Verified Bytecode Verification and Type-Certifying Compilation

Gerwin Klein, Martin Strecker

Technische Universität München, Fakultät für Informatik, D-85748 Garching

Abstract

This article presents a type certifying compiler for a subset of Java and proves the type correctness of the bytecode it generates in the proof assistant Isabelle. The proof is performed by defining a type compiler that emits a type certificate and by showing a correspondence between bytecode and the certificate which entails well-typing. The basis for this work is an extensive formalization of the Java bytecode type system, which is first presented in an abstract, lattice-theoretic setting and then instantiated to Java types.

Key words: Java, JVM, Compiler, Bytecode Verification, Theorem Proving

1 Introduction

This paper provides an in-depth analysis of type systems in compilation, by taking the Java source language and Java bytecode as examples and showing that the bytecode resulting from compiling a type correct source program yields type correct bytecode.

We do not cover all language constructs of Java and neglect some subtleties, in particular exceptions and the jump-subroutine mechanism, while otherwise using a faithful model of Java and the Java Virtual Machine (JVM). It is an advance of this work over previous investigations of this kind that the definitions and proofs have been done entirely within the Isabelle verification assistant, resulting in greater conceptual clarity, as far as notation is concerned, and

Email addresses: `Gerwin.Klein@in.tum.de` (Gerwin Klein), `Martin.Strecker@in.tum.de` (Martin Strecker).

[1] This research is funded by the EU project *VerifiCard*

a more precise statement of theorems and proofs than can be achieved with pencil-and-paper formalizations (see Section 7 for a discussion).

Type correctness of bytecode produced by our compiler, *comp*, is proved by having a type compiler, *compTp*, emit a type certificate and showing that this certificate is a correct type of the code, in a sense to be made precise. This type certificate is related to (even though not identical with) what would be inferred by a bytecode verifier (BCV). Transmitting such a certificate along with bytecode and then checking its correctness is an attractive alternative to full bytecode verification, in particular for devices with restricted resources such as smart cards. The idea of using separate type certificates is not novel (see the concept of "lightweight bytecode verification" [RR98,KN01,Ros02]); however, we are not aware of a Java compiler other than ours which explicitly generates them.

Apart from this potential application, compilation of types, in analogy to compilation of code, gives insight into type systems of programming languages and how they are related. Incompatibilities discovered in the source and bytecode type systems of Java [SS01] demonstrate the need for such a study. Even though these inconsistencies do not arise in the language subset we examine, we hope to cover larger fragments with the same techniques as presented below.

The work described here is part of a larger effort aiming at formalizing diverse aspects of the Java language, such as its operational and axiomatic semantics [Ohe01], its bytecode type system and bytecode verifier [Kle03] and the correctness (in the sense of preservation of semantics) of a compiler [Str02a].

This article extends on a previous paper [Str02b] by providing an in-depth exposition of the bytecode type system and by presenting large parts of the formalization of the type compiler and a detailed discussion of the type preservation proof. As far as we are aware, ours is the first fully formal treatment covering these aspects of Java in one comprehensive model.

In the following, we will first summarize the most important concepts of our Java and JVM formalization (Section 2). The bytecode type system and its relation to bytecode verification are further elaborated in Section 3. We define the code compiler *comp* in Section 4, the type compiler *compTp* in Section 5. The type correctness statement for generated code and a detailed discussion of the proof follow in Section 6. Section 7 concludes with a discussion of related and future work.

Due to space limitations, we can only sketch our formalization. The full Isabelle sources are available from `http://isabelle.in.tum.de/verificard/`.

2 Language Formalizations

In this section, we give an overview of Isabelle and describe the existing formalizations of Java in Isabelle: the source language, μJava, and the Java virtual machine language, μJVM. This reduced version of Java [NOP00] accommodates essential aspects of Java, like classes, subtyping, object creation, inheritance, dynamic binding and exceptions, but abstracts away from most arithmetic data types, interfaces, arrays and multi-threading. It is a good approximation of the JavaCard dialect of Java, targeted at smart cards.

2.1 An Isabelle Primer

Isabelle is a generic framework for encoding different object logics. In this paper, we will only be concerned with Isabelle/HOL [NPW02], which comprises a higher-order logic and facilities for defining data types as well as primitive and terminating general recursive functions.

Isabelle's syntax is reminiscent of ML, so we will only mention a few peculiarities: Consing an element `x` to a list `xs` is written as `x#xs`. Infix `@` is the append operator, `xs ! n` selects the `n`-th element from list `xs`.

We have the usual type constructors `T1 × T2` for product and `T1 ⇒ T2` for function space. The long arrow $\Longrightarrow$ is Isabelle's meta-implication, in the following mostly used in conjunction with rules of the form $[\![P_1; \ldots; P_n]\!] \Longrightarrow C$ to express that `C` follows from the premises $P_1 \ldots P_n$. Apart from that, there is the implication $\longrightarrow$ of the HOL object logic, along with the standard connectives and quantifiers.

The polymorphic option type

```
datatype 'a option = None | Some 'a
```

is frequently used to simulate partiality in a logic of total functions: Here, `None` stands for an undefined value, `Some x` for a defined value `x`. Lifted to function types, we obtain the type of "partial" functions `T1 ⇀ T2`, which just abbreviates `T1 ⇒ (T2 option)`.

The constructor `Some` has a left inverse, the function `the :: 'a option ⇒ 'a`, defined by the sole equation `the (Some x) = x`. This function is total in the sense that also `the None` is a legal, but indefinite value. Another frequently used term describing an indefinite value is the polymorphic `arbitrary`.

Ultimately, indefinite values are defined with Hilbert's ϵ operator. They denote

a fixed, but otherwise unknown value of their respective type.[2] In particular, they cannot be shown to be equal to any specific value of the type. Thus, we cannot prove an equation like `f arbitrary = arbitrary`. Indefinite values are therefore not "undefined" values in the sense of denotational semantics. One consequence is, for example, that an indefinite value delivered by the source semantics and mapped to the bytecode level is not equal to an indefinite value delivered by the bytecode semantics. We therefore always have to ensure that we deal with defined values – see Section 4.2.

2.2 Java Source Language

2.2.1 Terms and Programs

The Java language is embedded deeply in Isabelle, i.e. by an explicit representation of the Java term structure as Isabelle datatypes. We make the traditional distinction between expressions `expr` and statements `stmt`. The latter are standard, except maybe for `Expr`, which turns an arbitrary expression into a statement (this is a slight generalization of Java). For some constructs, more readable mixfix syntax is defined, enclosed in brackets.

```
datatype expr
  = NewC cname
  | Cast cname expr
  | Lit val
  | BinOp binop expr expr
  | LAcc vname
  | LAss vname expr      (_::=_)
  | FAcc cname expr vname
  | FAss cname expr vname
  | Call cname expr mname  (ty list) (expr list) ({_}_..._( {_}_))

datatype stmt = Skip
              | Expr expr
              | Comp stmt stmt       (_;; _)
              | Cond expr stmt stmt  (If (_) _ Else _)
              | Loop expr stmt       (While (_) _)
```

The μJava expressions form a representative subset of Java: `NewC` creates a new instance, given a class name `cname`; `Cast` performs a type cast; `Lit` embeds values `val` (see below) into expressions. μJava contains only a few binary operations `binop`: test for equality and integer addition. There is access to

[2] Since types in HOL are guaranteed to be non-empty, such an element always exists.

local variables with *LAcc*, given a variable name *vname*; assignment to local variables *LAss*; and similarly field access, field assignment and method call. The type annotations contained in braces *{ }* are not part of the original Java syntax; they have been introduced to facilitate type checking.

The type *val* of values is defined by

```
datatype val = Unit | Null | Bool bool | Intg int | Addr loc
```

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

The μJava types *ty* are either primitive types or reference types. *Void* is the result type of void methods; note that *Boolean* and *Integer* are not Isabelle types, but simply constructors of *prim_ty*. Reference types are the null pointer type *NullT* or class types. We abbreviate *RefT (ClassT C)* by *Class C* and *RefT NullT* by *NT*.

```
datatype prim_ty = Void | Boolean | Integer
datatype ref_ty  = NullT | ClassT cname
datatype ty      = PrimT prim_ty | RefT ref_ty
```

On this basis, we define field declarations *fdecl* and a method signatures *sig* (method name and list of parameter types). A method declaration *mdecl* consists of a method signature, the method return type and the method body, whose type is left abstract. The method body type *'c* remains a type parameter of all the structures built on top of *mdecl*, in particular *class* (superclass name, list of fields and list of methods), class declaration *cdecl* (holding in addition the class name) and program *prog* (list of class declarations).

```
types  fdecl    = vname × ty
       sig      = mname × ty list
       'c mdecl = sig × ty × 'c
       'c class = cname × fdecl list × 'c mdecl list
       'c cdecl = cname × 'c class
       'c prog  = 'c cdecl list
```

By instantiating the method body type appropriately, we can use these structures both on the source and on the bytecode level. For the source level, we take *java_mb prog*, where *java_mb* consists of a list of parameter names, list of local variables (i.e. names and types), and a statement block, terminated with a single result expression (this again is a deviation from original Java).

```
types  java_mb   = vname list × (vname × ty) list × stmt × expr
       java_prog = java_mb  prog
```

2.2.2 Typing

Typing judgements come in essentially two flavours:

- `E ⊢ e :: T` means that expression `e` has type `T` in environment `E`. We write `wtpd_expr E e` for `∃T. E ⊢ e :: T`.
- `E ⊢ c √` means that statement `c` is well-typed in environment `E`.

The *environment* `E` used here is `java_mb env`, a pair consisting of a Java program `java_mb prog` and a local environment `lenv`.

