#t\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{Dup\text{-}x1},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) &&= (t_1\#t_2\#t_1\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{IAdd},\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) &&= (\mathit{Init}\ (\mathit{PrimT}\ \mathit{Integer})\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{Ifcmpeq}\ b,\ \mathit{pc},\ (t_1\#t_2\#\mathit{st},\mathit{lt})) &&= (\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{Goto}\ b,\ \mathit{pc},\ s) &&= s\\
&\mathit{eff}'\ (\mathit{Jsr}\ t,\ \mathit{pc},\ (\mathit{st},\mathit{lt})) &&= ((\mathit{Init}\ (\mathit{RA}\ (\mathit{pc}{+}1)))\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{Ret}\ x,\ \mathit{pc},\ s) &&= s\\
&\mathit{eff}'\ (\mathit{Invoke}\ C\ \mathit{mn}\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) &&= \mathit{let}\ \mathit{st}' = \mathit{drop}\ (1{+}\mathit{size}\ \mathit{ps})\ \mathit{st};\\
& &&\qquad (\_,\mathit{rt},\_,\_,\_) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{mn},\mathit{ps}))\\
& &&\quad \mathit{in}\ (\mathit{Init}\ \mathit{rt}\#\mathit{st}',\ \mathit{lt})\\
&\mathit{eff}'\ (\mathit{Invoke\text{-}spcl}\ C\ \mathit{ps},\ \mathit{pc},\ (\mathit{st},\mathit{lt})) &&= \mathit{let}\ t = \mathit{st}!\mathit{size}\ \mathit{ps};\ i\ = \mathit{Init}\ (\mathit{theClass}\ t);\\
& &&\qquad \mathit{st}'' = \mathit{drop}\ (1{+}\mathit{size}\ \mathit{ps})\ \mathit{st};\\
& &&\qquad \mathit{st}' = \mathit{replace}\ t\ i\ \mathit{st}'';\\
& &&\qquad \mathit{lt}' = \mathit{replace}\ (\mathit{OK}\ t)\ (\mathit{OK}\ i)\ \mathit{lt};\\
& &&\qquad (\_,\mathit{rt},\_,\_,\_) = \mathit{the}\ (\mathit{method}\ (\Gamma,C)\ (\mathit{init},\mathit{ps}))\\
& &&\quad \mathit{in}\ (\mathit{Init}\ \mathit{rt}\#\mathit{st}',\ \mathit{lt}')\\
&\mathit{eff}'\ (\mathit{ArrLoad},\mathit{pc},(i\#\mathit{Init}\ (t.[])\#\mathit{st},\mathit{lt})) &&= (\mathit{Init}\ t\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{ArrStore},\ \mathit{pc},\ (v\#i\#a\#\mathit{st},\mathit{lt})) &&= (\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{ArrLength},\ \mathit{pc},\ (a\#\mathit{st},\mathit{lt})) &&= (\mathit{Init}\ (\mathit{PrimT}\ \mathit{Integer})\#\mathit{st},\mathit{lt})\\
&\mathit{eff}'\ (\mathit{ArrNew}\ T,\ \mathit{pc},\ (l\#\mathit{st},\mathit{lt})) &&= (\mathit{Init}\ (t.[])\#\mathit{st},\mathit{lt})
\end{aligned}$$

Figure 6.5: Effect of instructions on the state type.

Lemma 6.2 The transfer function *step*, built from *app* and *eff* as described in Section 2.3.3 and Section 6.4.2, is monotone, bounded, and type preserving (w.r.t. *states* and *size ins*).

The proof that *step* is bounded and monotone remains unaltered, because we do not need to look at single instructions for these properties. The proof that *step* is type preserving contains four new cases, and the old cases additionally have to assert that they do not introduce return addresses into arrays and that they do not cause array dimensions to grow larger that the maximum size. With a few basic lemmas about array dimensions, Isabelle handles this change in the old cases automatically.

