Formal Verification of a Java Compiler in Isabelle [⋆]

Martin Strecker

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

Abstract. This paper reports on the formal proof of correctness of a compiler from a substantial subset of Java source language to Java bytecode in the proof environment Isabelle. This work is based on extensive previous formalizations of Java, which comprise all relevant features of object-orientation. We place particular emphasis on describing the effects of design decisions in these formalizations on the compiler correctness proof.

1 Introduction

The compiler correctness proof presented in this paper is part of a more comprehensive research effort aiming at formalizing and verifying key aspects of a substantial subset of the Java programming language, in particular:

- the type system and operational semantics of Java, with a proof of type soundness [Ohe01a]
- an axiomatic semantics, with a proof of its equivalence to the operational semantics [Ohe01b]
- an abstract dataflow analyzer [Nip01], instantiated for Java bytecode, with a proof of correctness of Java bytecode verification [KN02].

All these formalizations and proofs have been carried out in the Isabelle system. They will be briefly reviewed in Section 2, as far as relevant for our purpose.

The present work links the existing Java source and bytecode formalizations,

- by defining an executable compiler from source to bytecode (Section 3), and
- by stating and proving a compiler correctness theorem with Isabelle (Section 4).

To date, there have been numerous machine-supported compiler correctness proofs and several pen-and-paper formalizations of aspects of Java (cf. Section 5). The current effort distinguishes itself from previous ones by being the first (to the best of our knowledge) to formally establish such a result for a realistic object-oriented language: our description of Java, even though incomplete in several respects, comprises all essential features of object-orientation. In addition, the

[⋆] This research is funded by the EU project *VerifiCard* source language model includes a notion of exceptions (which are, however, so far not taken into account in the correctness proof). To achieve a succinct presentation of the operational semantics, a few fundamental design decisions have been made. A recurring theme of this paper will be to analyse their consequences for compiler verification.

This paper can only give a survey of the overall effort – for details, consult the Isabelle sources at `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 "micro" edition of Java (see [NOP00] for a gentle introduction) accommodates essential aspects of Java, like classes, subtyping, object creation, inheritance, dynamic binding and exceptions, but abstracts away from the wealth of arithmetic data types, interfaces, arrays, access modifiers, and multi-threading. It is a good approximation of the JavaCard dialect of Java, targeted at smart cards.

2.1 Isabelle Preliminaries

Isabelle is a generic framework for encoding different object logics. In this paper, we will only be concerned with the incarnation Isabelle/HOL [NPW02], which comprises a higher-order logic and facilities for defining datatypes 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* at position *n*.

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 ⟦ *P1; ...; Pn* ⟧ $\Longrightarrow$ *C* to express that *C* follows from the premises *P1 ... Pn*. 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.

2.2 μJava Source Language

We will now sketch the formalization of the μJava source language in Isabelle in the following stages:

- We will describe the *structure* of a μJava program, building on a formalization of its constituents, i.e. raw terms and types. Obviously, this is prerequisite to defining a translation from Java to Java bytecode (Section 3).
- Using a notion of *well-typedness*, it is possible to single out "legal" expressions, statements and (in extenso) programs. Only these will be considered for compiler correctness, see Section 4.1.
- The behaviour of μJava programs is defined by an *operational semantics* in the form of an evaluation relation. The semantics is essential for the statement of compiler correctness, and it determines the structure of the proof (Section 4.2), which is by induction on the evaluation relation.
- The correctness proof requires still another proviso, namely that during execution, program states *conform* to expected types. As it turns out, this precondition is purely technical and does not impose a genuine restriction. It is satisfied for well-typed programs anyway, due to the *type-safety* of μJava.

The Structure of Programs The μJava 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 and quotes.

```
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` permits to create a new instance, given a class name `cname`; `Cast` performs a type cast; `Lit` embeds values `val` (see below) into expressions. μJava only knows a few binary operations `binop`: test for equality and integer addition. There is access to 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. This concludes the description of μJava terms.

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.

Let us briefly sketch the μJava type level, even though its deep structure is not modified by the compiler (which is reflected by some of the preservation lemmas of Section 4.2).

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

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.

On this basis, it is possible to define what is a field declaration `fdecl` and a method signature `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 Java 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
```

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`.

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.