In order to convey a feeling for the typing rules, we give a particularly unspectacular one:

$$[\![\ E \vdash e :: PrimT\ Boolean;\ \ E \vdash s1\surd;\ \ E \vdash s2\surd\ \ ]\!] \Longrightarrow E \vdash If(e)\ s1\ Else\ s2\surd$$

It says that a conditional is well-typed provided the expression `e` is Boolean and the statements `s1` and `s2` are well-typed.

A program `G` is well-formed (`wf_java_prog G`) if the bodies of all its methods are well-typed and in addition some structural properties are satisfied – mainly that all class names are distinct and the superclass relation is well-founded.

2.2.3 Operational Semantics

The operational semantics, in the style of a big-step (natural) semantics, describes how the evaluation of expressions and statements affects the program state, and, in the case of an expression, what is the result value. The semantics is defined as inductive relation, again in two variants:

- for expressions, `G ⊢ s -e≻ v-> s'` means that for program `G`, evaluation of `e` in state `s` yields a value `v` and a new state `s'` (note that the evaluation of expressions may have side-effects).
- for statements, `G ⊢ s -c→ s'` means that for program `G`, execution of `c` in state `s` yields a new state `s'`.

The *state* (of type `xstate`) is a triple, consisting of an optional exception component that indicates whether an exception is active, a heap `aheap` which maps locations `loc` to objects, and a local variable environment `locals` mapping variable names to values.

$$\begin{aligned} aheap &= loc \rightsquigarrow obj \\ locals &= vname \rightsquigarrow val \\ state &= aheap \times locals \\ xstate &= val\ option \times state \end{aligned}$$

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).

```
obj = cname × (vname × cname ⇒ val option)
```

The semantics has been designed to be non-blocking even in the presence of certain errors such as type errors. For example, dynamic method binding is achieved via a method lookup function *method* that selects the method to be invoked, given the dynamic type *dynT* of expression *e* (whereas *C* is the static type) and the method signature (i.e. method name *mn* and parameter types *pTs*). Again, the method *m* thus obtained is indefinite if either *dynT* does not denote a valid class type or the method signature is not defined for *dynT*.

```
⟦ ...;  m = the (method (G,dynT) (mn,pTs)); ... ⟧ ⟹
          G⊢Norm s0 -{C}e..mn({pTs}ps)≻v-> s'
```

The evaluation rules could be formulated differently so as to exclude indefinite values, at the expense of making the rules unwieldy, or they could block in the case of type errors, which would make a type correctness statement impossible (see [Ohe01] for a discussion). Fortunately, the type safety results provided in the following show that this kind of values does not arise anyway. Unfortunately, the rules force us to carry along this type safety argument in the compiler correctness proof.

2.2.4 Conformance and Type-Safety

The type-safety statement requires as auxiliary concept the notion of *conformance*, which is defined in several steps:

- Conformance of a value *v* with type *T* (relative to program *G* and heap *h*), written *G, h⊢v::≼T*, means that the dynamic type of *v* under *h* is a subtype of *T*.
- Conformance of an object means that all of its fields conform to their declared types.
- Finally, a state *s* conforms to an environment *E*, written as *s::≼ E*, if all "reachable" objects of the heap of *s* conform and all local variables of *E* conform to their declared types.

The type safety theorem says that if evaluation of an expression *e* well-typed in environment *E* starts from a conforming state *s*, then the resulting state is again conforming; in addition, if no exception is raised, the result value *v* conforms to the static type *T* of *e*. An analogous statement holds for evaluation of statements.

2.3 Java Bytecode

We shall now take a look at the μJava VM (Section 2.3.1) and its operational semantics, first without (Section 2.3.2) and then with (Section 2.3.3) runtime type checks.

2.3.1 State Space

The runtime environment, i.e. 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).

```
jvm_state  =  val option × aheap × frame list
```

The heap is the same as on the source level: a partial function from locations to objects.

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

```
    frame  =  opstack × registers × cname × sig × nat
  opstack  =  val list
registers  =  val list
```

2.3.2 Operational semantics

This section sketches the state transition relation of the μJava VM. Figure 1 shows the instruction set. Method bodies are lists of such instructions together with the exception handler table and two integers `mxs` and `mxl` containing the maximum operand stack size and the number of local variables (not counting the `this` pointer and parameters of the method which get stored in the first `0` to `n` registers). So the type parameter `'c` for method bodies gets instantiated with `nat × nat × instr list × ex_table`, i.e. `mdecl` becomes the following:

```
mdecl = sig × ty × nat × nat × instr list × ex_table
```

As exceptions are not yet handled by the compiler we do not define `ex_table` formally here.

```
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 and push 2 values down
    | IAdd                              integer addition
    | Goto int                          goto relative address
    | Ifcmpeq int                       branch if equal
    | Throw                             throw exception
```

Fig. 1. The μJava bytecode instruction set.

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

The 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 (G,C) sig));
    in find_handler (exec_instr (ins!pc) 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. Again, we leave out the formal definition of `find_handler`, because the compiler does not handle exceptions. As throughout the rest of this article, the program `G` is treated as a global parameter.

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

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

The definition of `exec_instr` is straightforward, but large. We only show one example here, the `Load idx` instruction: it takes the value at position `idx` in the register list and puts it on top of the stack. Apart from incrementing the program counter the rest remains untouched:

```
exec_instr (Load idx) hp stk regs Cl sig pc frs =
    (None, hp, ((regs ! idx) # stk, regs, Cl, sig, pc+1) # frs)
```

This style of VM is also called *aggressive*, because it does not perform any runtime type or sanity checks. It just assumes that everything is as expected, e.g. for `Load idx` that the index `idx` indeed is a valid index of the register set, and that there is enough space on the stack to push it. 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.

2.3.3 A Defensive VM

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

To indicate type errors, we introduce another datatype.

```
'a type_error = TypeError | Normal 'a
```

Similar to Section 2.3.2 we build on a function `check_instr` that is lifted over several steps. At the deepest level, we take apart the state to feed `check_instr` with parameters (which are the same as for `exec_instr`) and check that the `pc` is valid:

```
check :: jvm_state ⇒ bool
check (xp, hp, [])    = True
check (xp, hp, f#frs) = let (stk,reg,C,sig,pc) = f;
                            ins = 5th (the (method (G,C) sig));
in pc < size ins ∧ check_instr (ins!pc) hp stk reg C sig pc frs
```

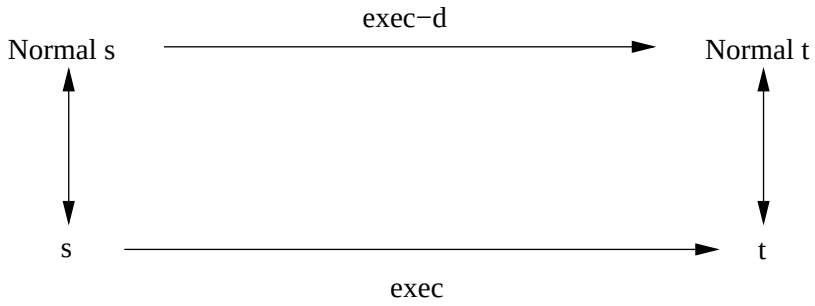


Fig. 2. Aggressive and defensive μJVM commute if there are no type errors.

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

```
exec_d :: jvm_state type_error ⇒ jvm_state option type_error
exec_d TypeError = TypeError
exec_d (Normal s) = if check s then Normal (exec s) else TypeError
```

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

```
s -djvm-> t ≡
(s,t) ∈ ({(s,t)|exec_d s = TypeError ∧ t = TypeError} ∪
         {(s,t)|∃t'. exec_d s = Normal (Some t') ∧ t = Normal t'})*
```

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

```
check_instr IAdd hp stk regs Cl sig pc frs =
1<size stk ∧ isIntg (hd stk) ∧ isIntg (hd (tl stk))
```

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

Theorem 1 *One step execution in aggressive and defensive machines commutes if there is no type error.*

```
exec_d (Normal s)≠TypeError ⟹ exec_d (Normal s)=Normal (exec s)
```

Figure 2 depicts this result as a commuting diagram.

For executing programs we will later also need a canonical start state. In the real JVM a program is started by invoking its static `main` method. In the μJVM this is similar. We call a method *main method* of class `C` if there is a method body `b` such that `method (G,C) (main,[]) = Some (C, b)` holds. For `main` methods we can define the canonical start state `start G C` as the state with exception flag `None`, an otherwise empty heap `start_hp G` that has

preallocated system exceptions [3], and a frame stack with one element: empty operand stack, `this` pointer set to `Null`, the rest of the register set filled up with a dummy value `arbitrary`, the class entry set to `C`, signature to `(main,[])` and program counter `0`.

```
start :: jvm_prog ⇒ cname ⇒ jvm_state
start G C ≡ let (_,_,_,mxl,_,_) = the (method (G,C) (main,[]));
                              regs = Null # replicate mxl arbitrary
     in Normal (None, start_hp G, [([], regs, C, (main,[]), 0)])
```

3 Bytecode Verification

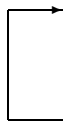
We begin the part about bytecode verification with an informal introduction in Section 3.1. The μJava bytecode verifier is then built in two steps: Section 3.2 presents an abstract typing framework, Section 3.3 instantiates it for the μJVM. The type safety theorem in Section 3.4 shows that the bytecode verifier we have constructed is sound.

3.1 An Example

Bytecode verification is an abstract interpretation of the bytecode program: instead of values we only consider their types. This abstraction allows us to view a program as a finite state machine working on so called *state types*. A state type characterizes a set of runtime states by giving type information for the operand stack and registers. For example the first state type in Figure 3 `([],[Class B, Int])` characterizes all states whose stack is empty, whose register `0` contains a reference to an object of class `B` (or to a subclass of `B`), and whose register `1` contains an integer. We say a method is *welltyped* if we can assign 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.

The example in Figure 3 shows the instructions on the left and the type of stack elements and registers on the right. The method type is the right-hand side of the table, a state type is one line of it. The type information attached

[3] We use preallocated system exceptions in the style of JavaCard to circumvent the unspecified situation where there is no space left to create a new `OutOfMemory` exception object.