The soundness and completeness results of $\textit{wt-jvm-prog}_k$ and $\textit{wt-jvm-prog}_l$ only build on the framework, on Lemma 6.2, and on the start values defined in the instantiation. Since all these remain unchanged, the soundness and completeness proofs can stay unchanged as well. For reference, I repeat the results below:

Theorem 6.2 The executable BV is sound and recognizes all welltyped programs:

$$\textit{wt-jvm-prog}_k\ \Gamma = (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

Theorem 6.3 If the LBV accepts a program, it is welltyped:

$$\textit{wt-jvm-prog}_l\ \Gamma\ \textit{Cert} \longrightarrow (\exists\,\Phi.\ \textit{wt-jvm-prog}\ \Gamma\ \Phi)$$

Theorem 6.4 The LBV accepts every welltyped program:

$$\textit{wt-jvm-prog}\ \Gamma\ \Phi \longrightarrow \textit{wt-jvm-prog}_l\ \Gamma\ (\textit{mk-cert}\ \Phi)$$

As before, I have generated ML code for both $\textit{wt-jvm-prog}_k$ and $\textit{wt-jvm-prog}_l$, showing that they are fully executable.

6.5 Type Safety

This section presents the type safety theorem. It implies that the bytecode verifier guarantees type safe execution, now with object initialization, bytecode subroutines, exceptions, and arrays. With Theorem 6.1, this again implies that the checks of the defensive machine are redundant and the aggressive machine can be used safely instead.

The theorem is still:

Theorem 6.5 If C is a class in Γ with a *main* method, then

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

The proof structure with the main lemmas and the invariant argument also remains the same:

Lemma 6.3 Conformance is invariant during execution in welltyped programs.

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

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

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

Lemma 6.5 An execution step started in a conformant state cannot produce a type error in welltyped programs.

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

This time it is not hard to adjust the invariant. The most important building block, single value conformance $hp \vdash v ::\preceq T$, already takes arrays into account (see Section 6.2). For the rest of the conformance relation, we merely have to look for occurrences of a direct heap access in the predicates.

The first of these occurrences is heap conformance:

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

$$\begin{array}{l}
\textit{econf} :: \textit{aheap} \Rightarrow \textit{heap-entry} \Rightarrow \textit{bool}\\
\textit{econf hp entry} \equiv \textit{case entry of}\\
\qquad\qquad\qquad \textit{Obj}\ C\ \textit{fs} \Rightarrow \textit{lconf hp fs}\ (\textit{map-of}\ (\textit{fields}\ (\Gamma, C)))\\
\qquad\qquad\quad\ \mid \textit{Arr}\ T\ l\ \textit{en} \Rightarrow \textit{lconf hp en}\ (\lambda x.\ \textit{if}\ 0 \leq x \wedge x < l\ \textit{then Some}\ T\ \textit{else None})
\end{array}$$

$$\begin{array}{l}
(\text{-}\ \surd) :: \textit{aheap} \Rightarrow \textit{bool}\\
hp\ \surd \equiv \forall a\ \textit{entry}.\ hp\ a = \textit{Some entry} \longrightarrow \textit{econf hp entry}
\end{array}$$

In the definition above, there is now a predicate *econf* for heap entries instead of *oconf* for objects. The object branch is the same as the old *oconf*, and the array branch says that for an array of length l and component type T at least the array indices from 0 to l must be defined, and that the value at these positions must conform to T.

Unsurprisingly, *iheap* conformance undergoes the same structural change:

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

$$\begin{array}{l}
\textit{e-init} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{heap-entry} \Rightarrow \textit{bool} \\
\textit{e-init hp ih entry} \equiv \textit{case entry of} \\
\qquad\qquad\qquad \textit{Obj C fs} \Rightarrow \textit{l-init G hp ih fs}\ (\textit{map-of}\ (\textit{fields}\ (\Gamma,C))) \\
\qquad\qquad\quad\ \mid \textit{Arr T l en} \Rightarrow \textit{l-init hp ih en}\ (\lambda x.\ \textit{if}\ 0 \le x \wedge x < l\ \textit{then Some T else None})
\end{array}$$

$$\begin{array}{l}
\textit{h-init} :: \textit{aheap} \Rightarrow \textit{iheap} \Rightarrow \textit{bool} \\
\textit{h-init hp ih} \equiv \forall\, a\ \textit{entry}.\ \textit{hp}\ a = \textit{Some entry} \longrightarrow \textit{e-init hp ih entry}
\end{array}$$

This is all that needs to be done for arrays. The rest remains as in Section 5.5.

