Extracting a formally verified, fully executable compiler from a proof assistant

Stefan Berghofer [1] Martin Strecker [2,3]

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

Abstract

Compilers that have been formally verified in theorem provers are often not directly usable because the formalization language is not a general-purpose programming language or the formalization contains non-executable constructs. This paper takes a comprehensive, even though simplified model of Java, formalized in the Isabelle proof assistant, as starting point and shows how core functions in the translation process (type checking and compilation) are defined and proved correct. From these, Isabelle's program extraction facility generates ML code that can be directly interfaced with other, possibly "unsafe" code.

Key words: Java, JVM, Compiler, Theorem Proving, Code Extraction

1 Introduction

Due to the growing performance of proof assistants, it is becoming possible to model increasingly complex programming languages, defining compilers for them and rigorously proving them correct. In many cases, though, the resulting development cannot directly be put to work, for the following reasons:

- Functions are defined in a language tailored for the specific proof assistant, which is usually not a general-purpose programming language. In order to be able, for example, to translate programs with the compiler implementation that has been proved correct, the compiler has to be reimplemented manually in a programming language, which is tedious and error-prone.
- Definitions are made with the purpose of facilitating proofs and are therefore often given in relational (instead of functional) form or contain non-executable constructs or highly inefficient functions.

[1] Email: Stefan.Berghofer@in.tum.de

[2] Email: Martin.Strecker@in.tum.de

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

These issues are addressed by a code extraction facility which has recently been added to the proof assistant Isabelle:

- Function definitions in Isabelle can be extracted to the ML programming language. The mechanism is sufficiently general to deal with inductively defined relations as well, which are evaluated in a Prolog-style manner.
- The extractor permits to replace non-constructive or inefficient functions by provably equivalent executable ones.

In this paper, we illustrate the procedure for the translation process from a subset of Java source language to Java bytecode (see Section 2). We show how core functions can be derived from a comprehensive, even though simplified, formalization of Java, presented in Section 3.2. In particular, we look at

- a type checking and -inference function for which a type soundness result can be shown (Section 3.3)
- a compilation function which provably preserves semantics and which in addition can be shown to produce type correct bytecode (Section 3.4).

We will then discuss the general principles of program extraction and describe some of the generated functions (Section 4). The work presented here is still in progress: Even though the approach has turned out to be viable in general, both theoretic and practical questions need to be resolved; these are discussed throughout the text and summarized in Section 5.

Due to space limitations, we can only sketch our formalization and the resulting compiler. The full Isabelle development is available from the web page http://isabelle.in.tum.de/verificard/.

2 Processing Steps

The translation from Java source language to Java bytecode proceeds in several stages, as depicted in Figure 1, and uses both verified and unverified functions. Here, a function is called *verified* if it has been defined in Isabelle and certain correctnesss properties have been shown to hold.

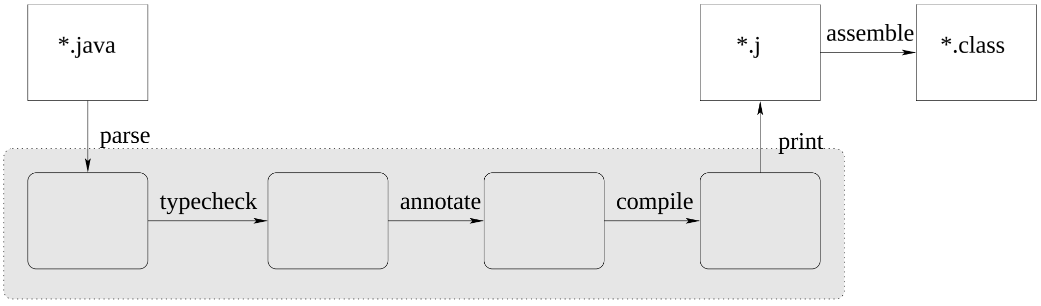

Fig. 1. Processing steps during compilation

The single steps are as follows:

(i) A Java source file is *parsed*, using a parser generated from a downsized Java grammar by means of a parser generator that is contained in the Standard ML of New Jersey toolkit [SML].