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

Fig. 3. Example of a welltyping

to an instruction characterizes the state *before* execution of that instruction. We assume that class `B` is a subclass of `A` and that `A` has a field `F` of type `A`.

Execution starts with an empty stack and the two registers hold a reference to an object of class `B` and an integer. The first instruction loads register `0`, a reference to a `B` object, on the stack. The type information associated with the following instruction may puzzle at first sight: it says that a reference to an `A` object is on the stack, and that usage of register `1` may produce an error. This means the type information has become less precise but is still correct: a `B` object is also an `A` object and an integer is now classified as unusable (`Err`). The reason for these more general types is that the predecessor of the `Store` instruction may have either been `Load 0` or `Goto -3`. Since there exist different execution paths to reach `Store`, the type information of the two paths has to be "merged". The type of the second register is either `Int` or `Class A`, which are incompatible, i.e. the only common supertype is `Err`. The `Some` before each of the type entries means that we were able to predict some type for each of the instructions. If one of the instructions had been unreachable, the type entry would have been `None`.

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 3 was an example for a welltyped method (we were able to find a welltyping). Had 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.

3.2 An Abstract Framework

The abstract framework for data flow analysis is independent of the JVM, its typing rules, and instruction set. Since it is a slightly extended version of the

framework already presented in [Nip01] and (with more detail) in [KN02], we concentrate on the general setting and the result of the data flow analysis. We leave out the data flow analysis itself, i.e. Kildall's algorithm.

3.2.1 Orders and semilattices

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

Partial orders Partial orders are formalized as binary predicates. Based on the type synonym `'a ord = 'a ⇒ 'a ⇒ bool` and the two order notations `x ≤r y = r x y` and `x <r y = (x ≤r y ∧ x ≠ y)` we say that `r` is a *partial order* iff the predicate `order :: 'a ord ⇒ bool` holds for `r`:

```
order r = (∀x. x ≤r x) ∧ (∀x y. x ≤r y ∧ y ≤r x ⟶ x=y) ∧
          (∀x y z. x ≤r y ∧ y ≤r z ⟶ x ≤r z)
```

Semilattices Based on the type synonyms `'a binop = 'a ⇒ 'a ⇒ 'a` and `'a sl = 'a set × 'a ord × 'a binop` and the notation `x +f y = f x y` for the supremum, we call the tuple `(A,r,f) :: 'a sl` a *semilattice* iff the predicate `semilat :: 'a sl ⇒ bool` holds:

```
semilat (A,r,f) = order r ∧ closed A f ∧
                  (∀x y∈A. x ≤r x +f y) ∧ (∀x y∈A. y ≤r x +f y) ∧
                  (∀x y z∈A. x ≤r z ∧ y ≤r z ⟶ x +f y ≤r z)
```

where `closed A f = ∀x y∈A. x +f y ∈ A`.

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

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

```
datatype 'a err = Err | OK 'a
err A = {Err} ∪ {OK a | a ∈ A}
```

An ordering `r` on `'a` can be lifted to `'a err` by making `Err` the top element. To do so, we define a functional `le` that takes an existing partial order `r :: 'a ord` and lifts it to `'a err ord`.

```
le r (OK x) (OK y)  =  x ≤r y
le r _      Err     =  True
le r Err    (OK y)  =  False
```

The following lifting functional is useful below:

```
lift2 :: ('a ⇒ 'b ⇒ 'c err) ⇒ 'a err ⇒ 'b err ⇒ 'c err
lift2 f (OK x) (OK y)  =  f x y
lift2 f    _      _    =  Err
```

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

```
'a ebinop = 'a ⇒ 'a ⇒ 'a err
'a esl    = 'a set × 'a ord × 'a ebinop
```

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

```
esl :: 'a sl ⇒ 'a esl
esl(A,r,f) = (A,r,λx y. OK(f x y))

sl :: 'a esl ⇒ 'a err sl
sl(A,r,f) = (err A,le r,lift2 f)
```

Now we define `L :: 'a esl` to be an *err-semilattice* iff `sl L` is a semilattice. It follows easily that `esl L` is an err-semilattice if `L` is a semilattice. In a strongly typed environment like HOL we found err-semilattices easier to work with than semilattices with top element.

3.2.2 Welltypings

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

Data flow analysis and type systems are based on an abstract view of the semantics of a program in terms of types instead of values. Since our programs are sequences of instructions the semantics can be characterized by two functions `app :: nat ⇒ 's ⇒ bool` and `eff :: nat ⇒ 's ⇒ (nat × 's) list`.

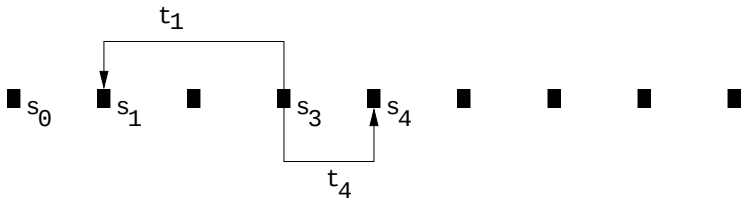


Fig. 4. Data flow graph for *eff* 3 s_3 = $[(1,t_1),(4,t_4)]$

While *app* checks if an instruction is applicable in the current state type, *eff* is the abstract execution function: *eff 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, *eff 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 4 depicts the information we get when *eff 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 that the structure of the data flow graph dynamically changes in the iteration process of the analysis. We will not use this flexibility to its fullest extent in this article, but it is necessary to handle more advanced features of the BCV like the *Jsr*/*Ret* instructions.

The correctness and termination theorem about the dataflow analysis algorithm imposes several restrictions (like monotonicity) on *app*, *eff*, and the semilattice order *r* (see [Nip01,KN02] for more). Here, we are only interested in the nature of welltypings, and for that no such restrictions are necessary. Using the semilattice order and the functions *app* and *eff*, we can define when a method type φ :: 's list is a welltyping:

wt_app_eff $\varphi \equiv$
  $\forall$ *p<size* φ. *app p* (φ*!p*) $\wedge$ ($\forall$ *(q,t)*$\in$*set(eff p* (φ*!p*)). *t* $\leq_r$ φ*!q)*

This is very natural: every instruction is applicable in its start state, and the effect is compatible with the state expected by all successor instructions. The JVM specification also requires a start condition to be met. We shall come to that in the next section.

3.3 Instantiating the Framework

In the following sections we shall instantiate the abstract typing framework with a concrete type system for the μJVM. We define the semilattice structure

in Section 3.3.1, the data flow functions `app` and `eff` in Section 3.3.2, and finally refine the notion of welltyping to Java-specifics in Section 3.3.3.

3.3.1 The Semilattice

In this section we take the first step to instantiate the framework of Section 3.2. We define the semilattice structure on which μJava's bytecode verifier builds. We begin by turning the μJava types `ty` into a semilattice.

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

```
types = {T| is_type G T}
```

The order is the standard subtype ordering $\preceq$ of μJava. It builds on the direct subclass relation `subcls G` induced by the program `G`.

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

The expression `(C,D)` $\in$ `(subcls G)`* means that `C` is a subclass of `D`. For every class hierarchy, i.e. for every program, this subtype ordering may be a different one. In the Isabelle formalization the ordering $\preceq$ therefore has `G` as an additional parameter, in this paper `G` is implicit.

The supremum operation follows the ordering.

```
sup :: ty ⇒ ty ⇒ 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 t1         t2        = if t1 = t2 then OK t1 else Err
```

The `lub` function computes the least upper bound of two classes by walking up the class hierarchy until one is a subclass of the other. Since, in a well-formed program, every class is a subclass of `Object`, this least upper bound is guaranteed to exist. We call a program `G` *wellformed* if each subclass has at most one direct superclass, i.e. `G` represents a single inheritance hierarchy, and if `subcls G` is acyclic.

With these three components we proved the following theorem.

Theorem 2 *The triple* `JType.esl` $\equiv$ `(types,` $\preceq$`, sup)` *is an err-semilattice provided the program* `G` *is wellformed.*

We can now construct the stack and register structure. State types in the

μJava BCV are the same as in the example in Figure 3: values on the operand stack must always contain a known μJava type `ty`, values in the local variables may be of an unknown type and therefore be unusable (encoded by `Err`). To handle unreachable code, the BCV will not directly work on `state_type`, but on `state_type option` instead. If `None` occurs in the welltyping, the corresponding instruction is unreachable. A method type is then a list of such state types.

```
state_type  =  ty list × ty err list
method_type =  state_type option list
```

It is easy to prove

Theorem 3 *If `G` is wellformed then method types form an err-semilattice.*

The three components of the semilattice are `states` (the carrier set), `<='` (the subtype ordering lifted pointwise to stack and registers), and `sup` (the supremum, also lifted pointwise).

The executable BCV of [Nip01,KN02] contains an additional `Err` layer on top which turns the err-semilattice into a proper semilattice and which is used to indicate type errors in the data flow analysis. Since we are only interested in the result of the analysis, the welltyping, we have left it out here.

3.3.2 Applicability and Effect

In this section we will 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. We only show the definitions for the normal case here.

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

`G :: program`	the program,
`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,
`pc :: nat`	program counter of the current instruction.