The type safety proof itself is still by case distinction over the instruction set, now with four new cases for the four new instructions. With some basic lemmas about single value conformance and arrays, the change in heap structure is handled gracefully by Isabelle's automatic tactics: manual adjustments for the old instructions are few and simple. They mostly consist of replacing explicit statements of the form $hp\ x = (C,fs)$ by $hp\ x = Obj\ C\ fs$.

6.6 Conclusion

In this chapter, I have presented the last and largest stage of the framework instantiations for the μJava language. The two executable verified bytecode verifiers now support classes, objects, inheritance, virtual methods, exception handling, constructors, object initialization, bytecode subroutines, and arrays.

The formalization grew by 1,600 lines from 13,600 to 15,200 lines of Isabelle code (333 pages in the resulting proof document). The main workload this time was for the new heap model, the definition of the operational semantics of the new instructions (aggressive and defensive), and the four new cases in the type safety proof.

The additional proof burden for the four new instructions was small (compared to, say, *Invoke-spcl*), and the changes to the formalization were restricted mainly to the virtual machine and the abstract transfer function. Contrary to object initialization or subroutines, the additional complexity in the type system only played a minor role, namely in establishing that the set of basic types is still finite.

This suggests that the framework presented in this thesis does not only scale well for additional features of the BV, but also that adding new instructions is a painless procedure: extend the definitions of virtual machine and transfer function, and prove the additional type safety cases. The rest of the formalization is not affected as long as the execution model and the type system do not change drastically.

As for the other type systems, the full specification and Isabelle proofs are available from the VerifiCard project web site [89].

7 Conclusion

7.1 Summary

In this thesis, I have presented a fully formal, executable, and machine checked specification of a representative subset of the Java Virtual Machine and its bytecode verifier, together with a proof that this bytecode verifier is correct.

The abstract framework for bytecode verification in Chapter 2 proved powerful and flexible enough to handle four different type systems, the most expressive of which covers all important properties of Java bytecode verification: classes, objects, inheritance, virtual methods, exception handling, constructors, object initialization, bytecode subroutines, and arrays.

The instantiations each yielded two executable verified bytecode verifiers: an iterative data flow algorithm and a lightweight bytecode verifier.

All specifications and proofs have been carried out in the interactive theorem prover Isabelle/HOL. They are available on the web [89] as proof documents in PDF format and also as Isabelle sources. All formal proofs in this thesis are machine checked and generated directly from Isabelle sources.

The following table gives an overview of the size of the framework and the four JVM formalizations. Each formalization is an extension of the one above it, and contains all BV features of the formalization before.

Formalization	Lines of Isabelle code	Pages of proof document
Framework	3,500	87
Exceptions	11,100	245
Object Initialization	14,300	303
Subroutines	13,600	293
Arrays	15,200	333

This thesis mainly contributes to the following areas:

Java Platform The Java programming platform and execution environment has been the target of large, international research efforts and is now well understood. This thesis has shown that the type system used in the JVM and the BV is safe and can be proved correct in a theorem prover. I have formalized and verified all important features of the BV, including properties like object initialization that go beyond type safety. The formalization in this thesis is one of the most comprehensive formalizations of the BV that have been published, and it is the most comprehensive one in a theorem prover. The paper formalization of Freund [26] is comparable in the set of properties of the BV, but the type system he presents is very complex and contains subtle problems (see also Chapter 4 and 5). The formalization of Stärk et al. [81, 82, 83] also includes a comparable BV, but proofs are short and sketchy. Their treatment of subroutines requires a stronger definite assignment condition than the Java Language Specification, whereas mine accepts more safe programs than the specification requires. The formalization of Barthe et al. [5, 6] contains more instructions, but their main focus is the JVM, so they only have a simplified treatment of subroutines in the BV.