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.

```
types   aheap  = loc ⇀ obj
        locals = vname ⇀ val
        state  = aheap × locals
        xstate = xcpt option × state
```

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`.

```
Call: ⟦ ...    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 [Ohe01a] 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 – see Section 4.2.

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`$\vdash$`v::`$\preceq$`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::`$\preceq$ `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

For the Isabelle formalization of the Java Virtual Machine, μJVM, we have in principle to go through the same steps as for μJava, in particular definition of the language structure and operational semantics. There are however quite different mechanisms for dealing with typing issues; they are only skimmed in the following.

The μJava bytecode instructions manipulate data of type `val`, as introduced in Section 2.2. The instruction set is a simplification of the original Java bytecode in that the `Load` and `Store` instructions are polymorphic, i.e. operate on any type of value. In μJVM, there are so far only system exceptions; exceptions cannot be thrown explicitly and cannot be handled.[1]

```
datatype
  instr = Load nat              | Store nat
        | LitPush val           | New cname
        | Getfield vname cname  | Putfield vname cname
        | Checkcast cname       | Invoke cname mname (ty list)
        | Return                | Pop
        | Dup                   | Dup_x1
        | Dup_x2                | Swap
        | IAdd                  | Goto int
        | Ifcmpeq int
```

As mentioned in Section 2.2, much of the program structure is shared between source and bytecode level. Simply by exchanging the method body type, we can define the type of Java virtual machine programs:

```
types   bytecode = instr list
        jvm_prog = (nat × nat × bytecode) prog
```

[1] This situation is currently being remedied.

Apart from the bytecode, the method body contains two numbers (maximum stack size and length of local variable array) which are required by the bytecode verifier but need not concern us here.

The type `jvm_prog` reflects the structure of a Java class file rather directly up to minor differences, such as version numbers, redundant administrative information (e.g. methods count), and data related to interfaces, which are not handled in μJava and can thus be assumed to be void.

Ensuring type correctness of bytecode is the responsibility of the bytecode verifier. In analogy to the type safety result for the source level, it can be shown that if bytecode passes a correct bytecode verifier, it can be executed "safely" – see [KN02] for details.

The JVM operational semantics defines the effect of executing instructions on the `jvm_state`, which is a triple consisting of an optional component indicating the presence of an exception, a heap and a frame stack.

```
types   opstack    = val list
        locvars    = val list
        frame      = opstack ×  locvars × cname ×  sig × nat
        jvm_state  = xcpt option × aheap × frame list
```

Each frame holds an operand stack `opstack`, a list of local variables `locvars`, the class name and signature identifying the currently executing method, and the program counter. `xcpt`, `aheap` and `sig` are the same as on the source level. The only genuine data refinement is for the representation of local variables: In μJava, the method-local variables `locals` are a mapping from names to values. In μJVM, `locvars` is a list $this, p_1, \ldots, p_n, l_1, \ldots, l_m$ containing a reference $this$ to the current class and the parameters $p_1, \ldots, p_n$ and local variable values $l_1, \ldots, l_m$ of the current method. This refinement is achieved by function `locvars_locals`, still needed further below.

The function `exec_instr` takes an instruction and the constituents of a state and computes the follow-up state. The `Load` instruction, for example, produces no exception (`None`), leaves the heap `hp` unmodified, and changes the topmost frame by pushing the contents of the local variable with index `idx` on the operand stack `stk` and incrementing the program counter `pc`.

```
exec_instr (Load idx) G hp stk vars C S pc frs =
     (None, hp, ((vars ! idx) # stk, vars, C, S, pc+1)#frs)
```

Function `exec` carries out a single step of computation: It looks up the current method, given μJVM program `G`, class name `C` and signature `S`, selects the instruction indicated by the program counter and executes it.

The relation `G ⊢ s -jvm-> t`, defined by means of the transitive closure of `exec`, expresses that state `t` can be reached from state `s` by a sequence of successful execution steps:

```
exec_all :: [jvm_prog,jvm_state,jvm_state] => bool  ("_ ⊢ _ -jvm-> _")
G ⊢ s -jvm-> t == (s,t) ∈ {(s,t). exec(G,s) = Some t}^*
```

3 Compiler Definition

Compilation is straightforwardly defined with the aid of a few directly executable functions. To begin with, `mkExpr :: java_mb => expr => instr list` and `mkStat :: java_mb => stmt => instr list`, defined in Figures 1 and 2, translate expressions resp. statements. The function `index` computes the index of variable name `vn` in method body `jmb` by looking up its position in a list of the form $this, p_1, \ldots, p_n, l_1, \ldots, l_m$.

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

Fig. 1. Compilation of expressions

On this basis, compilation is extended to more complex structures, first method bodies, then classes and finally entire programs. `mkMethod` translates method bodies, essentially by appending code for the method body block `blk` and the result expression `res` and adding a `Return` instruction:

```
mkMethod :: java_mb => (nat * nat * bytecode)
mkMethod jmb == let (pn,lv,blk,res) = jmb
                in (0, 0,
                    (concat (map (mkInit jmb) lv)) @
                    (mkStat jmb blk) @ (mkExpr jmb res) @ [Return])
```

Prepended to this are instructions initializing the local variables `lv` to their default values – a complication which could be avoided if variables were known to be assigned to before being read. Such a check, as embodied in Java's "definite assignment" principle, is however not part of our current well-formedness condition of μJava programs. As mentioned in Section 2.3, the first two components of the result of `mkMethod` are only relevant for bytecode verification. Indeed, it can be shown that the compiler only produces bytecode which is type correct in the sense that it passes bytecode verification [Str02]. Since there is no space to discuss this issue here, we set these components to zero.

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

Fig. 2. Compilation of statements

Classes are translated by generating code for the method bodies and leaving the remaining structure untouched – recall that μJava and μJVM classes essentially differ in their method bodies.

```
mkClass :: java_mb cdecl  =>  (nat * nat * bytecode) cdecl
mkClass == λ(cn,cno,fdl,jmdl). (cn,cno,fdl,
           map (λ(s,t,mb). (s,t, mkMethod mb)) jmdl)

comp :: java_mb prog  =>  jvm_prog
comp jp == map mkClass jp
```

As mentioned in Section 2.3, the structure of `jvm_prog` is essentially the Java class file format. Even though the compiler only produces a symbolic class file and not an executable binary, this last step is relatively straightforward: It is possible to generate ML code from the Isabelle definition, using the code extraction facility described in [BN00], and then supply the print functions in ML.

4 Compiler Verification

4.1 Compiler Correctness Statement

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 here and further motivated in Section 4.2.

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 `length 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 relation `progression`, defined as:

```
progression :: jvm_prog ⇒ cname ⇒ sig ⇒
               aheap ⇒ opstack ⇒ locvars ⇒
               bytecode ⇒
               aheap ⇒ opstack ⇒ locvars ⇒
               bool
               ("{_,_,_} ⊢ {_, _, _} >- _ → {_, _, _}" )
{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,length pre)#frs) -jvm->
  (None,hp1,(os1,lvars1,C,S,(length pre) + (length 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 `length pre`) to the position right behind `instrs` (at `length pre + length 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 engaging in 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 `ex` is part of the method which can be identified by program `G`, class `C` and signature `S`. When running the bytecode `mkExpr (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'` (recall from Section 2.3 that the representation of local variables is refined by function `locvars_locals`).

```
theorem compiler_correctness_eval:
  ⟦ 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;
  (hp,loc) ::≼ (env_of_jmb G C S) ⟧ ⟹
  {(comp G), C, S} ⊢
    {hp, os, (locvars_locals G C S loc)}
      >- (mkExpr (gmb G C S) ex) →
    {hp', val#os, (locvars_locals G C S loc')}
```

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

$$\begin{tikzcd}
\texttt{(None,hp,loc) -ex} \arrow[r] \arrow[d, dash] & \texttt{val -> (None,hp',loc')} \arrow[d, dash] \\
\texttt{hp, os, loc >-} \arrow[r, "\texttt{mkExpr E ex}"] & \texttt{hp', val\#os, loc'}
\end{tikzcd}$$

Let us now take a look at the preconditions:

- The source program has to be well-formed as described in Section 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:
  ```
  class_sig_defined G C S ==
  is_class G C ∧ (∃ m. (method (G, C) S = Some m )
  ```
- Expression `ex` is well-typed in the environment of the method body. This environment `(env_of_jmb G C S)` is essentially generated by the types of the local variables and the method parameters.
- Finally, the start state of the computation, `(hp, loc)`, conforms (Section 2.2) to this environment.