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

We begin with applicability of instructions in the normal, non-exception case.

```
app' :: instr × state_type ⇒ bool

app' (Load idx, (ST,LT))       = idx < LT ∧ LT!idx ≠ Err ∧
                                 size ST < mxs
app' (Store idx, (T#ST,LT))    = idx < size LT
app' (LitPush v, (ST,LT))      = size ST < mxs ∧
                                 typeof v ≠ None
app' (Getfield F C, (T#ST,LT)) = is_class G C ∧ T ≼ Class C ∧
                                (∃T'. field (G,C) F = Some (C, T'))
app' (Putfield F C, (T1#T2#ST,LT)) = is_class G C ∧
                                (∃T'. field (G,C) F = Some (C,T') ∧
                                 T2 ≼ Class C ∧ T1 ≼ T')
app' (New C, (ST,LT))          = is_class G C ∧ size ST < mxs
app' (Checkcast C, T#ST,LT))   = is_class G C ∧ isRefT T
app' (Dup, (T#ST,LT))          = 1+size ST < mxs
app' (Dup_x1, (T1#T2#ST,LT))   = 2+size ST < mxs
app' (IAdd, (T1#T2#ST,LT))     = T1 = T2 ∧ T1 = PrimT Integer
app' (Ifcmpeq b,(T1#T2#ST,LT)) = 0 ≤ int pc + b ∧ ((T1 = T2) ∨
                                 (isRefT T1 ∧ isRefT T2))
app' (Goto b, s)               = 0 ≤ int pc + b
app' (Return, (T#ST,LT))       = T ≼ rT
app' (Throw, (T#ST,LT))        = isRefT T
app' (Invoke C mn ps, (ST,LT)) = size ps < size ST ∧
                                 is_class G C ∧
                                 method (G,C) (mn,ps) ≠ None ∧
                                 let as = rev (take (size ps) ST);
                                     t  = ST!size ps
                                 in t ≼ Class C ∧ as [≼] ps
app' (i,s)                     = False
```

Fig. 5. Applicability of instructions.

We ignore the `option` layer at first: `app'`, defined in Figure 5, works on `state_type`, `app` then lifts it to `state_type option`.

In `app'`, a few new functions occur: `typeof :: val ⇒ ty option` returns `None` for addresses, and the type of the value otherwise; `field` is analogous to `method` and looks up declaration information of object fields (defining class and type); `rev` and `take` are the obvious functions on lists, `[≼]` pointwise lifts the subtyping relation ≼ to lists.

With `app'`, we can now build the full applicability function `app`: an instruction is applicable when it is unreachable (then it can do no harm) or when it is applicable in the normal and in the exceptional case (`xcpt_app`). Additionally,

```
succs :: instr ⇒ nat ⇒ nat list

succs (Ifcmpeq b) pc = [pc+1, nat (int pc + b)]
succs (Goto b) pc    = [nat (int pc + b)]
succs Return pc      = []
succs Throw pc       = []
succs i pc           = [pc+1]
```

Fig. 6. Successor program counters for the non-exception case.

we require that the *pc* does not leave the instruction sequence.

```
app :: instr ⇒ state_type option ⇒ bool
app i s ≡ case s of None ⇒ True
                  | Some s ⇒ xcpt_app i ∧ app' (i,s) ∧
                              (∀(pc',s')∈set (eff i s). pc'<mpc)
```

This concludes applicability. It remains to build the effect function *eff*. In *eff* we must calculate the successor program counters together with new state types. We define them separately in Figure 6.

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 if there is no exception.

As with *app* we first define the effect *eff'* on *state_type* (Figure 7). The destructor *ok_val* is defined by *ok_val (OK x) = x*. The large *method* expression for *Invoke* merely determines the return type of the method in question. Note that it must drop *1+size ps* elements form the stack: the parameters and the reference on which the method was invoked.

```
eff' :: instr × state_type ⇒ state_type

eff' (Load idx, (ST,LT))         = (ok_val (LT!idx)#ST, LT)
eff' (Store idx, (T#ST,LT))      = (ST, LT[idx:= OK T])
eff' (LitPush v, (ST,LT))        = (the (typeof v)#ST, LT)
eff' (Getfield F C,(T#ST,LT))    = (snd (the (field (G,C) F))#ST,LT)
eff' (Putfield F C, (T1#T2#ST,LT)) = (ST,LT)
eff' (New C, (ST,LT))            = (Class C # ST,LT)
eff' (Checkcast C, (T#ST,LT))    = (Class C # ST,LT)
eff' (Dup, (T#ST,LT))            = (T#T#ST,LT)
eff' (Dup_x1, (T1#T2#ST,LT))     = (T1#T2#T1#ST,LT)
eff' (IAdd, T1#T2#ST,LT))        = (PrimT Integer#ST,LT)
eff' (Ifcmpeq b, (T1#T2#ST,LT)) = (ST,LT)
eff' (Invoke C mn ps, (ST,LT))   = let ST' = drop (1+size ps) st;
                                   (_,rT,_,_,_) = the (method (G,C) (mn,ps))
                                   in (rT#ST', LT)
```

Fig. 7. Effect of instructions on the state type.

We use `option_map :: ('a ⇒ 'b) ⇒ 'a option ⇒ 'b option` to lift functions to the option type canonically:

```
option_map f None = None
option_map f (Some x) = Some (f x)
```

Lifting `eff'` to `state_type option` is then:

```
norm_eff :: instr ⇒ state_type option ⇒ state_type option
norm_eff i s ≡ option_map (λs. eff' (i,s))
```

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

```
eff :: instr ⇒ state_type option ⇒ (nat × state_type option) list
eff i s ≡
  (map (λpc'. (pc', norm_eff i s)) (succs i pc)) @ (xcpt_eff i s)
```

3.3.3 Welltypings

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

The framework of Section 3.2 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 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: when method `m` in class `C` is invoked, the stack is empty, the first register contains the `this` pointer (of type `Class C`), the next registers contain the parameters of `m`, the rest of the registers is reserved for local variables (which do not have a value yet). As above, `ps` are the parameters, and `mxl` the number of local variables (which is related to `mxr` by `mxr = 1+size ps+mxl`). Below, for the compiler, this context will be expanded. The `<='` is the semilattice order on `state_type option` of Section 3.3.1.

```
wt_start φ ≡
  Some ([],(OK (Class C))#(map OK ps)@(replicate mxl Err)) <=' φ!0
```

We call φ a welltyping, if it satisfies `wt_method`.

```
wt_method φ ≡ 0 < mpc ∧ wt_start φ ∧ wt_app_eff φ
```

For the type compiler it is useful to have a more fine-grained version of `wt_app_eff` for single instructions:

$$\texttt{wt\_instr}\ p\ \varphi \equiv \texttt{app}\ p\ (\varphi!p) \wedge (\forall (q,t) \in \texttt{set(eff}\ p\ (\varphi!p)).\ t <=' \varphi!q)$$

With this, we get the following equality for `wt_method`:

$$\texttt{wt\_method}\ \varphi = 0 < \texttt{mpc} \wedge \texttt{wt\_start}\ \varphi \wedge (\forall p < \texttt{mpc}.\ \texttt{wt\_instr}\ p\ \varphi)$$

It remains to lift welltypings from methods to programs. Welltypings of programs are functions Φ `:: cname` $\Rightarrow$ `sig` $\Rightarrow$ `state_type option list` that return a welltyping for each method and each class in the program. We call a program welltyped if there is a welltyping Φ such that `wt_jvm_prog G` Φ holds. The function `wt_jvm_prog` returns true if `wt_method (`Φ `C sig)` holds for every `C` and `sig` such that `C` is a class in `G` and `sig` a method signature declared in `C`. Additionally, `wt_jvm_prog` checks that `G` is wellformed, i.e. that the class hierarchy is a well founded single inheritance hierarchy.

3.4 Type Safety

This section presents the type safety theorem. It says that the bytecode verifier is correct, that it guarantees safe execution. If the bytecode verifier succeeds and we start the program `G` in its canonical start state (see Section 2.3.3), the defensive μJVM will never return a type error.

Theorem 4 *If `C` is a class in `G` with a* `main` *method, then*

$$[\![\ \texttt{wt\_jvm\_prog}\ G\ \Phi;\ \texttt{start}\ G\ C \xrightarrow{\text{djvm}} \tau\ ]\!] \Longrightarrow \tau \neq \texttt{TypeError}$$

To prove this theorem, we set out from a program `G` for which the bytecode verifier returns true, i.e. for which there is a Φ such that `wt_jvm_prog G` Φ holds. The proof builds on the observation that all runtime states σ that conform to the types in Φ are type safe. If σ conforms to Φ, we write $\Phi \vdash \sigma\surd$. For $\Phi \vdash \sigma\surd$ to be true, the following must hold: if in state σ execution is at position `pc` of method `(C,sig)`, then the state type `(`Φ `C sig)!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`. We have shown that conformance is invariant during execution if the program is welltyped.

Lemma 5 *Conformance is invariant during execution in welltyped programs.*

$$[\![\ \texttt{wt\_jvm\_prog}\ G\ \Phi;\ \Phi \vdash \sigma\surd;\ \sigma \xrightarrow{\text{jvm}} \tau\ ]\!] \Longrightarrow \Phi \vdash \tau\surd$$

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 (e.g. non-empty stack). Then we execute the instruction and observe that the new state τ conforms to the corresponding *t* in *eff pc s*.

For the proof to go through, the intuitive notion of conformance we have given above is not enough, the formal conformance relation $\Phi \vdash \sigma\surd$ is stronger. It describes the states that can occur during execution, the form of the heap, and the form of the method invocation stack. As it is very large (about four pages of pure Isabelle code) and [Pus99,NOP00,KN02] already contain detailed descriptions of it, we will not formally define the full conformance relation here.

Lemma 5 is still not enough, though: it might be the case that there is no σ such that $\Phi \vdash \sigma\surd$. Lemma 6 shows that this is not so.

Lemma 6 *If* C *is a class in* G *with a* main *method, then*

$$wt\_jvm\_prog\ G\ \Phi \Longrightarrow \Phi \vdash (start\ G\ C)\surd$$

Lemmas 5 and 6 together say that all states that occur in any execution of program G conform to Φ if we start G in the canonical way.

The last step in the proof of Theorem 4 is Lemma 7.