Methodology The abstract typing framework in this thesis has made it possible to cleanly distinguish between executable algorithm and type system. It enabled a uniform treatment of Kildall's algorithm and lightweight bytecode verification leading to a lightweight bytecode verifier that handles properties beyond both the original version by Rose [74, 75] and the industrial version by Sun Microsystems [87, 88]: object initialization is not treated by Rose and subroutines are treated neither by Rose nor by Sun. Each new type system that meets the conditions of the framework (a semilattice with an order satisfying the ascending chain condition and a monotone, type preserving transfer function) automatically gives rise to two executable, correct algorithms. It is important to note that these algorithms are not only executable in theory, but that ML prototypes have been generated from the specification. The formal distinction between applicability and effect makes the BV specification modular and clear. The readable Isabelle/Isar proofs allow a deeper insight into interesting lemmas in the type safety argument. An earlier version of the μJVM formalization has served Basin et al. [7] as a basis for their correctness proof of the model checking approach to bytecode verification. This suggests that not only the framework but also the instantiation is an excellent starting point for further work in the area of bytecode verification and static analysis of machine code. The abstract and modular structure of the framework makes it possible to integrate new algorithms as well as new type systems into the existing formalization.

Type Systems This thesis provides a study of multiple type systems for the BV. I have discussed the merging type systems (for exceptions and object initialization) that are used in current BV implementations, and the more recent set based type system that is especially useful for bytecode subroutines (and in this thesis also used for arrays). For the merging type system with object initialization I have shown the first proof of correctness in a theorem prover that considers a representative subset of the JVM. For the set based type system I have also presented the first proof of correctness in a theorem prover, and I have shown that the approach scales to a representative, object oriented subset of the JVM. The original version in [15, 16] did not even contain classes. The set based type system is very expressive in the sense that it accepts a large set of type safe programs. Leroy's work [50] suggests that the formalization presented here is an excellent basis for optimizations of the type system combining the expressiveness of the set based approach with the efficiency of the merging type systems.

7.2 Further Work

The formalization in this thesis can serve as a basis for further work in several dimensions:

Breadth An obvious direction is extending the instruction set in the formalization to the full set of instructions supported by Java. As the experience in Chapter 6 suggests, the modular style of the framework should make this relatively easy. The difficult type system related problems have been solved in this thesis, the remaining problem is the sheer number of instructions. The work of Casset et al. [12, 13, 14, 72] suggests that with a smart grouping of instructions factoring out common properties is possible and will reduce the work load significantly. Judging from their description in [72], the automation capabilities of Isabelle as a general purpose theorem prover for higher order logic are significantly better than those of the B-method tool Casset et al. used, so the task seems entirely possible.

Features There are some features of the BV this thesis has not treated. The most notable of these is interfaces. They pose a problem to bytecode verification, because with interfaces, a type system that uses Java's subtyping relation as ordering is not a semilattice anymore: the least common supertype is not unique, because a class or interface may have multiple superinterfaces. For the set based approach this does not matter, because the order is the subset relation and interfaces can be introduced without harm. For merging type systems there exist two solutions in the literature, one by Qian [69] and one by Knoblock and Rehof [42, 43]. The one by Qian has been formalized and proved correct in Isabelle by Pusch [68]. It should not be difficult to integrate her formalization into the one presented here.

The approach in the Sun verifier is not documented in the JVM specification, but experiments show that it is very simple: the Sun verifier treats all interfaces as class *Object* and the `invokeinterface` instruction checks dynamically whether the reference implements the specified interface.

Wide data types are handled in the BV by introducing types like `double_low` and `double_high`. Only the corresponding instructions can access values of these types, and they can only use them together. Wide instructions are merely a decoding issue. Preventing jumps into the middle of an instruction encoding can for instance be achieved by marking program counters in the middle of instruction encodings as invalid and by testing for this mark in the applicability part of the transfer function.

As they are checked at runtime in the JVM instead of statically, packages and access modifiers are more of an issue for the operational semantics and for the method and field lookup functions than for the BV. Schirmer [76] has shown how to integrate packages and access modifiers into the existing Bali formalization which is similar to μJava but considers the source level only. The part for declarations and lookup functions can directly be transferred to μJava, the part for runtime checks in the μJVM should be analogous to the checking rules Schirmer formalized for the source language.