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.

4.2 Compiler Correctness Proof

The correctness proof is by mutual induction on the evaluation relation `G ⊢ _ -_ ≻_->_` resp. execution relation `_ ⊢ _ -_-> _`. Apart from the rules that pass on exceptions, which are dealt with trivially under our assumptions, we essentially obtain a case distinction on the constructs of the source language. These are handled uniformly and, except for pathological cases such as method call, without difficulty:

- First, we establish preconditions so as to be able to use the induction hypotheses for subcomputations.
- After that, we apply the induction hypotheses, mostly exploiting transitivity of the relation `progression`, and then symbolically evaluate the bytecode with Isabelle's simplifier.

The reasoning for obtaining preconditions is as follows:

- `class_sig_defined` is obvious for most cases, when remaining within the same method body. An exception is the case "method call", where the preservation lemmas mentioned below are applied.
- Establishing `wtpd_expr` mostly requires showing that it holds for subexpressions (such as `wtpd_expr E e1`) when it is known to hold for a compound expression (such as `wtpd_expr E (BinOp op e1 e2)`), which is achieved by inversion of the typing rules. Again, method call is more intricate.
- Showing that conformance is still satisfied in the state reached after performing a number of subcomputations (e.g. after evaluating `G ⊢ s0 -e1≻ v1-> s1` and before evaluating `G ⊢ s1 -e2≻ v2-> s2`) requires repeated application of the type-soundness theorem.

Even though the proof is fairly straightforward, it has a few rough edges, some of which can be directly traced back to object-oriented concepts such as subclassing and dynamic method lookup: In the method `Call` rule of the operational semantics, we use a lookup function `method` which gives an indefinite result under certain conditions (cf. Section 2.2), for example when being applied to a class that is not defined in the current program `G`. It is the purpose of the preconditions of the correctness theorem, in particular the conformance requirement, to exclude this situation.

The same `method` function is also used by the `Invoke` bytecode instruction in the translated program `comp G`. To make sure that definedness of method lookup in the source program carries over to the bytecode program, we have established a series of *preservation lemmas* which incidentally formalize the claim that compilation leaves most of the structure of programs unmodified (cf. Section 3). The preservation lemmas for the direct subclass relation *subcls1* and for method lookup are:

```
lemma comp_subcls1: subcls1 G = subcls1 (comp G)

lemma comp_method : ⟦ wf_prog wf_mb G; is_class G C⟧ ⟹
  (method (G, C) S) = Some (D, rT, mb) ⟶
  (method (comp G, C) S) = Some (D, rT, mkMethod mb)
```

Method lookup for a program, a class *C* and signature *S* returns the defining class *D*, the return type *rT* and the source method body *mb* resp. the translated method body *mkMethod mb*.

In view of the above remarks, it may be surprising that the preconditions of the correctness theorem are not exclusively motivated by object-oriented features, but are rather a consequence of the particular style of semantics definition and resulting minor differences between μJava and μJVM semantics. They would – in a similar form – also be required for a much simpler language presented in the same fashion.

We illustrate this point with the translation of conditionals. Our limited μJVM instruction set forces us to translate the *If* statement to the *Ifcmpeq* operator, which compares the result of evaluating the condition with the constant *Bool False*. If evaluation of the condition did not leave behind a boolean value on top of the stack (which we know it does), *Ifcmpeq* would not be perturbed by the type-inconsistency, but would deterministically select the "else" branch. This is an example of an "offensive" JVM behaviour, close to an actual implementation, that does not bother to care for situations that cannot happen. In contrast, the behaviour of μJava is not determined in this case, so the source and bytecode level behave differently unless we assume that type inconsistencies cannot arise.

5 Conclusions

After a review of the existing Isabelle/HOL formalizations of Java, this paper has described the formalization and correctness proof of another key component, a compiler from source to bytecode. Because the compiler had to fit into an existing framework, the definitions of source and target language could not be "tuned" so as to suit the needs of compiler verification. Under these circumstances, the overall effort invested (4-5 months of work for a novice Isabelle user) can be considered moderate. This seems to indicate that

– proof assistant technology has progressed enough to allow for an analysis of realistic, complex languages.
– the existing formalizations are sufficiently mature to serve as a basis for further investigations.