Lemma 7 *An execution step started in a conformant state cannot produce a type error in welltyped programs.*

$$[\![\ wt\_jvm\_prog\ G\ \Phi;\ \Phi \vdash \sigma\surd\ ]\!] \Longrightarrow exec\_d\ (Normal\ \sigma) \neq TypeError$$

The proof of Lemma 7 is a case distinction on the current instruction in σ. Similar to the proof of Lemma 5, the conformance relation together with the *app* part of *wt_jvm_prog* ensure *check_instr* in *exec_d* returns true. Because we know that all states during execution conform, we can conclude Theorem 4: there will be no type errors in welltyped programs.

4 Compiling Code

4.1 Definition of Compiler

Compilation is defined with the aid of a few directly executable functions. Expressions resp. statements are compiled by *compExpr* and *compStmt*, whose definitions we give in Figure 8 resp. Figure 9 for comparison with the type compilation functions defined in Section 5.

The compiler definitions are straightforward: Apart from the expression resp. statement to be compiled, the functions take a *java_mb* as argument. It is required to compute a mapping from variable names to indices in the register array, which is accomplished by function *index*.

Note that our compiler makes no attempt at optimizing generated code. For example, in order to maintain the invariant used in the compiler correctness statement, the bytecode for an assignment expression of the form *vn::=e* contains the instruction *Dup* which duplicates the value on top of the operand stack. When used as an assignment statement of the form *Expr (vn::=e)*, this and the following *Pop* instruction are superfluous.

```
compExpr  :: java_mb ⇒ expr ⇒ instr list
compExprs :: java_mb ⇒ expr list ⇒ instr list

compExpr jmb (NewC c) = [New c]
compExpr jmb (Cast c e) = compExpr jmb e @ [Checkcast c]
compExpr jmb (Lit val) = [LitPush val]
compExpr jmb (BinOp bo e1 e2) = compExpr jmb e1 @ compExpr jmb e2 @
 (case bo of
   Eq  ⇒ [Ifcmpeq 3,LitPush(Bool False),Goto 2,LitPush(Bool True)]
 | Add ⇒ [IAdd])
compExpr jmb (LAcc vn) = [Load (index jmb vn)]
compExpr jmb (vn::=e) =
  compExpr jmb e @ [Dup, Store (index jmb vn)]
compExpr jmb ( {cn}e..fn ) =
  compExpr jmb e @ [Getfield fn cn]
compExpr jmb (FAss cn e1 fn e2) =
  compExpr jmb e1 @ compExpr jmb e2 @ [Dup_x1, Putfield fn cn]
compExpr jmb (Call cn e1 mn X ps) =
  compExpr jmb e1 @ compExprs jmb ps @ [Invoke cn mn X]

compExprs jmb []     = []
compExprs jmb (e#es) = compExpr jmb e @ compExprs jmb es
```

Fig. 8. Compilation of expressions

```
compStmt  :: java_mb ⇒ stmt ⇒ instr list

compStmt jmb Skip = []
compStmt jmb (Expr e) = (compExpr jmb e) @ [Pop]
compStmt jmb (c1;; c2) = (compStmt jmb c1) @ (compStmt jmb c2)
compStmt jmb (If(e) c1 Else c2) =
      (let cnstf = LitPush (Bool False);
           cnd   = compExpr jmb e;
           thn   = compStmt jmb c1;
           els   = compStmt jmb c2;
           test  = Ifcmpeq (int(size thn +2));
           thnex = Goto (int(size els +1))
       in [cnstf] @ cnd @ [test] @ thn @ [thnex] @ els)
compStmt jmb (While(e) c) =
      (let cnstf = LitPush (Bool False);
           cnd   = compExpr jmb e;
           bdy   = compStmt jmb c;
           test  = Ifcmpeq (int(size bdy +2));
           loop  = Goto (-(int((size bdy) + (size cnd) +2)))
       in [cnstf] @ cnd @ [test] @ bdy @ [loop])
```

Fig. 9. Compilation of statements

Compilation is then gradually extended to the more complex structures presented in Section 2.2, first of all methods. Our compiler first initializes all local variables (*compInitLvars*), then translates the body statement and return expression. Incidentally, we have to refer to the type compilation function *compTpMethod* here already to determine the maximum operand stack size reached by executing the bytecode. This, together with the length of the register array, are the two numbers required by bytecode verification, as indicated in Section 2.3.2. Also note that the exception table component, the last component of a *java_mb mdecl*, is left empty because we do not take exception handling into account here.

```
compMethod :: java_mb prog ⇒ cname ⇒ java_mb mdecl
              ⇒ jvm_method mdecl
compMethod G C jmdl ≡ let (sig, rT, jmb) = jmdl;
                          (pns,lvars,blk,res) = jmb;
                          mt = (compTpMethod G C jmdl);
                          bc = compInitLvars jmb lvars @
                               compStmt jmb blk @ compExpr jmb res @
                               [Return]
                    in (sig, rT, max_ssize mt, size lvars, bc, [])
```

The compilation function *comp* for programs is essentially defined by mapping

compMethod over all methods of all classes.

```
compClass :: java_mb prog => java_mb cdecl=> jvm_method cdecl
compClass G ≡ λ (C,cno,fdls,jmdls).
                (C,cno,fdls, map (compMethod G C) jmdls)

comp :: java_mb prog => jvm_prog
comp G ≡ map (compClass G) G
```

This concludes the definition of the compiler.

4.2 Compiler Correctness

Let us briefly review the compiler correctness statement and its proof – we refer the reader to [Str02a] for a more detailed discussion.

In a rough sketch, the compiler correctness statement takes the form of the traditional "commuting diagram" argument: Suppose execution of a statement c transforms a μJava state s into a state s'. Then, for any μJVM state t corresponding to s, executing the bytecode resulting from a translation of c yields a state t' corresponding to s'.

This sketch has to be refined in that the notion of correspondence has to be made precise, both for expressions and for statements. Besides, compiler correctness depends on a few assumptions that will be spelled out below.

We first need a notion describing the effects of completely evaluating an expression or executing a statement on a μJVM state, in analogy to the evaluation and execution relations on the μJava level. We note the following:

- Apart from the exception indicator and the heap, only the topmost frame is affected, but not the remaining frame stack.
- When executing an instruction sequence *instrs*, the program counter advances by *size instrs*, provided *instrs* is part of the bytecode of a method body (which in particular implies that the start and end positions of the program counter are well-defined).

Of course, these observations do not hold for intermediate steps of a computation, e.g. when frames are pushed on the frame stack during a method call or when jumping back to the start of a while loop, but only after completion, when the frames have been popped off again or the whole while loop has finished.

This suggests a *progression* relation, defined as:

```
{G,C,S} ⊢ {hp0, os0, lvars0} >- instrs → {hp1, os1, lvars1} ≡
  ∀pre post frs.
  (gis (gmb G C S) = pre @ instrs @ post) ⟶
   G ⊢ (None,hp0,(os0,lvars0,C,S,size pre)#frs) -jvm-> 
       (None,hp1,(os1,lvars1,C,S,(size pre) + (size instrs))#frs)
```

Here, `{G,C,S} ⊢ {hp0, os0, lvars0} >- instrs → {hp1, os1, lvars1}` expresses that execution of instructions `instrs` transforms heap `hp0`, operand stack `os0` and local variables `lvars0` into `hp1`, `os1` and `lvars1`. Since exceptions are excluded from consideration here, the exception indicator of the states is invariantly `None`.

The instructions `instrs` are a subsequence of the instructions (selected by `gis`) of the method body (selected by `gmb`) of signature `S` in class `C` of program `G`. During execution, the program counter advances from the first position of `instrs` (at `size pre`) to the position right behind `instrs` (at `size pre + size instrs`). This indirect coding of the program counter movement not only makes the correctness statement more concise. It is also helpful in the proof, as it removes the need for complex "program counter arithmetic" – abstract properties like transitivity of `progression` are sufficient most of the time.

We are now prepared to clarify the notion of correspondence between μJava and μJVM states and present the correctness theorem for evaluation of expressions (the one for execution of statements is analogous).

Suppose that evaluation of expression `ex` in μJava state `(None, hp, loc)` yields result `val` and state `(None, hp', loc')`, and some other conditions explained in a moment are met. We assume that expression `ex` is part of the method which can be identified by program `G`, class `C` and signature `S`. When running the bytecode `compExpr (gmb G C S) ex` generated for `ex` in a μJVM state having the same heap `hp`, an (arbitrary) operand stack `os` and local variables as in `loc`, we obtain heap `hp'`, the operand stack with `val` on top of it and local variables as in `loc'` (the representation of local variables is refined by function `locvars_locals`). Thus, we obtain the following

Theorem 8

```
⟦ G ⊢ (None,hp,loc) -ex ≻ val-> (None,hp',loc');
  wf_java_prog G;
  class_sig_defined G C S;
  wtpd_expr (env_of_jmb G C S) ex;
  (None,hp,loc) ::≼ (env_of_jmb G C S) ⟧ ⟹
  {(TranslComp.comp G), C, S} ⊢
    {hp, os, (locvars_locals G C S loc)}
      >- (compExpr (gmb G C S) ex) →
    {hp', val#os, (locvars_locals G C S loc')}
```

The theorem is displayed diagrammatically below – note the simplification regarding local variables on the bytecode level.

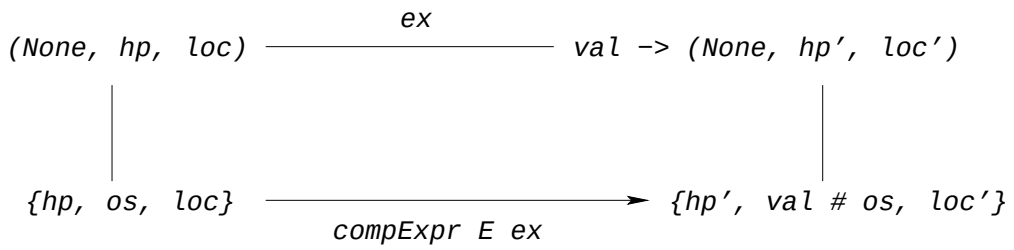


Fig. 10. Compiler Correctness statement

Let us now take a look at the preconditions:

- The source program has to be well-formed as described in Section 2.2.2.
- The class signature has to be defined in the sense that `C` is a valid class in `G` and method lookup with `S` gives a defined result.
- Expression `ex` is well-typed in the environment of the method body. This environment `(env_of_jmb G C S)` is generated by the types of the local variables and the method parameters.
- Finally, the start state of the computation, `(hp, loc)`, conforms to this environment, in the sense of Section 2.2.4.