(ii) Parsing yields an internal format, called a *pre-program* (we similarly talk of pre-classes and pre-terms – see the description in Section 3.2). This pre-program is then checked for structural well-formedness and well-typing of pre-terms (Section 3.3).

(iii) If it passes these tests, the pre-program can be transformed to a genuine program, which contains some additional annotations that are further used during compilation, but also facilitates description of typing rules and the operational semantics.

(iv) The program is then compiled into bytecode, which is still a symbolic format and not binary code.

(v) For obtaining binary class files, we first print the symbolic bytcode and then convert it using the Jasmin assembler [Mey].

The verified functions, shown in the shaded area of Figure 1, are ML functions that have been extracted from Isabelle definitions. The mechanisms underlying program extraction will be further explained in Section 4. Note that the extraction facility itself has not been proved "correct", so even the verified functions cannot be claimed to be error-free with ultimate certainty. This problem and possible solutions will be expounded in Section 5.

3 Isabelle Formalization

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.

3.1 An Isabelle Primer

Let us first introduce a few elementary concepts of Isabelle that will be used in the following.

Isabelle is a generic framework that permits to encode 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*.

3.2 Java Language Definitions

3.2.1 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.

As mentioned in Section 2, the structures we obtain from parsing Java source files are pre-programs, -terms and the like, which are then converted to genuine programs, terms etc.. In our presentation, we concentrate on the genuine terms and then make the correspondence to pre-terms precise.

In our term structure, we follow 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). Note that currently only a few binary operations are supported – adding more operations increases volume, but not logical complexity.

```
datatype expr
  = NewC cname               -- class instance creation
  | Cast cname expr          -- type cast
  | Lit val                  -- literal value, also references
  | BinOp binop expr expr    -- binary operation
  | LAcc vname               -- local (incl. parameter) access
  | LAss vname expr          (_::=_ )      -- local assign
  | FAcc cname expr vname    ({_}_.._ ) -- field access
  | FAss cname expr vname expr    ({_}_.._:=_ ) -- field ass.
  | Call cname expr mname
    (ty list) (expr list)         ({_}_.._( {_}_) ) -- method call

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

For some constructs, more readable mixfix syntax is defined, enclosed in brackets. As compared to Java input syntax, our expressions contain type annotations, enclosed in braces. They indicate the name (`cname`) of the defining class for field access, field assignment and method call, and the types (`ty list`) of method parameters. Their role will become apparent when looking at the typing rules in Section 3.3.1.

Pre-terms have the same structure as terms – in particular, we have the same distinction between expressions `pre_expr` and statements `pre_stmt`, with the only difference that type annotations are omitted:

```
datatype pre_expr  = ... |   pCall pre_expr mname (pre_expr list)  (_.._(_) )
```

and similarly for field access and assignment. Thus, a method call that is written `{C}a..mn({[T1,T2]}[a1,a2])` with annotations becomes `a..mn([a1,a2])` without.

It is now easy to define a function `erase_tp_e :: expr ⇒ pre_expr` which erases type information from expressions. This function is the inverse of type inference, in a sense made more precise in Section 3.3.1.

Terms are among the elementary building blocks for creating more complex structures; in addition, we need to define types `ty` and name spaces, like the one for classes (`cname`), variables (`vname`) and the like. For reasons of space, we can only refer the reader to [NOP00] for a more detailed discussion.

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). Here again, we have restricted ourselves to a simplified Java, excluding, among others, interfaces and arrays. A more comprehensive Isabelle formalization [Sch03] shows how they can be added.

```
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 in different ways: on the Java source level for terms and for pre-terms, and then again 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
```

Similarly, `java_pre_mb` replaces `stmt` and `expr` by `pre_stmt` and `pre_expr`.

3.3 Type checking and Well-Formedness

3.3.1 Typing

For terms, we have typing judgements making precise what the type of a term is, under a given environment. These judgements essentially serve as *type checking* rules. For pre-terms, we additionally want to infer type annotations, provided the pre-term is well-typed, so the typing judgements are combined type checking and *type inference* rules.

Let's look at typing for terms first. The typing judgements are defined as inductive relations and 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. For expression list es and type list Ts, we have the auxiliary notation E ⊢ es [::] Ts with the obvious meaning.
- E ⊢ s √ means that statement s is well-typed in environment E.

The *environment* E used here is 'c env, a pair consisting of a program 'c prog and a local environment lenv, mapping variable names to types. The corresponding selectors are prg and localT.

The most interesting rule describes type correctness for method calls:

```
Call: ⟦ E⊢a::Class C;   E⊢ps[::]pTs;
          max_spec (prg E) C (mn, pTs) = {((md,rT),pTs')} ⟧ ⟹
        E⊢{C}a..mn({pTs'}ps)::rT
```

A method call expression is well-typed, provided:

- object a has type Class C,
- the parameters ps have types pTs, and
- method lookup with parameter types pTs for class C yields a single most specific applicable method, which is defined in Class md (which may be more general than Class C), has return type rT and parameter types pTs' (again possibly more general than pTs).

Apart from elucidating the role of the type annotations, the rule is noteworthy because it is defined using a non-constructive function max_spec. That this function is not directly executable is, among others, apparent from the fact that it yields a result of type set. We will have to say a word more about this function in Section 3.3.2.

Meanwhile, we turn to the typing rules for pre-terms. Our typing judgements are now predicates having one more argument than the corresponding judgement for terms:

- E ⊢ pe ⇝ e :: T, which has to be read as: "In environment E, pre-term pe can be turned into term e of type T".
- E ⊢ ps ⇝ s √, with an analogous meaning for statements.

With this, the rule for method call becomes:

```
pCall: ⟦ E⊢pa ⇝ a ::Class C;  E⊢pps ⇝ ps [::]pTs;
```

```
        (max_spec_exec (prg E) C (mn, pTs)) = [((md,rT),pTs')] ⟧ ⟹
     E⊢pa..mn(pps) ⤳ {C}a..mn({pTs'}ps) :: rT
```

which makes it appear plausible that, in general, annotated terms `e` and types `T` can be computed from an environment `E` and a pre-term `pe`. The mode analysis presented in Section 4 confirms that this is indeed the case. In passing, note that we have replaced the non-executable function `max_spec` by an executable variant `max_spec_exec`, which operates on lists instead of sets and whose definition is constructive.

What is the correspondence between the two typing relations? Correctness states that annotated terms inferred by the pre-term typing relation are also well-typed for the term typing relation, and that the original pre-term can be recovered by erasing type information:

```
lemma ty_pre_correct: wf_prog_struct (prg E) ⟶
      E⊢ pe ⤳ e :: T ⟶  E⊢ e :: T ∧ pe = erase_tp_e e
```

Completeness expresses that any well-typed term is the result of type inference for its type erasure:

```
lemma ty_pre_complete: wf_prog_struct (prg E) ⟶
      E⊢ e :: T ⟶  E⊢ (erase_tp_e e) ⤳ e :: T
```

Both lemmas are proved by a rather straightforward rule induction. Both lemmas hold under the proviso that the program component of environment `E` is structurally well-formed – a condition that we will turn to now.

3.3.2 Well-Formedness conditions

For some correctness properties to hold, we have to ensure that several well-formedness conditions are satisfied. We have already encountered the predicate `wf_prog_struct` above. Compiler correctness requires even stronger assumptions, as will be seen in Section 3.4.2. These well-formedness or well-typing conditions are ordered in the sense that verifying some of them presupposes that others have already been checked. Taking care of verifying these conditions in the correct order is usually left to the common sense of the compiler writer – in our case, it is enforced by preconditions in Isabelle rules and theorems.

We now look at the most important conditions:

```
 wf_prog_struct :: 'c prog => bool
  wf_prog_struct G ==
  wf_syscls G ∧ unique G ∧ (∀c∈set G. wf_cdecl_struct G c)
```

`wf_prog_struct` expresses that a program is structurally well-formed, which means that all system classes exist (`wf_syscls`), all classes in `G` have a unique name and all classes of program `G` are well-formed (`wf_cdecl_struct`). The latter implies, in particular, that the class hierarchy is acyclic.

The mentioned predicates are lengthy, but not difficult and can be converted to executable functions by the Isabelle extraction facility. Note that in general, quantification over sets, as in ∀x∈ `S. P x`, is not executable, but here, we deal with quantification over a finite domain: `set` is the conversion from lists to sets, and so `∀x∈ set G. P x` is extracted to the ML expression `forall P G`.

The precondition required for compiler correctness is

