Investigating Type-Certifying Compilation with Isabelle *

Martin Strecker

Fakultät für Informatik, Technische Universität München
http://www4.in.tum.de/~streckem

Abstract. This paper 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.

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). We consider it 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 a more precise statement of theorems and proofs than can be achieved with pencil-and-paper formalizations (see Section 6 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. 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]); 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, permits to gain insight into type systems of programming languages and how they are related. Incompatibilities discovered in the source

* This research is funded by the EU project *VerifiCard*

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 [KN02] and the correctness (in the sense of preservation of semantics) of a compiler [Str02]. Seen in this context, this paper is interesting because it is another piece in the puzzle.

In the following, we will first summarize the most important concepts of our Java and JVM formalization (Section 2), in particular source and bytecode type systems. We define the code compiler *comp* in Section 3, the type compiler *compTp* in Section 4. The type correctness statement for generated code and a detailed discussion of the proof follow in Section 5. Section 6 concludes with a discussion of related 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 that permits to encode 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 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* at position *n*.

We have the usual type constructors *T1* $\times$ *T2* for product and *T1* $\Rightarrow$ *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* $\rightsquigarrow$ *T2*, which just abbreviates *T1* $\Rightarrow$ *(T2 option)*.

The constructor *Some* has a left inverse, the function *the :: 'a option* $\Rightarrow$ *'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 Java Source Language

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 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 Java 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.

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.

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

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.

2.3 Java Bytecode

The Isabelle formalization of the Java Virtual Machine, μJVM, follows the same lines as the formalization of μJava. Here, we will concentrate on a description of the instruction set and the bytecode type system.

Instructions 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. As mentioned above, we do not consider exceptions so far, even though we plan to do so in the future.

```
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
```

Apart from the bytecode, the method body contains two numbers (maximum stack size and size of local variable array) which are required by the bytecode verifier and whose role will be elucidated in Section 3.

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.

Execution Similarly as for the source code level, details of the JVM operational semantics need not concern us here. However, as described below, the bytecode verifier carries out a computation abstractly, not on values, but on types. For a better understanding, it is therefore instructive to take a glance at the concrete semantics first.

The semantics is defined by describing the effect of 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* indicates an exception, the heap *aheap* is a mapping

from locations to objects, and *sig* is the same as on the source level. The local variable array *locvars* is a list $this, p_1, \ldots, p_n, l_1, \ldots, l_m$ containing a reference *this* to the current class, the method parameters $p_1, \ldots, p_n$ and local variable values $l_1, \ldots, l_m$ of the current method.

Typing We will now sketch the type system of the bytecode level, which we have borrowed from [KN02]. The discussion tries to convey a general idea, but is necessarily incomplete.

Every Java runtime environment comes equipped with a bytecode verifier which ensures, by means of static analysis, that the bytecode fulfills certain criteria, to be described further below. Given some bytecode, the bytecode verifier essentially performs a combination of type inference, computing a type for the bytecode, and type checking, verifying that the computed type satisfies the criteria. In the process, the bytecode verifier builds up a *bytecode type* of the code it examines; since bytecode verification works on a per-method basis, we will also refer to this type as *method type*.

As mentioned before, the analysis performed by the verifier is mostly an abstract computation on types instead of values. The effect of executing an instruction on the heap is not made explicit, only changes of the operand stack and the local variable array are taken into account by recording, for each instruction of the bytecode, the types found on the operand stack (*opstack_type*) and the variable array (*locvars_type*) before executing the instruction. Thus, a method type is essentially a list of *state types*, one for each instruction. The following summarizes the type structure we use; consult [KN02] for details on the *err* and *option* wrappers:

```
types
  opstack_type = ty list
  locvars_type = ty err list
  state_type = opstack_type ×locvars_type
  method_type = state_type option list
  prog_type = cname ⇒sig ⇒method_type
```

It is the purpose of our type compiler (function *compTpMethod* in Section 4) to produce a method type for each source code method, thus supplanting the type inference aspect of bytecode verification. We will now turn to the definition of the predicate *wt_method* which embodies the type checking aspect and expresses well-typing of bytecode with respect to a method type. In Section 5, it is shown that this predicate is indeed satisfied for generated bytecode resp. method types.