On a technical level, this work has given insight into the interaction of language formalization and compiler correctness proofs:

– The big-step semantics leads to a concise, intuitive correctness theorem because only states at the end of a computation are compared. In contrast, a small-step (structural operational) semantics, such as the ASM formalization in [SSB01], requires juxtaposition of intermediate states, leading to a complex equivalence relation involving the contents of entire frame stacks.
– However, it is a (general) drawback of a big-step semantics that it only permits to talk about terminating computations and cannot express concurrency and related concepts.
– Object-orientation made the reasoning slightly more involved than it would have been for a plain imperative language, but had no decisive influence.
– A few places of the μJava operational semantics have a non-constructive flavour due to indefinite values resulting from functions like `the`. These make the evaluation rules more elegant, but buy us nothing in the compiler correctness proof – undefined situations have to be excluded a priori by preconditions of the theorem. Thus, we are confident that our proof could be easily recast in a constructive logic.
– Apart from that, the formalization uses few Isabelle specifics, as witnessed by the definitions and theorems presented in this paper. A transfer to other proof environments offering notions such as a higher-order logic, primitive recursion and inductive definitions should be possible without great effort.

There is a long history of mechanized compiler verification, conducted with different provers and for diverse language paradigms (imperative, functional) [MW72,Cur93,Bou95]. Gradually, the field is evolving away from a demonstration of feasibility to an analysis of complex artifacts: A "stack" of system components, ranging from a high-level language down to a microprocessor, has been examined in the ACL2 system [MBHY89]. Here, the emphasis is on a verified chain of refinements; the techniques employed in the individual phases [You89] are not substantially different from ours. Another direction, pursued in the Verifix project [DV01], is to refine the compiler program itself from an abstract "compiling specification" down to an executable version. Our work has still another focus: it aims at an in-depth investigation of aspects of the Java language.

Traditionally, languages have been studied semi-formally, with proofs in the form of pen-and-paper arguments. A very comprehensive account of this kind is given in [SSB01] for Java. This description covers far more language constructs and Java-specific concepts (multi-threading; class loading) than ours. The consequences of a purely technical difference, namely the use of the ASM formalism, akin to a small-step semantics, have already been discussed. Even though ASMs have previously been used in a fully formal verification [Sch99], it may be difficult to cope with the sheer amount of detail in Java.

Future work on our μJava compiler will add missing features, notably exceptions, however without trying to be complete in a literal sense. Also, it may be worth while to look at some compiler optimizations, for example mapping different source variables having different life times to a single bytecode variable. However, many optimizations that can sensibly be performed on the bytecode are already applicable on the source level.

Acknowledgements

The compiler presented in Section 3 has originally been written by Cornelia Pusch and Giampaolo Bella and only been slightly modified for the correctness proof. Johannes Pfeifroth has given a first version of the compiler correctness statement of Section 4.1. I am grateful to Axel Dold, Gerwin Klein, Marko Luther and Norbert Schirmer for comments on a draft version of this paper.

References

[BN00] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Proc. TYPES Working Group Annual Meeting 2000*, LNCS, 2000. Available from `http://www4.in.tum.de/~berghofe/papers/TYPES2000.pdf`.

[Bou95] Samuel Boutin. Preuve de correction de la compilation de Mini-ML en code CAM dans le système d'aide à la démonstration COQ. Technical Report 2536, INRIA Rocquencourt, April 1995.

[Cur93] Paul Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993. Available from http://www.cl.cam.ac.uk/Research/HVG/vista/.

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

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

[MBHY89] J.S. Moore, W.R. Bevier, W. A. Hunt, and W. D. Young. System verification. *Special issue of J. of Automated Reasoning*, 5(4), 1989.

[MW72] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–70, 1972.

[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*, 2001.

[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, 2002.

[Ohe01a] 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/`.

[Ohe01b] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience*, 13(13), 2001.

[Sch99] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen.* PhD thesis, Universität Ulm, 1999.

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

[Str02] Martin Strecker. Compilation and bytecode verification in μJava. Forthcomming, preprint available from `http://www4.in.tum.de/~streckem/Publications/compbcv02.html`, 2002.

[You89] William D. Young. A mechanically verified code generator. *J. of Automated Reasoning*, 5(4):493–518, 1989.