```
wf_prog wf_mb G ==
   wf_prog_struct G ∧ (∀c∈ set G. wf_mrT G c ∧ wf_cdecl_mdecl wf_mb G c)
```

Apart from structural well-formedness, it requires method return types to be well-formed and the method declarations (and in consequence also the method body) to be well-typed. The predicate `wf_mrT` can, with minor effort, be made executable. `wf_cdecl_mdecl` is not very interesting, it checks the method head (signature and return types have to be valid types) and then delegates most of the work to its parameter `wf_mb`:

```
wf_mdecl wf_mb G C == λ(sig,rT,mb). wf_mhead G sig rT ∧ wf_mb G C (sig,rT,mb)
```

Recall that method declarations are parametric in the type of the method body, so the method declaration is parameterized by a predicate `wf_mb` of type `'c prog ⇒ cname ⇒ 'c mdecl ⇒ bool`. For method bodies of type `java_pre_mb`, we employ the test predicate `wf_java_pre_mdecl`, which we do not show in entirety. One of its subconditions is that the body statement `pblk` and the return expressions `pres` can be turned into a well-typed annotated term:

```
wf_java_pre_mdecl :: 'c prog ⇒ cname ⇒ java_pre_mb mdecl ⇒ bool
 wf_java_pre_mdecl G C == λ((mn,pTs),rT,(pns,lvars,pblk,pres)).
  ... ∧
  (let E = (G,map_of lvars(pns[↦]pTs)(This↦Class C)) in
     (∃ blk. E⊢ pblk ⇝ blk√) ∧ (∃ res T. E⊢ pres ⇝ res::T ∧ G⊢T≼rT))
```

The existential quantifiers look intimidating. However, one of the functions generated by Isabelle's code extraction enumerates the set `{blk. E⊢ pblk ⇝ blk}`, and so there is an effective means of checking whether this set is empty or not. A similar remark holds for the second conjunct, even though the matter is more complex there. Converting the above definition of `wf_java_pre_mdecl` to an executable one still requires some human intervention, in the sense that auxiliary predicates have to be defined for the existentially quantified conditions. We are currently in the process of extending Isabelle to handle these cases automatically.

After having defined diverse well-formedness predicates, we are in a better position to understand the problem caused by function `max_spec` of Section 3.2.1, and we can envisage a solution. The function is defined as

```
max_spec G C sig ==
     {m. m ∈appl_methds G C sig ∧
        (∀m'∈appl_methds G C sig.  more_spec G m' m --> m' = m)}
```

One source of trouble is the set comprehension based on sets `appl_methds G C sig` which are not obviously finite (even though in fact they are). The greatest problem is the definition of `appl_methds`, which yields the set of applicable methods for class `C` and signature `sig` in program `G`:

```
appl_methds G C == λ(mn, pTs).
     {((Class md,rT),pTs') |md rT mb pTs'.
                     method (G,C)  (mn, pTs') = Some (md,rT,mb) ∧
                     list_all2 (λT T'. G⊢T≼T') pTs pTs'}
```

The function `list_all2` tests that corresponding elements of two lists satisfy a binary predicate, and the set comprehension expression stands for `{z. ∃ md rT mb pTs'. z = ((Class md,rT),pTs') ∧ ...`. This particular definition has turned out to be useful for verification purposes, but it is a catastrophe in terms of executability: We want to collect unknown values, such as `pTs'`, which are in a non-functional relation `list_all2 (λT T'. G⊢T≼T')` with given values, such as `pTs`. In principle, we could enumerate all values `pTs'`, since their number is finite, but this hopelessly inefficient. Instead, we have devised another solution: we define an executable version

```
appl_methds_exec :: 'c prog ⇒ cname ⇒ sig ⇒ ((ty × ty) × ty list) list
  appl_methds_exec G C == λ(mn, pTs).
     map (λ ((mn', pTs'), md, rT, mb). ((Class md,rT),pTs'))
     [((mn', pTs'), md, rT, mb) ∈ method_list (G, C).
             mn = mn' ∧ list_all2 (λT T'. G⊢T≼T') pTs pTs']
```

and base our definition of `max_spec` on it. Are the two functions equivalent? Yes, under some preconditions:

```
lemma appl_methds_appl_methds_exec: ⟦ wf_prog_struct G; is_class G C ⟧ ⟹
  appl_methds G C (mn, pTs) = set (appl_methds_exec G C (mn, pTs))
```

The proofs of lemmas `ty_pre_correct` and `ty_pre_complete` only succeed because we know that these preconditions are satisfied in the context of these proofs.

We have sketched at some length the type checking portion of Figure 1. With the terminology established so far, we naturally obtain functions for the term annotation stage. For lack of space, we do not spell them out here.

3.3.3 Pains and Rewards

Why did we go through the pains of formalizing the above-mentioned features of Java, sometimes even in a non-executable fashion? Our original aim is, in the first place, to analyse properties of the Java language with a proof assistant. Therefore, the formalizations strive to make proofs as easy as possible, which is sometimes in conflict with efficient executability.

The reward of our effort is, among others, a type safety result, which expresses that any statically well-typed Java programs does not produce type violations at runtime. In order to obtain this result, other properties have to be formalized (see [Ohe01] for a detailed exposition), among them

- an operational semantics, describing how the program state changes when evaluating expressions and statements,
- a conformance relation, providing an invariant between dynamic and static

types.

The operational semantics, formalized as an inductive relation, could be subjected to the same treatment as the typing relations above, so as to yield an executable interpreter for the Java source language. Of course, we can do better by compiling Java to bytecode and executing it, which we will turn to now.

3.4 Compiler

3.4.1 Compiler Definition

The compiler takes expressions of the source language, as defined in Section 3.2.1, and produces bytecode, where bytecode is a list of instructions.

The instruction set is a reduced version of real Java bytecode, in that many arithmetic operations are not present, and it is simplified, in the sense that the `Load` and `Store` operations are polymorphic, thus obviating the distinction made in Java between *aload*, *iload* and the like. As an aside, let us mention that we do not really incur a loss of information here, as we compute bytecode types along with bytecode instructions [Str02b], and thus can retrieve the type of each instruction at any moment.

Compilation is now defined with the aid of a few primitive recursive functions. Expressions resp. statements are compiled by `compExpr` and `compStmt`. Apart from the expression resp. statement to be compiled, these 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.

In the following, we give a few representative clauses defining the translation. For the complete set, see [Str02a] or consult the Isabelle sources:

```
compExpr  :: java_mb ⇒ expr ⇒ instr list
compExprs :: java_mb ⇒ expr list ⇒ instr list
compStmt  :: java_mb ⇒ stmt ⇒ 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 (vn::=e) =
  compExpr jmb e @ [Dup, Store (index jmb vn)]
compExpr jmb (Call cn e1 mn X ps) =
  compExpr jmb e1 @ compExprs jmb ps @ [Invoke cn mn X]
```