Class loading Class loading is a feature of the JVM that is interesting for bytecode verification because the class loader is part of the name space for class resolving (together with the package and class name). Java makes it possible to define and use one's own class loaders. By tricking the BV into believing that two classes loaded by different class loaders are equal, type safety could be broken. The BV in this thesis takes the approach used on the JavaCard platform and assumes that all classes of the program have been preloaded and that no dynamic class loading at runtime is possible. This assumption is expressed by the wellformedness predicate for programs: the class hierarchy must be acyclic for the algorithm to be correct. To test this, all classes of the program must be known. Since this condition is only necessary for termination, it might be possible to lift the restriction to something weaker. Qian, Goldberg, and Coglio [29, 71] have formalized Java class loading. One line of further work might investigate how to integrate the solution they propose with the formalization presented here.

Threads Threads again mostly concern the JVM; the monitor enter and exit instructions that programs use for synchronization are checked at runtime. Laneve and Bigliardi [44, 45, 46] have implemented a bytecode verifier that checks structured handling of thread monitors statically. Structured handling means that each lock acquired for an object is released, regardless of whether the method returns normally or exceptionally. This is complicated by the fact that there may be locks on different objects acquired and released in arbitrary order and that references to objects may be aliased. The effort for extending the present formalization by checks for structured monitor handling should be moderate. The alias analysis that is required is comparable in complexity to the one for object initialization in Chapter 4; the type safety proof should be easier than the one for object initialization because the condition for structured monitor handling is local to each method and does not require reasoning over something like constructor chains. A larger part of the effort will probably be not the BV but extending the JVM formalization by monitors and concurrency.

Beyond type safety The typing framework and the JVM formalization presented here can be used for more than pure type safety. The framework is extensible in two dimensions: new algorithms that can be proved correct w.r.t. the framework, and new instantiations of the existing algorithms. An interesting candidate for a new algorithm is Leroy's on-card verifier for Java smart cards [48]. It relies on a previous transformation of the bytecode program, but after this transformation, verification can be performed with very low memory requirements. It should be possible to integrate this algorithm into the framework and prove it safe. On the instantiation side, the possibilities for further research are numerous: everything that can be expressed by a semilattice type system is suitable, including many standard static analysis problems. Simple examples are interval analysis for eliminating array bounds checks in the JVM, or a null pointer analysis that can be used both to eliminate null pointer checks in the JVM and to alert the programmer to potential problems in the program. More difficult and interesting properties that could be expressed by type systems include information flow, time constraints, simple termination properties, and resource usage. The framework would provide both the standard data flow analysis and also the lightweight verification algorithm. The latter turns checking these properties into something that is very similar to a framework for proof carrying code.

List of Figures

Bibliography

[1] Bali project website. `http://isabelle.in.tum.de/bali`, 2002.

[2] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proc. TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[3] Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In H. Kirchner and C. Ringeissen, editors, *Proc. AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer-Verlag, 2002.

[4] Gilles Barthe, Guillaume Dufay, Marieke Huisman, Simão Melo de Sousa, and Bernard Paul Serpette. Jakarta: a toolset to reason about the JavaCard platform. In I. Attali and T. Jensen, editors, *Proc. e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.

[5] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Simão Melo de Sousa, and Bernard Paul Serpette. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proc. ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.

[6] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Simão Melo de Sousa, and Bernard Paul Serpette. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proc. VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 32–45. Springer-Verlag, 2002.

[7] David Basin, Stefan Friedrich, and Marek Gawkowski. Verified Bytecode Model Checkers. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proc. TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 47–66. Springer-Verlag, 2002.

[8] David Basin, Stefan Friedrich, Joachim Posegga, and Harald Vogt. Java bytecode verification by model checking. System abstract. In N. Halbwachs and D. Peled,

editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 491–494. Springer-Verlag, 1999.

[9] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES'00)*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer-Verlag, 2002.