Most of these conditions are provided in order to maintain a consistent and well-defined program state throughout execution of the source program. We thus avoid having to deal with undefined values, as discussed in Section 2.1.

These requirements are not very restrictive: the well-formedness and well-typing conditions are standard for compilers; the conformance condition is satisfied when a program is started with an empty heap and the local variables are initialized to their default values.

5 Compiling Types

5.1 Motivation

Given the above correctness theorem, the question arises whether semantically correct code could be type-incorrect. Quite abstractly, note that a type system always imposes a constraint on a language, thus marking even "valid" programs as type-incorrect. And indeed, the empirical evidence given in [SS01] shows that there is a mismatch between the Java source and bytecode type systems: Code containing a `try ... finally` statement is accepted by a standard Java typechecker, compiled to semantically equivalent bytecode, but then

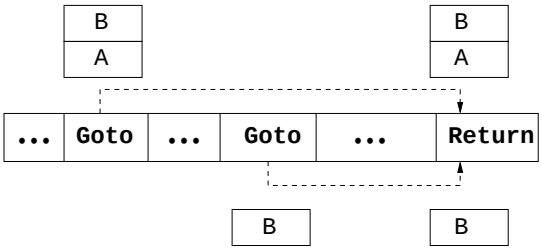


Fig. 11. Semantically unproblematic, but type-incorrect bytecode

rejected by the bytecode verifier. The deeper reason is that source code type-checker and bytecode verifier have different notions of when variables have been "definitely assigned" [BGJS00]. Since our restricted language fragment does not contain the above-mentioned construct, we cannot reproduce this problem.

Still, there are sufficient sources of potential bytecode type errors. For example, different branches leading to a jump target could generate operand stacks of different heights - possibly an innocuous situation if not all of the operand stack is used in the sequel (see Figure 11). However, such code is rejected by the bytecode verifier.

5.2 Definition of Type Compiler

In a first approximation, generation of the type certificate proceeds in analogy to compilation of code with the aid of functions `compTpExpr`, `compTpStmt` etc. that yield a list of state types having the same length as the bytecode produced by `compExpr`, `compStmt` etc. However, it becomes apparent in the proofs that the resulting state type lists are not self-contained and therefore the immediately following state type also has to be taken into account. For example, the position directly behind the code of an `If` statement can be reached via at least two different paths: either by a jump after completion of the *then* branch of the statement, or by regular completion of the *else* branch. When proving type correctness of the resulting code, it has to be shown that both paths lead to compatible state types.

This suggests that, e.g., `compTpExpr` should not have type $\texttt{expr} \Rightarrow \texttt{method\_type}$ but rather $\texttt{expr} \Rightarrow \texttt{state\_type} \Rightarrow \texttt{method\_type} \times \texttt{state\_type}$. The function definitions are shown in Figures 12 and 13. For technical reasons, the function takes two other arguments, a Java program `G`, and a Java method body `jmb`, which essentially is used for computing the variable type, given a variable name (for example in the case of variable access).

Composition of the results of subexpressions is then not simple list concatenation, but rather a particular kind of function composition $\texttt{f1} \ \square \ \texttt{f2}$, defined as

```
compTpExpr :: java_mb ⇒ java_mb prog ⇒ expr ⇒ state_type
              ⇒ method_type × state_type
compTpExprs :: java_mb ⇒ java_mb prog ⇒ expr list
              ⇒ state_type ⇒ method_type × state_type

compTpExpr jmb G (NewC c) = pushST [Class c]
compTpExpr jmb G (Cast c e) =
     (compTpExpr jmb G e) □ (replST 1 (Class c))
compTpExpr jmb G (Lit val) = pushST [the (typeof (λv. None) val)]
compTpExpr jmb G (BinOp bo e1 e2) =
     (compTpExpr jmb G e1) □ (compTpExpr jmb G e2) □
     (case bo of
        Eq  ⇒ popST 2 □ pushST [PrimT Boolean] □
              popST 1 □ pushST [PrimT Boolean]
      | Add ⇒ replST 2 (PrimT Integer))
compTpExpr jmb G (LAcc vn) =
     (λ(ST,LT).  pushST [ok_val (LT ! (index jmb vn))] (ST, LT))
compTpExpr jmb G (vn::=e) =
     (compTpExpr jmb G e) □ dupST □ (popST 1)
compTpExpr jmb G ( {cn}e..fn ) =
     (compTpExpr jmb G e) □ replST 1 (snd (the (field (G,cn) fn)))
compTpExpr jmb G (FAss cn e1 fn e2 ) =
     (compTpExpr jmb G e1) □ (compTpExpr jmb G e2) □
     dup_x1ST □ (popST 2)
compTpExpr jmb G ({C}a..mn({fpTs}ps)) =
  (compTpExpr jmb G a) □ (compTpExprs jmb G ps) □
  (replST ((size ps) + 1) (rT_of (the (method (G,C) (mn,fpTs)))))

compTpExprs jmb G [] = comb_nil
compTpExprs jmb G (e#es) =
     (compTpExpr jmb G e) □ (compTpExprs jmb G es)
```

Fig. 12. Compilation of expression types

```
compTpStmt :: java_mb ⇒ java_mb prog ⇒ stmt ⇒ state_type
              ⇒ method_type × state_type

compTpStmt jmb G Skip = comb_nil
compTpStmt jmb G (Expr e) =  (compTpExpr jmb G e) □ popST 1
compTpStmt jmb G (c1;; c2) =
   (compTpStmt jmb G c1) □ (compTpStmt jmb G c2)
compTpStmt jmb G (If(e) c1 Else c2) =
   (pushST [PrimT Boolean]) □ (compTpExpr jmb G e) □ popST 2 □
   (compTpStmt jmb G c1) □ nochangeST □ (compTpStmt jmb G c2)
compTpStmt jmb G (While(e) c) =
   (pushST [PrimT Boolean]) □ (compTpExpr jmb G e) □ popST 2 □
   (compTpStmt jmb G c) □ nochangeST
```

Fig. 13. Compilation of statement types

```
λx0. let (xs1, x1) = (f1 x0); (xs2, x2) = (f2 x1) in (xs1 @ xs2, x2).
```

A few elementary functions describe the effect on a state type or components thereof. For example, *pushST* pushes types *tps* on the operand type stack, and *replST n tp* replaces the topmost *n* elements by *tp*, whereas *storeST* stores the topmost stack type in the local variable type array:

```
pushST       :: ty list ⇒ state_type ⇒ method_type × state_type
pushST tps   ≡ λ(ST, LT). ([Some (ST, LT)], (tps @ ST, LT))

replST n tp  ≡ λ(ST, LT). ([Some (ST, LT)], (tp # (drop n ST), LT))

storeST i tp ≡ λ(ST,LT). ([Some (ST,LT)], (tl ST, LT [i:=OK tp]))

nochangeST sttp ≡ ([Some sttp], sttp)
dupST        ≡ λ(ST, LT). ([Some (ST, LT)], (hd ST # ST, LT))

dup_x1ST     ≡ λ(ST, LT). ([Some (ST, LT)],
                  (hd ST # hd (tl ST) # hd ST # (tl (tl ST)), LT))
popST n      ≡ λ(ST, LT). ([Some (ST, LT)], (drop n ST, LT))
```

In order to make the inner workings of these definitions more transparent, let us take a look at how the type compiler would translate an expression *1+2*, more precisely *BinOp Add (Lit 1) (Lit 2)*.

Note that the bytecode emitted by *compExpr* is the instruction sequence

```
[LitPush 1, LitPush 2, IAdd]
```

Translation of *(Lit 1)* with *compTpExpr* yields a function which, given a state type *(ST, LT)*, produces the method type *[(ST, LT)]* plus the state type *(PrimT Integer # ST, LT)*, which indicates that *LitPush 1* has the effect of leaving behind an integer on the operand stack. Translation of *(Lit 2)* yields a function which transforms a state type, such as the one just obtained, by pushing another *PrimT Integer* on the operand type stack. The □ operator concatenates the resulting method type lists and returns the state type *(PrimT Integer # PrimT Integer # ST, LT)*. Finally, the *IAdd* instruction pops the topmost two elements from the operand type stack and leaves behind a method type and state type as depicted in Figure 14.