```
compStmt jmb (Expr e) = (compExpr jmb e) @ [Pop]
compStmt jmb (c1;; c2) = (compStmt jmb c1) @ (compStmt jmb c2)
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])
```

Obviously, these definitions can directly be transformed into an ML-style functional program.

Compilation is then gradually extended to the more complex structures presented in Section 3.2.1, such as methods, classes and entire programs. Sine Java source and bytecode are sufficiently similar, hardly any data refinement is necessary. For lack of space, we skip their definition.

3.4.2 Compiler Correctness

Let us briefly review the compiler correctness statement and its proof. The presentation has to remain sketchy – we refer the reader to [Str02a] for a more faithful description.

Roughly speaking, the compiler correctness statement takes the form of the traditional "commuting diagram" argument: Suppose execution of a statement c transforms a Java state s into a state s'. Then, for any Java Virtual Machine (JVM) state t corresponding to s, executing the bytecode resulting from a translation of c yields a state t' corresponding to s'.

States on the Java source level essentially are a tripel: exception component (indicating the presence of an exception), heap `hp` and local variable assignment `loc`. The structure of states on the JVM level is more complex. However, for the purpose of compiler verification, we can focus on the heap, a local operand stack `os` and local variable assignment.

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

The diagram can now be interpreted as follows: Suppose that evaluation of expression `ex` in Java state `(None, hp, loc)` yields result `val` and state `(None, hp', loc')` (we assume that no exceptions arise during evaluation of the expression). When running the bytecode `compExpr mb ex` generated for `ex` in a JVM 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'`.

This result only holds under some preconditions, where the well-formedness constraints of Section 3.3.2 come into play again. Among others,

- the source program has to be well-formed: `wf_prog wf_java_mdecl G`
- expression `ex` is well-typed (for a specific environment `E`): `wtpd_expr E ex`

These requirements are not very restrictive, but they exist, and it is good to know that the preconditions of a correctness statement can be effectively verified, as outlined above and further expanded in the following.

4 From specifications to executable code

We now focus on the question of how to translate the Isabelle/HOL specifications presented in the previous sections into an executable program. As a target language, we chose the functional programming language ML. The main ingredients of Isabelle/HOL are *datatypes*, *recursive functions* and *inductive definitions*. Whereas the ML translation of the first two is relatively straightforward, translating inductive definitions is more challenging. Therefore, the rest of this section will mainly be devoted to this topic. The key idea behind the translation of inductive definitions is to interpret them as a *logic program* in the style of Prolog. A logic program is a set of so-called *Horn Clauses*, which have the form

$$q_1(u^1_1,\ \dots,\ u^1_{n_1}) \Longrightarrow \dots \Longrightarrow q_m(u^m_1,\ \dots,\ u^m_{n_m}) \Longrightarrow p(t_1,\ \dots,\ t_k)$$

Logic programs usually involve *nondeterminism*. This means that a query may have *multiple* or even *infinitely many* solutions, and that *backtracking* may be required while searching for a solution. Moreover, a predicate may be used in different ways in the sense that an argument of a predicate may either be treated as *input* or *output*. A mapping that marks each position as *input* or *output* is called a *mode*. In general, a predicate may have more than one possible mode. To be able to execute a predicate p as a functional program, we have to find a suitable execution order for the predicates $q_1, \dots, q_n$ in the body of each clause of p, which can be viewed as a kind of *dataflow analysis*: When execution starts, the only variables whose value is known are those occurring in input arguments in the clause head. Executing a predicate q_i in the body of p requires the values of all variables occurring in its *input arguments* to be known. After the execution of a predicate q_i, the values of all variables occurring in its *output arguments* are known and may then be used as input for subsequent executions of the remaining predicates. Finally, when all predicates in the body of the clause have been executed, all variables occurring in the clause head must be known. A mode for which we can find such an execution order is called *legal*. In the sequel, we will denote a mode by the set of indexes of all of its input arguments.

The ML translation of a predicate will be a function that takes as arguments the input arguments of the predicate and returns a lazy list of possible solutions. In place of *unification*, the translated logic program will use ML's *pattern matching* mechanism. Rather than formally defining the notions of legal modes and execution orders, we will illustrate them by means of an example. The interested reader may find the formal definition of these concepts in [BN00].

As an example for the translation, consider the predicate *E⊢ pe ↝ e :: T* of Section 3.3. Some of its clauses are

```
pNewC: is_class (prg E) C ⟹
  E⊢ (pNewC C) ↝ (NewC C) :: (Class C)

pCast: ⟦ E⊢pe ↝ e::Class C; is_class (prg E) D;  prg E⊢C≼? D ⟧ ⟹
     E⊢pCast D pe ↝ Cast D e ::Class D