The definition of *wt_method* and related predicates is given in Figure 1. Let *G* be a program, *C* a class name, *pTs* the parameter types of the method under consideration, *rT* its return type, *mxs* the maximum stack size reached during execution, *mxl* the maximum number of local variables, *ins* the instruction list and *phi* the method type (the parameter *et* is an exception table that is not taken into account by us). *wt_method* requires the instruction list to be non-empty, a start condition *wt_start* to be satisfied and all instructions in the instruction list to be well-typed. The start condition essentially expresses correct

```
app' (Store idx, G, pc, maxs, rT, (ts#ST, LT)) = (idx < length LT)
app' (IAdd, G, pc, maxs, rT, (PrimT Integer#PrimT Integer#ST,LT)) = True

wt_instr :: [instr,jvm_prog,ty,method_type,nat,p_count,
             exception_table,p_count] ⇒bool
wt_instr i G rT phi mxs max_pc et pc ==
  app i G mxs rT pc et (phi!pc) ∧
  (∀(pc',s') ∈set (eff i G pc et (phi!pc)).
    pc' < max_pc ∧G ⊢s' <=' phi!pc')

wt_start :: [jvm_prog,cname,ty list,nat,method_type] ⇒bool
wt_start G C pTs mxl phi ==
  G ⊢Some ([],(OK (Class C))#((map OK pTs))@(replicate mxl Err))
      <=' phi!0

wt_method :: [jvm_prog,cname,ty list,ty,nat,nat,instr list,
              exception_table,method_type] ⇒bool
wt_method G C pTs rT mxs mxl ins et phi ==
  let max_pc = length ins in
  0 < max_pc ∧wt_start G C pTs mxl phi ∧
  (∀pc. pc<max_pc ⟶wt_instr (ins ! pc) G rT phi mxs max_pc et pc)

wt_jvm_prog :: [jvm_prog,prog_type]⇒bool
wt_jvm_prog G phi ==
  wf_prog (λG C (sig,rT,(maxs,maxl,b,et)).
           wt_method G C (snd sig) rT maxs maxl b et (phi C sig)) G
```

Fig. 1. Well-typedness of Bytecode

initialization of the first element of method type `phi`. An instruction `i` at position `pc` is well-typed (predicate `wt_instr`) for `phi` if it is applicable (`app`) at `pc` and all follow-up positions `pc'` of `i` are within the bounds of the instruction list and all follow-up state types `s'` resulting from symbolically executing `i` are subtypes of `phi ! pc'`.

The function `app` is defined by case distinction on instructions, with the aid of `app'`. We can only show a representative subset: For example, the `Store` instruction is only applicable if there is an element on top of the operand stack, and the store index is within the bounds of the local variable array. Integer addition `IAdd` requires both operands to be of integer type. Still other instructions test that the maximum operand stack size `mxs` is not exceeded when pushing new arguments on the stack.

Function `eff` computes the set of effects of an instruction on a state type. For example, the branch-on-equal instruction `Ifcmpeq` makes the program counter advance to the next instruction or to the branch target and in both cases pops the two topmost elements from the operand type stack and leaves the local variable type array unchanged. Thus, `eff (Ifcmpeq b) G pc et (Some (T1 # T2`

`# ST, LT))` yields [1] the result `[(pc + 1, Some (ST, LT)), (pc + b, Some (ST, LT))]`.

It can be shown that the bytecode type system described so far is compatible with the operational semantics of μJVM– see the type soundness theorem in [KN02].