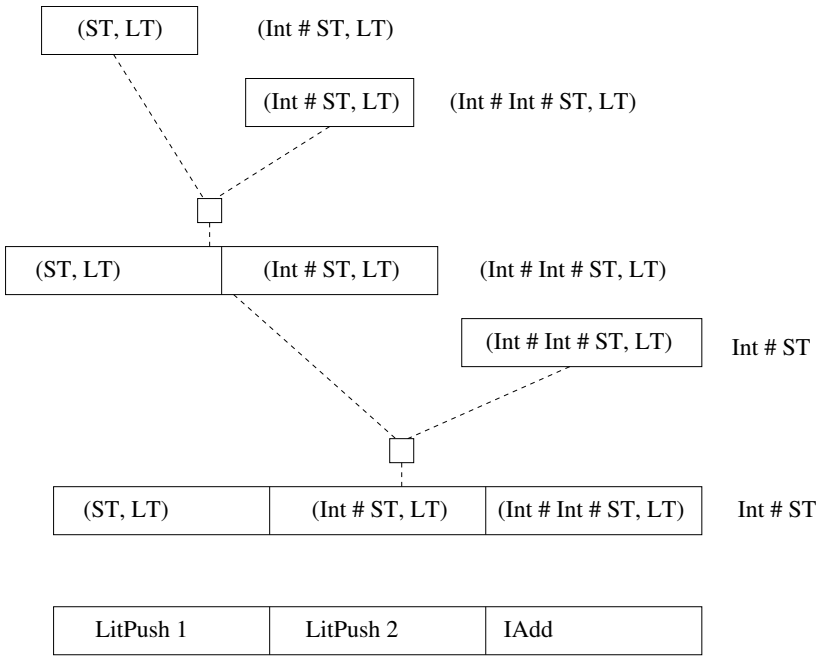


Fig. 14. Example of type compilation

Given the above functions, the function generating the bytecode type of a method can be defined:

```
start_ST :: opstack_type
start_ST ≡ []

start_LT :: cname ⇒ ty list ⇒ nat ⇒ ty err list
start_LT C pTs n ≡ (OK (Class C))#(map OK pTs)@(replicate n Err)

compTpMethod  :: [java_mb prog, cname, java_mb mdecl] ⇒ method_type
compTpMethod G C ≡ λ((mn,pTs),rT, jmb).
                   let (pns,lvars,blk,res) = jmb
                   in (mt_of
                       ((compTpInitLvars jmb lvars □
                         compTpStmt jmb G blk □
                         compTpExpr jmb G res □
                         nochangeST)
                        (start_ST, start_LT C pTs (size lvars))))
```

Starting with a state type that consists of an empty operand type stack and a local variable type array that contains the current class `C` (corresponding to the `this` pointer), the parameter types `pTs` and types of uninitialized local variables, we first initialize the variable types (`compTpInitLvars`), then compute the type of the method body and the return expression. The final `Return` instruction does not change the state type, which accounts for `nochangeST`. These computations yield a pair `method_type` × `state_type`, from which we extract the desired method type (`mt_of`).

Finally, *compTp* raises compilation of bytecode types to the level of programs, in analogy to *comp*:

```
compTp :: java_mb prog ⇒ prog_type
compTp G C sig ≡ let (D, rT, jmb) = (the (method (G, C) sig))
                 in compTpMethod G C (sig, rT, jmb)
```

Is there any difference between computed method types and method types a bytecode verifier would infer? Possibly yes: Our procedure yields a method type which is a fixpoint wrt. the type propagation carried out by a bytecode verifier, but not necessarily the least one. As an example, take the bytecode a compiler would produce for the method

```
void foo (B b) { A a; a = b; return; }
```

with `B` a subtype of `A`. We would assign the type `A` to the bytecode variable representing `a`, but a bytecode verifier would infer the more specific type `B`, because in any computation, variable `a` holds at most values of type `B`.

6 Well-Typedness: Theorem and Proof

We can now state our main result:

Theorem 9 *The code generated by* `comp` *is well-typed with respect to the bytecode type generated by* `compTp`*, provided the program* `G` *to be compiled is well-formed:*

$$\texttt{wf\_java\_prog G} \Longrightarrow \texttt{wt\_jvm\_prog (comp G) (compTp G)}$$

Let us first give a sketch of the proof before going into details: In a first step, we essentially unfold definitions until we have reduced the problem to verifying well-typedness of individual methods, i.e. to showing that the predicate *wt_method* holds for the results of *compMethod* and *compTpMethod*. For this, we need to show that the start condition *wt_start* is satisfied for the state type *(start_ST, start_LT C pTs n)*, which is straightforward, and then prove that *wt_instr* holds for all instructions of the bytecode.

The functions constructing bytecode and bytecode types have a very similar structure, which we exploit to demonstrate that a relation *bc_mt_corresp* between bytecode and method types is satisfied and which gives us the desired result about *wt_instr*. In particular, *bc_mt_corresp* is compatible with the operators `@` and □, so that correspondence of *compMethod* and *compTpMethod* is decomposed into correspondence of *compExpr* and *compTpExpr* resp. *compStmt* and *compTpStmt*. The key lemmas establishing this correspondence are proved

by induction on expressions resp. statements and constitute the major part of the proof burden.

We will now look at some details, beginning with the definition of predicate *bc_mt_corresp*, which states that bytecode *bc* and state type transformer *f* correspond in the sense that when *f* is applied to an initial state type *sttp0*, it returns a method type *mt* and a follow-up state type *sttp* such that each instruction in *bc* up to an index *idx* is well-typed.

```
bc_mt_corresp :: [bytecode, state_type ⇒ method_type × state_type,
                  state_type, jvm_prog, ty, p_count] ⇒ bool
bc_mt_corresp bc f sttp0 cG rT idx ≡
let (mt, sttp) = f sttp0 in
  size bc = size mt ∧
  (∀mxs pc.
    mxs = max_ssize (mt@[Some sttp]) ⟶
    pc < idx ⟶
    wt_instr (bc!pc) cG rT (mt@[Some sttp]) mxs (size mt+1) [] pc)
```

As mentioned in Section 5, when checking for *wt_instr*, we also have to peek at the position directly behind *mt*, so we have to use the state type list *mt@[Some sttp]* instead of just *mt*. The definition of *bc_mt_corresp* is further complicated by the fact that *wt_instr* depends on the maximum operand stack size, which we keep track of by computing *max_ssize*.

bc_mt_corresp is compatible with *@* and □, provided that the results of the state type transformers *f1* and *f2* are seamlessly fitted together (expressed by *start_sttp_resp*).

Lemma 10 *Decomposition of* bc_mt_corresp_comb*:*

```
⟦ bc_mt_corresp bc1 f1 sttp0 cG rT (size bc1);
  bc_mt_corresp bc2 f2 (sttp_of (f1 sttp0)) cG rT (size bc2);
  start_sttp_resp f2 ⟧
⟹ bc_mt_corresp (bc1@bc2) (f1□f2) sttp0 cG rT (size (bc1@bc2))
```

At first glance, this lemma looks abstract, i.e. does not seem to refer to particular instructions. A closer analysis reveals that this is not so: In the proof of the lemma, we have to show that well-typed code can be "relocated" without losing its type-correctness. For example, adding bytecode *bc_post* resp. bytecode types *mt_post* to the end, as in the following lemma, does not impair well-typing of an instruction at position *pc*:

Lemma 11

```
⟦ wt_instr (bc ! pc) cG rT mt mxs max_pc et pc;
  bc' = bc @ bc_post; mt' = mt @ mt_post;
```

```
  mxs ≤ mxs'; max_pc ≤ max_pc';
  pc < size bc; pc < size mt;  max_pc = (size mt)⟧
⟹ wt_instr (bc' ! pc) cG rT mt' mxs' max_pc' et pc
```

The proof of this lemma requires, among others, monotonicity of the *app* predicate of Section 3.3.2 with respect to the maximum stack size *mxs* - intuitively because executing more instructions might lead to an increase in the maximum stack size:

```
⟦ app i G mxs rT pc s; mxs ≤ mxs' ⟧ ⟹ app i G mxs' rT pc s
```

This is shown by case distinction over the instruction *i* and so indirectly requires properties that depend on a particular instruction set.

Let us now turn to the cornerstone of our proof, the correspondence between bytecode and bytecode types for expressions and statements. To provide an intuition for the argument, let us contrast type inference, as carried out by a bytecode verifier, with our *a priori* computation of a method type. During type inference, a bytecode verifier has to compare the state types that result from taking different data paths in the bytecode, such as when jumping to the instruction following a conditional from the *then* and *else* branch. If these state types differ, an attempt is made to merge them, by computing the least common supertype. If merging fails because there is no such supertype, the bytecode is not typeable. Otherwise, type inference continues with the updated state type.

Why is the bytecode type we compute with *compTpExpr* and *compTpStmt* stable in the sense that no such updates are necessary? Recall that our compiler initializes all local variables at the beginning of a method. It is now possible to determine the most general type a bytecode variable can assume: it is the type the variable has in the source language. Any assignment of a more general type on the bytecode level would indicate a type error on the source code level.

The predicate *is_inited_LT* expresses that the local variable array has been initialized with the appropriate types:

```
is_inited_LT :: [cname, ty list, (vname × ty) list, ty err list]
                ⇒ bool
is_inited_LT C pTs lvars LT ≡
   (LT = (OK (Class C))#(map OK pTs)@(map (OK ∘ var_type) lvars))
```

We can now enounce the lemma establishing the correspondence between *compStmt* and *compTpStmt* – the one for expressions is similar:

Lemma 12

```
⟦ wf_prog wf_java_mdecl G; jmb = (pns,lvars,blk,res);
```

```
  E = (local_env G C (mn, pTs) pns lvars); E ⊢ s√;
  is_inited_LT C pTs lvars LT;
  bc' = (compStmt jmb s);  f' = (compTpStmt jmb G s) ⟧
⟹  bc_mt_corresp bc' f' (ST, LT) (comp G) rT (size bc')
```

Note the two most important preconditions: the statement `s` under consideration has to be well-typed (`E ⊢ s√`) and the local variable array `LT` has to be initialized properly.

The proof of this lemma is by induction on statements. Apart from decomposition (Lemma 10), it makes use of lemmas which further clarify the effect of the state type transformers. The lemma for expressions reads, in abridged form:

```
⟦  E ⊢ ex :: T; is_inited_LT C pTs lvars LT ⟧
 ⟹  sttp_of (compTpExpr jmb G ex (ST, LT)) = (T # ST, LT))
```

It states that the bytecode computing the value of an expression `ex` leaves behind its type `T` on the operand type stack `ST` and does not modify the local variable type array `LT`, provided the latter is appropriately initialized. Thus, it can be understood as an abstraction of the compiler correctness statement of Section 4.2.