pLAcc: ⟦ localT E v = Some T; is_type (prg E) T ⟧ ⟹
  E⊢pLAcc v ↝ LAcc v ::T

pBinOpAdd: ⟦ E⊢pe1 ↝ e1::T; E⊢pe2 ↝ e2::T ⟧ ⟹
  E⊢pBinOp Eq pe1 pe2 ↝ BinOp Eq e1 e2 ::PrimT Boolean
```

Apart from other inductive predicates such as *prg E⊢C≼? D*, the above clauses also refer to other *functions* such as *is_class*, *localT* and *is_type*. In contrast to *constructor functions*, other functions cannot easily be inverted in general. Therefore, their occurrence is restricted to *output positions* in the clause head and *input positions* in the clause body. We will now show that the mode *{1, 2}* is legal, i.e. we have to demonstrate that for each *E* and *pe*, we can compute *e* and *T* such that *E⊢ pe ↝ e :: T*.

- Consider clause *pNewC*. Assume *E* and *pNewC C* are known. We can then find out the value of *C* (since *pNewC* is a constructor) and hence know the value of *NewC C* and *Class C*. Therefore, the mode *{1, 2}* is legal for this clause.
- Consider clause *pBinOpAdd*. Assume *E* and *pBinOp Eq pe1 pe2* are known. We therefore know the value of *pe1* and *pe2*. For the recursive call, we may already assume that the mode *{1, 2}* is legal. Therefore, from *E* and *pe1*, we can recursively compute *e1* and *T*. By another recursive call, we can also compute *e2* from *E* and *pe2*. Since *BinOp Eq e1 e2* is now known completely, we may conlude that the mode *{1,2}* is also legal for this clause.
- Similar reasoning shows that the mode *{1,2}* is also legal for the remaining clauses.

The ML code generated for mode $\{1, 2\}$ of the above predicate is shown in Figure 2. The function `:->`, which is central to the translation, is defined as follows:

```
fun s :-> f = Seq.flat (Seq.map f s);
```

Its purpose is to compose subsequent calls of predicates, by feeding elements of the result sequence `s` of a predicate into the subsequent predicate `f`. The initial sequence `Seq.single inp` on the left of the predicate call chain con-

```
fun ty_pre_expr__1_2 inp =
  Seq.single inp :->
    (fn (E, pNewC C) =>
      ?? (is_class (fst E) C) :->
        (fn () => Seq.single (NewC C, RefT (ClassT C)) | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (E, pCast (D, pe)) =>
      ty_pre_expr__1_2 (E, pe) :->
        (fn (e, RefT (ClassT C)) =>
          ?? (is_class (fst E) D) :->
            (fn () =>
              cast__1_2 (fst E) (C, D) :->
                (fn () => Seq.single (Cast (D, e), RefT (ClassT D))
                  | _ => Seq.empty)
              | _ => Seq.empty)
          | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (E, pLAcc v) =>
      op__61__1 (snd E v) :->
        (fn (Some T) =>
          ?? (is_type (fst E) T) :->
            (fn () => Seq.single (LAcc v, T) | _ => Seq.empty)
          | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (E, pBinOp (Eq, pe1, pe2)) =>
      ty_pre_expr__1_2 (E, pe1) :->
        (fn (e1, T) =>
          ty_pre_expr__1_2_4 (E, pe2, T) :->
            (fn (e2) => Seq.single (BinOp (Eq, e1, e2), PrimT Boolean)
              | _ => Seq.empty)
          | _ => Seq.empty)
      | _ => Seq.empty) ++
  ...
```

Fig. 2. Code generated for inductive predicate

sists of just one element corresponding to the input of the predicate which is currently translated. Similarly, the last function in the chain returns a singleton sequence, which just contains the final result. Each clause may generate a sequence of possible solutions, which are concatenated using the ++ operator. It should be noted that we do not only allow inductive predicates to occur in the body of a clause, but also arbitrary side conditions, which are just boolean expressions. To make these fit into the our framework, we convert them into sequences using the function