3 Compiling Code

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 2 resp. Figure 3 for comparison with the type compilation functions defined in Section 4.

```
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. 2. Compilation of expressions

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 local variable array, are the two numbers required by bytecode verification, as indicated during the definition of `jvm_prog` (see Section 2.3).

[1] in slightly beautified form

```
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(length thn +2));
        thnex = Goto (int(length 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(length bdy +2));
        loop = Goto (-(int((length bdy) + (length cnd) +2)))
   in [cnstf] @ cnd @ [test] @ bdy @ [loop])
```

Fig. 3. Compilation of statements

```
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, length lvars, bc)
```

The compilation function *comp* for programs is essentially defined by mapping *compMethod* over all methods of all classes. We refer the reader to [Str02] to further details concerning the compiler.

This reference also contains a proof that the compiler is correct in the sense that it is semantics-preserving. Therefore, 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, which does however not show up in the restricted language fragment we consider.

4 Compiling Types

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.

All this suggests that, for example, *compTpExpr* should not have type *expr* ⇒ *method_type* but rather type *expr* ⇒ *state_type* ⇒ *method_type* × *state_type*. (For technical reasons, the function takes two other arguments, a Java method body *jmb* and a Java program *G*). The function definitions are shown in Figures 4 and 5.

Composition of the results of subexpressions is then not simple list concatenation, but rather a particular kind of function composition *f1* □ *f2*, defined as *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*, defined as

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

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:

```
storeST :: [nat, ty, state_type] ⇒method_type × state_type
storeST i tp == (λ(ST, LT). ([Some (ST, LT)], (tl ST, LT [i:= OK tp])))
```

Given these 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 ⇒locvars_type
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 (length lvars))))
```

```
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 ((length ps) + 1) (method_rT (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. 4. 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. 5. Compilation of statement types

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)
```

5 Well-Typedness: Theorem and Proof

We can now state our main result: 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:

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

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 ...)*, 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.

```
constdefs
  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 (length bc = length mt ∧
    (∀ mxs pc.
    mxs = max_ssize (mt@[Some sttp]) ⟶
    pc < idx ⟶
    wt_instr (bc ! pc) cG rT (mt@[Some sttp]) mxs (length mt + 1) [] pc))
```

As mentioned in Section 4, 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.

```
lemma bc_mt_corresp_comb:
  ⟦bc_mt_corresp bc1 f1 sttp0 cG rT (length bc1);
  bc_mt_corresp bc2 f2 (sttp_of (f1 sttp0)) cG rT (length bc2);
  start_sttp_resp f2 ⟧
  ⟹bc_mt_corresp (bc1 @ bc2) (f1 □f2) sttp0 cG rT (length (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 by adding bytecode resp. bytecode types to the front or to the end, as in the following lemma:

```
lemma wt_instr_prefix:
  ⟦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 < length bc; pc < length mt; max_pc = (length mt)⟧
  ⟹wt_instr (bc' ! pc) cG rT mt' mxs' max_pc' et pc
```

The proof of this lemma 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, locvars_type] ⇒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 wt_method_compTpStmt_corresp:
  ⟦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 (length bc')
```

Note the two most important preconditions: the statement *s* under consideration has to be well-typed (*E* $\vdash s\surd$) and the local variable array *LT* has to be initialized properly.

The proof of this lemma is by induction on statements. Apart from the decomposition lemma *bc_mt_corresp_comb*, 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.

To summarize, we have shown that the method types computed by *compTp* are valid types for the bytecode generated by *comp*. 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`. The type `A` would be assigned to the bytecode variable representing `a` by us, but a bytecode verifier would infer the less general type `B`, because in any computation, `a` holds at most values of type `B`.

6 Conclusions

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].

Compilation taking into account types has been tackled for some time [Mor95], mostly with an emphasis on functional languages. The extensive pencil-and-paper formalization of Java using Abstract State Machines [SSB01] is complementary to ours: Whereas the ASM formalization is much more complete with

respect to language features, the proofs are often sketchy. Even if they take another route to showing type correctness of generated bytecode, not relying on a separate type certificate, there should be some analogy of argument. However, to take an example, it is not quite clear which kind of induction (structural on expressions / length of bytecode?) is actually performed for showing type correctness, essential ingredients akin to our lemma `wt_instr_prefix` are missing.

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. We hope to do so with the current formalization, by taking exception handling and related concepts into account.

Acknowledgements

The major part of the definitions of the well-typedness predicates of Section 2.3 is due to Gerwin Klein. I am grateful to him and to Tobias Nipkow and Norbert Schirmer for discussions about this work.

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`.

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.

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

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.

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/`.

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

Str02. Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. CADE*, volume 2392 of *LNCS*, pages 63–77. Springer Verlag, 2002.