7 Conclusions

7.1 Related Work

In this paper, we have defined a type certifying compiler and shown the type correctness of the code it generates. Even though the definitions are given in the proof assistant Isabelle, we can convert them to executable ML code using Isabelle's extraction facility [BN00,BS03].

Our encoding of the Java source language owes much to the formalization on paper in [DE99], which has also been the basis for an alternative formalization, including a type soundness proof, in the Declare system [Sym99]. Both differ from our definition in that they use a small-step operational semantics. Each approach has particular merits: A small-step semantics is suitable for modelling non-terminating computations and concurrency; type soundness can be defined in terms of a defensive machine, as for the JVM in Section 2.3.3. However, it is often clumsy to handle: For stating a compiler correctness theorem, we would have needed a complex bisimulation relation between source and target states.

Barthe *et al.* [BDJ+01,BDJ+02] employ the Coq system for proofs about the JVM and bytecode verification. They formalize the full JavaCard bytecode language, but do not have a compiler.

In [PV98], Posegga and Vogt look at bytecode verification from a model checking perspective. They transform a given bytecode program into a finite state machine and check type safety, which they phrase in terms of temporal logic, by using an off-the-shelf model checker. Basin, Friedrich, and Gawkowski [BFG02] use Isabelle/HOL, μJava, and the abstract BCV framework [Nip01] to prove the model checking approach correct.

Working towards a verified implementation in SPECWARE, Qian, Goldberg and Coglio have specified and analyzed large portions of the bytecode verifier [CGQ98,CGQ00]. Goldberg [Gol98] rephrases and generalizes the overly concrete description of the BCV given in the JVM specification [LY99] as an instance of a generic data flow framework. Qian [Qia99] specifies the BCV as a set of typing rules, a subset of which was proved correct formally by Pusch [Pus99]. Qian [Qia00] also proves the correctness of an algorithm for turning his type checking rules into a data flow analyzer.

Stata and Abadi [SA98] were the first to specify a type system for a subset of Java bytecode. They focused on the problem of bytecode subroutines. The typing rules they use are clearer and more precise than the JVM specification, but they accept fewer safe programs.

Freund and Mitchell [FM98,FM99,Fre00] develop typing rules for increasingly large subsets of the JVM, including exception handling, object initialization, and subroutines. They do not look at compilation.

Leroy [Ler01,Ler03] gives a very good overview on bytecode verification, and proposes a polyvariant data flow analysis in the BCV to solve the notorious subroutine problem. Coglio [Cog01,Cog02] provides an even simpler analysis for handling subroutines in the BCV. The most recent version of our BCV [Kle03] uses this scheme as basis for the formalization of subroutines in μJava.

Compiler correctness proofs have for a long time been an active research area, starting with pencil-and-paper proofs for a simple expression language [MP67], and more recently using diverse specification formalisms such as Z [Ste98] and verification systems such as ACL2 [You89,Goe00], HOL [Cur93] and PVS [DV01]. Little attention is given to preservation of type correctness, which is not surprizing since the source language (such as Lisp) or the target language only have a weak type system. The Verifix project [GZ99] has attempted to develop an appropriate compiler correctness criterion for finite resources (such as memory) and nondeterministic programs. Neither of these is a problem in our case. In particular, our source and target language are abstracted over the

same memory model.

In recent years, compilation with types has been the subject of intense study, which however has mostly ignored the aspect of general compiler correctness and instead focused on the preservation of certain safety properties. The source languages are mostly functional, having ML-like [Mor95,SA95] type systems or even stronger ones such as System F [MWCG99]. The purpose is to exploit types for a program analysis that allows for more efficient closure conversion [WDMT97] or that avoids boxing polymorphic variables. In [LST02], Java is compiled not to bytecode, but to a functional intermediate language which can also be used as the target of functional programming languages [LST03].

Since compilation is a multi-stage process involving several intermediate languages, well-typing of programs has to be preserved during compilation. The type correctness statement is mostly proved on paper. Critical questions such as naming of bound variables and α-convertibility are often glossed over, even though there is good evidence that proofs of typing properties of lambda calculi become quite demanding once these details are taken into account [NN99].

The extensive pencil-and-paper formalization of Java using Abstract State Machines in [SSB01] is complementary to ours: whereas the ASM formalization is much more complete with respect to language features, the proofs are less detailed, and some of the underlying proof principles are unclear, such as, for example, extending inductive proofs "modularly" to deal with new language constructs.

7.2 Extensions

We are not proponents of the idea of necessarily carrying proofs to ultimate perfection, but believe that once a fully formal basis has been laid, it can be extended with moderate effort and provides a convenient experimental platform for new language features.

The dataflow framework described in Section 3 is sufficiently general to encompass, among others, exception handling and object initialization. Even Java's tricky jump-subroutine mechanism can be incorporated, by changing the notion of state types to *sets* of stack and local variable types. This is described in detail in [Kle03].

When trying to extend the compiler to deal with exceptions, we have to face the problem that the target semantics ceases to precisely simulate the source code semantics, in the following sense: Evaluating a source code expression terminates as soon as the expression has been fully processed, no matter whether an exception results or not. However, the JVM keeps running after an excep-

tion has occurred and pops stack frames even beyond the frame in which the current computation has been started (all this is hidden in the definition of `find_handler` in Section 2.3.2). A remedy is to define a more fine-grained execution function than `exec` with which source and target machines can be synchronized.

Also, the non-local transfer of control caused by exceptions makes the type compiler more difficult. Recall that type compilation for an expression yields a function of type $\texttt{state\_type} \Rightarrow \texttt{method\_type} \times \texttt{state\_type}$, where the second component delivered by the function is the `state_type` resulting from executing the corresponding code. If control can be transferred to several destinations, several different state types can result. Accordingly, the composition function $\Box$ will become more complex. Given this, we expect the type compiler to be stated as naturally as in Section 5.2 even for a language including exceptions.

There are several other limitations of our source language whose consequences for the type compiler have not yet been fully explored. For example, our operational semantics of the source language initializes all variables at the beginning of a method body. In standard Java, no such initialization is necessary, but a "Definite Assignment" check ensures that variables are assigned to before being used. When adapting our source code type system, we could include contextual information about initialization status in the type compiler. In a similar vein, we could deal with block structure with local variable declarations, by replacing the fixed parameter `jmb` in the type compiler by a context-dependent type assignment.

Acknowledgements

We are grateful to Tobias Nipkow, Norbert Schirmer and Martin Wildmoser for discussions about this work.

References

[BDJ+01] G. Barthe, G. Dufay, L. Jakubiec, S. Melo de Sousa, and B. Serpette. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer Verlag, 2001.

[BDJ+02] G. Barthe, G. Dufay, L. Jakubiec, S. Melo de Sousa, and B. Serpette. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294

of *Lecture Notes in Computer Science*, pages 32–45. Springer Verlag, 2002.

[BFG02] David Basin, Stefan Friedrich, and Marek Gawkowski. Verified bytecode model checkers. In *Theorem Proving in Higher Order Logics (TPHOLs'02)*, volume 2410 of *Lecture Notes in Computer Science*, pages 47–66, Virginia, USA, August 2002. Springer-Verlag.

[BGJS00] Gilad Bracha, James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, second edition, June 2000.

[BN00] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Proc. TYPES Working Group Annual Meeting 2000*, LNCS, 2000.

[BS03] Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. In *Proc. 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2003)*, Electronic Notes in Theoretical Computer Science, 2003.

[CGQ98] 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.

[CGQ00] 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.

[Cog01] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. Technical Report, Kestrel Institute, December 2001.

[Cog02] 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.

[Cur93] Paul Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993.

[DE99] Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer Verlag, 1999.

[DV01] A. Dold and V. Vialard. A mechanically verified compiling specification for a Lisp compiler. In *Proc. FSTTCS 2001*, December 2001.

[FM98] 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.

[FM99] 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.

[Fre00] Stephen N. Freund. *Type Systems for Object-Oriented Intermediate Languages*. PhD thesis, Stanford University, 2000.

[Goe00] W. Goerigk. Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof. In *Proc. of the ACL2'2000 Workshop*, Austin, Texas, U.S.A., October 2000.

[Gol98] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*, 1998.

[GZ99] G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 201–230, 1999.

[Kle03] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[KN01] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, 2001. Invited contribution to special issue on Formal Techniques for Java.

[KN02] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. to appear.

[Ler01] 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.

[Ler03] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 2003. To appear.

[LST02] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):112–152, March 2002.

[LST03] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving Java compiler. In *Proc. Int'l. Conf. on Compiler Construction*, April 2003.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, April 1999.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, CMU, December 1995.

[MP67] John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, RI, 1967.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In M. Miculan F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[NOP00] 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.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.

[Ohe01] 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/`.

[Pus99] 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.

[PV98] Joachim Posegga and Harald Vogt. Java bytecode verification using model checking. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

[Qia99] 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.

[Qia00] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.

[Ros02] Eva Rose. *Vérification de Code d'Octet de la Machine Virtuelle Java. Formalisation et Implantation.* PhD thesis, Université Paris VII, 2002.

[RR98] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, CA, 1995.

[SA98] 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.

[SS01] R. F. Stärk and J. Schmid. The problem of bytecode verification in current implementations of the JVM. Technical report, Department of Computer Science, ETH Zürich, Switzerland, 2001.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer Verlag, 2001.

[Ste98] Susan Stepney. Incremental development of a high integrity compiler: experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, November 1998.

[Str02a] Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

[Str02b] Martin Strecker. Investigating type-certifying compilation with Isabelle. In *Proc. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[Sym99] Donald Syme. *Declarative Theorem Proving for Operational Semantics.* PhD thesis, University of Cambridge Computer Laboratory, 1999.

[WDMT97] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn A. Turbak. A typed intermediate language for flow-directed compilation. In *TAPSOFT*, pages 757–771, 1997.

[You89] William D. Young. A mechanically verified code generator. Technical Report 37, Computational Logic Inc., January 1989.