```
fun ?? b = if b then Seq.single () else Seq.empty;
```

which returns the empty sequence if the condition evaluates to *False*, or the singleton sequence consisting of just a dummy element if the condition is *True*.

Also note that the equality constraint on the type *T* of the arguments of addition is taken care of by invoking the function generated for mode $\{1,2,4\}$.

5 Conclusions

This paper shows how important components of a compiler, such as type checking, type inference and code generation, can be formally verified in a proof assistant and then be made executable in a general-purpose programming language such as ML. The contribution of this paper is not primarily technical, as most of the methods have been described elsewhere (see citations throughout the text). Rather, we want to argue that a seamless formal development process is possible, without the ruptures (and, consequently, errors) introduced by verifying a compiler in one environment (such as a theorem prover) and then manually porting it to another environment or programming language where it can be executed.

The main mechanisms are:

- Explicit definition of executable functions (such as the compilation function *compExpr* in Section 3.4.1).
- Substitution of non-executable or inefficient functions by provably equivalent executable ones, as for function *appl_methds* in Section 3.3.2.
- Generation of executable functions out of inductive definitions, as for the typing relation of pre-terms in Section 4.

Similar mechanisms for code extraction are already provided in Coq [Log02] and could easily be integrated into other theorem provers.

There is a long history of formal language definition and compiler verification. To cite but a few: Vlisp [GMR$^+$92] was a major effort at formal compiler verification of Scheme down to machine code; however, it did not use any machine assistance for carrying out proofs.

The verification of a compiler from a Pascal dialect to a micro-controller assembler is described in [SCSW97,Ste98]. It uses Z as specification language, some proofs are performed in PVS, and the formalization has been ported to Prolog to make it executable. This work is impressive in that it is one of the few real-life case studies, carried out in an industrial setting. However, it also demonstrates the difficulty of having to deal with a multitude of formalisms.

Since the language underlying the ACL2 prover is a subset of Lisp, a major advantage of compiler verifications [You89,Goe00] carried out in this system is direct executability. However, as the logic is quantifier-free, some properties are awkward to express. In fact, the type systems of the languages in the cited references are very simple. For the complex type system of Java, extraction of a type checking function from concise typing rules is very convenient.

Using Abstract State Machines, a model of the Java language has been developed which is much more comprehensive than ours, together with a compiler [SSB01]. The semantics and compiler are executable; however, proofs have not been mechanically checked.

In our own approach, there are still several loose ends: From a fundamentalist perspective, it is reasonable to question that our development is fully formal, since it relies on the Isabelle code extraction facility which itself is a complex piece of code and thus subject to errors. Therefore, it could be worth while to formalize the extraction process.

On the more practical side, our coverage of the Java language is far from complete. Many operations are missing both on the source and the bytecode level, class initialization is still treated in an ad-hoc manner. Besides, it is not quite clear how to integrate libraries, such as for input/output, and how to generate useful error messages for incorrect programs.

A major issue is the verification of the as yet unverified steps in Figure 1: parser, assembler and printer. A proof checking approach [GHZG99,DV99] could be promising here.

Acknowledgement

A great part of the formalizations presented here has been developed by Gerwin Klein and David von Oheimb. The parser has been implemented by Tao Yu. We are grateful to Tobias Nipkow, Norbert Schirmer and Martin Wildmoser 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.

[DV99] Axel Dold and Vincent Vialard. Formal verification of a compiler backend generic checker program. In *Proc. of the Andrei Ershov Third International Conference Perspectives of System Informatics (PSI'99)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[GHZG99] T. Gaul, A. Heberle, W. Zimmermann, and W. Goerigk. Construction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends. In Amir Pnueli and Paolo Traverso, editors, *Proceedings of RTRV '99: Workshop on Runtime Result Verification*, Trento, Italy, 1999.

[GMR+92] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to vlisp, a verified programming language implementation. Technical Report M92B091, The MITRE Corporation, September 1992.

[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. To appear.

[Log02] Project Team LogiCal. *The Coq Proof Assistant Reference Manual, Version 7.3.1.* INRIA Rocquencourt – CNRS - ENS Lyon, May 2002.

[Mey] Jon Meyer. Jasmin. `http://mrl.nyu.edu/~meyer/jasmin/`.

[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/`.

[Sch03] Norbert Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 2003. To appear.

[SCSW97] David Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In *FME'97: Formal Methods: Their Industrial Application and Strengthened Foundations*, Lecture Notes in Computer Science, 1997.

[SML] SML-NJ. Standard ML of New Jersey homepage. `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`.

[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. Online version available from `http://public.logica.com/~stepneys/bib/ss/z/incdev.htm`.

[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.

[You89] William D. Young. A mechanically verified code generator. Technical Report 37, Computational Logic Inc., January 1989. Available from `www.cs.utexas.edu/users/boyer/ftp/cli-reports/037.pdf`.