[10] Yves Bertot. Formalizing a JVML Verifier for Initialization in a Theorem Prover. In A. Finkel G. Berry, H. Comon, editor, *Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 14–24. Springer-Verlag, 2001.

[11] Pascal Brisset. Vers un vérifieur de bytecode Java certifié. Seminar given at Ecole Normale Supérieure, Paris, October 1998. (According to [50]).

[12] Ludovic Casset. Development of an embedded verifier for Java Card byte code using formal methods. In L.-H. Eriksson and P. A. Lindsay, editors, *Proc. FME'02: Formal Methods - Getting IT Right, International Symp. Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 290–309. Springer-Verlag, 2002.

[13] Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an Embedded Verifier for Java Card Byte Code. In *Proc. DSN'02 (International Conference on Dependable Systems and Networks)*, pages 51–58. IEEE Computer Society Press, 2002.

[14] Ludovic Casset and Jean Luis Lanet. A formal specification of the Java bytecode semantics using the B method. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP'99 Workshop Formal Techniques for Java Programs*, pages 1–7. Technical Report 251, Fernuniversität Hagen, 1999.

[15] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. Technical Report, Kestrel Institute, December 2001.

[16] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. In *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs.* Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 2002.

[17] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proc. 2nd ECOOP*

Workshop on Formal Techniques for Java Programs. Technical Report 269, Fernuniversität Hagen, 2000.

[18] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

[19] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2*, pages 403–410. IEEE Computer Society Press, 2000.

[20] Allesandro Coglio. Improving the official specification of Java bytecode verification. In *3rd ECOOP Workshop on Formal Techniques for Java programs*, 2001.

[21] Allesandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, 2001.

[22] Richard Cohen. The defensive Java virtual machine specification. Technical report, Computational Logic Inc., 1997. `http://www.cli.com/software/djvm/`.

[23] The Coq proof assistant. `http://coq.inria.fr/`, 2002.

[24] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy: May 6–8, 1996, Oakland, California*, pages 190–200. IEEE Computer Society Press, 1996.

[25] Stephen N. Freund. The costs and benefits of Java bytecode subroutines. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

[26] Stephen N. Freund. *Type Systems for Object-Oriented Intermediate Languages.* PhD thesis, Stanford University, 2000.

[27] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[28] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1999.

[29] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*, 1998.

[30] Li Gong. *Inside Java 2 Platform Security*. The Java Series. Addison Wesley, 1999.

[31] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

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

[33] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In G. Levi, editor, *Static Analysis (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 1998.

[34] Pieter H. Hartel and Luc A. V. Moreau. Formalizing the safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[35] Isabelle website. `http://isabelle.in.tum.de/`, 2002.

[36] Gary A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[37] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000.

[38] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001.

[39] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. 48 pages, to appear.

[40] Gerwin Klein and Martin Strecker. Verified Bytecode Verification and type-certifying Compilation. *Journal of Logic Programming*, 2002. 41 pages, submitted for publication.

[41] Gerwin Klein and Martin Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning, Special Issue on Bytecode Verification*, 2003. 38 pages, submitted for publication.

[42] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for Java bytecode. In *Proc. POPL'00, 27th ACM SIGPLAN-SIGACT Sympòn Principles of Programming Languages*, pages 228–242, Boston, Massachusetts, 2000.

[43] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for Java bytecode. *ACM Transactions on Programming Languages and Systems*, 23(2):243–272, 2001.

[44] Cosimo Laneve. A type system for JVM threads. *Theoretical Computer Science*, 209(1):741–778, 2002.

[45] Cosimo Laneve and Gaetano Bigliardi. A type system for JVM threads. Technical Report UBCLS-2000-06, University of Bologna, July 2000.

[46] Cosimo Laneve and Gaetano Bigliardi. A type system for JVM threads. In *The Third ACM SIGPLAN Workshop on Types in Compilation (TIC'00)*. Technical Report CMU-CS-00-161, Carnegie Mellon University, 2000.

[47] Xavier Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[48] Xavier Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Smart card programming and security, Proc. e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2001.

[49] Xavier Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.

[50] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 2003. To appear.

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

[52] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes and Antidotes.* John Wiley and Sons, 1996.

[53] Gary McGraw and Edward W. Felten. *Securing Java: Getting Down to Business with Mobile Code.* John Wiley and Sons, 2nd edition, 1999.

[54] Sun Microsystems. JavaCard Technology. `http://java.sun.com/products/javacard/`, 2002.

[55] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[56] George C. Necula. Proof-carrying code. In *Proc. POPL'97, 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[57] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[58] Tobias Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS'01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, 2001.

[59] Tobias Nipkow. A compact introduction to structured proofs in Isar/HOL. In *Types for Proofs and Programs (TYPES'02)*, 2002. Submitted for publication.

[60] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

[61] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[62] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. POPL'98, 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[63] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. POPL'99, 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 70–78. ACM Press, 1999.

[64] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.

[65] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer-Verlag, 1999.

[66] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods – Getting IT Right (FME'02), International Symp. Formal*

Methods Europe, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.

[67] Joachim Posegga and Harald Vogt. Byte code verification for Java smart cards based on model checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Computer Security — ESORICS 98*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.

[68] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 1999.

[69] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–311. Springer-Verlag, 1999.

[70] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Programming Languages and Systems*, 22(4):638–672, 2000.

[71] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. *ACM SIGPLAN Notices*, 35(10):325–336, 2000.

[72] Antoine Requet. A B model for ensuring soundness of a large subset of the Java Card virtual machine. *Science of Computer Programming*, 46:283–306, 2003. Online version appeared at `http://www.sciencedirect.com/`.

[73] Eva Rose. Towards Secure Bytecode Verification on a Java Card. Master's thesis, DIKU, University of Copenhagen, 1998.

[74] Eva Rose. *Vérification de Code d'Octet de la Machine Virtuelle Java. Formalisation et Implantation.* PhD thesis, Université Paris VII, 2002.

[75] Eva Rose and Kristoffer Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

[76] Norbert Schirmer. Analysing the Java Package/Access concepts in Isabelle/HOL. In *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs.* Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 2002.

[77] Joachim Schmid. *Introduction to AsmGofer*, 2001. `http://www.tydo.org/AsmGofer/`.

[78] David A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proc. POPL'98, 25th ACM SIGPLAN-SIGACT Sympȯn Principles of Programming Languages*, pages 38–48, San Diego, California, 1998.

[79] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3:30–50, 2000.

[80] Emin Gün Sirer, Sean McDirmid, and Brian Bershad. Kimera: A Java system security architecture. Technical report, University of Washington, 1997.

[81] Robert Stärk and Joachim Schmid. Java bytecode verification is not possible. In R. Moreno-Díaz and A. Quesada-Arencibia, editors, *Formal Methods and Tools for Computer Science (Proc. Eurocast'01)*, pages 232–234, 2001.

[82] Robert Stärk and Joachim Schmid. The problem of bytecode verification in current implementations of the JVM. Technical report, Department of Computer Science, ETH Zürich, 2001.

[83] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.

[84] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. POPL'98, 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 149–161. ACM Press, 1998.

[85] Martin Strecker. Formal verification of a Java compiler in Isabelle. In A. Voronkov, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2002.

[86] Martin Strecker. Investigating type-certifying compilation with Isabelle. In M. Baaz and A. Voronkov, editors, *Proc. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*, pages 403–417. Springer-Verlag, 2002.

[87] Sun Microsystems. CLDC and the K Virtual Machine (KVM), 2000. `http://java.sun.com/products/cldc/`.

[88] Sun Microsystems. Connected, limited device configuration, version 1.0, May 2000. `http://java.sun.com/aboutJava/communityprocess/final/jsr030/`.

[89] VerifiCard website, Munich. `http://isabelle.in.tum.de/verificard/`, 2002.

[90] David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and

A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000.

[91] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[92] Markus Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002.

[93] Martin Wildmoser. Subroutines and Java Bytecode Verification. Master's thesis, Institut für Informatik, Technische Universität München, 2002.

[94] Phillip M. Yelland. A compositional account of the Java virtual machine. In *Proc. POPL'99, 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 57–69. ACM Press, 1999.

Index