Executing Verified Compiler Specification

Koji Okuma[1] and Yasuhiko Minamide[2]

[1] Doctoral Program in Engineering
University of Tsukuba
[2] Institute of Information Sciences and Electronics
University of Tsukuba
{okuma,minamide}@score.is.tsukuba.ac.jp

Abstract. Much work has been done in verifying a compiler specification, both in hand-written and mechanical proofs. However, there is still a gap between a correct compiler specification and a correct compiler implementation. To fill this gap and obtain a correct compiler implementation, we take the approach of generating a compiler from its specification. We verified the correctness of a compiler specification with the theorem prover Isabelle/HOL, and generated a Standard ML code corresponding to the specification with Isabelle's code generation facility. The generated compiler can be executed with some hand-written codes, and it compiles a small functional programming language into the Java virtual machine with several program transformations.

1 Introduction

Correctness of a compiler can be achieved by a correct specification and implementation. A specification of a compiler is formalized with a mapping between source and target languages. This mapping is required to be proved that it preserves a meaning between the two languages. An implementation of a compiler need to be proved correct in terms of this compiler specification. Our approach here, is to prove correctness of compiler specification and derive an implementation from it.

Our compiler takes a small functional programming language as input and compiles it to the Java virtual machine. The syntax of the source language is based on Scheme and it has the basic features of Lisp, such as lists and higher-order functions. We verified the correctness of its compiler specification with the theorem prover Isabelle/HOL [13] and generated a Standard ML code corresponding to the specification with Isabelle's code generation facility [2].

We specified the compiler with a subset of Isabelle/HOL including datatypes and primitive recursive functions that directly correspond to the functional language Standard ML. Although Isabelle/HOL can translate some other features, this restriction makes the translation more trustworthy and easier to coordinate with the hand-written parser and output routine. The other part of the specification including the semantics of the languages need not be executed. Therefore, it can be written with the full expressive power of Isabelle/HOL without this restriction.

A. Ohori (Ed.): APLAS 2003, LNCS 2895, pp. 178–194, 2003.

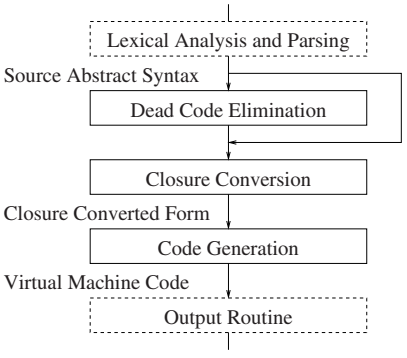

Fig. 1. The structure of our compiler

The structure of our compiler is shown in Figure 1. The closure conversion phase translates a program with lexically nested functions into a program with only top-level functions. The code generation phase translates a closure-converted program into a Java virtual machine code. Operational semantics of these three languages are formalized as inductively defined relations. We verified that all translations in the compiler preserve the meaning of a program. As an example of an optimization, we also formalized simple dead code elimination for the source language. The verification of this optimization was relatively easy compared to the other part of our verification.

To make our compiler executable, we wrote a parser and an output routine by hand in ML. In the figure, the code generated by Isabelle/HOL is shown in solid boxes, and the code written by hand is shown in dashed boxes. In addition to these routines, we supplied a small library routine written in Java to implement some primitives. However, these library routines are not verified in this paper. The resulting compiler successfully compiled several Scheme programs of 200–550 lines. The specification of our compiler can be obtained from `http://www.score.is.tsukuba.ac.jp/~okuma/vc/`.

This paper is organized as follows. In Section 2, we explain the notation used in this paper, and describe the code generation facility provided by Isabelle/HOL. In Sections 3 and 4, we describe the formalization of our compiler along its organization. In Section 5, we describe integration of the generated code and hand-written programs, and describe some experimental results. Finally, we review related work and present some conclusions.

2 Isabelle/HOL

In this section, we describe the basic features of Isabelle/HOL. In particular, we explain Isabelle/HOL's code generation facility, which is used to generate our executable compiler.

2.1 Isabelle/HOL

Isabelle is a generic, interactive theorem prover that can be instantiated with several object logics. Isabelle/HOL is an instantiation of Isabelle to Church's higher-order logic [13]. We wrote our proofs in Isabelle's new syntax Isar[18], which enabled us to write human-readable and structured proofs. We refer to Isabelle/HOL as HOL in the remainder of this paper.

HOL is equipped with the usual logical connectives, such as $\vee$, $\wedge$, $\rightarrow$, $\forall$, $\exists$. Types follow the syntax of ML, except the function arrow is $\Rightarrow$. HOL supports inductive datatypes similar to those of functional programming languages, and both primitive and well-founded recursive functions. In HOL, we can define a set inductively that we will use to formalize operational semantics.

2.2 Executable Specification in HOL

HOL has a facility to generate a code from a specification [2]. It translates several HOL elements into an executable Standard ML code. Elements that can be code-generated are: datatypes, recursive functions (both primitive recursive and well-founded recursive functions), and inductive definitions. The translation of datatypes and recursive functions is straightforward. However, the translation of an inductive definition is more complicated. HOL applies a mode analysis and generates a program that simulates a logic programming language.

Of the executable features of HOL, we only use features that directly correspond to those of ML, such as datatype definitions and primitive recursive functions. We do this because we think it would be easier to interact and the translation can be trusted. Note that these restrictions are only applied to the specification of the compiler. Semantics and other relations that are defined to prove correctness need not be executed. Specifications of a compiler can be written easily in functional style, and proofs can be written with HOL's full expressive power.

Consider the following specification in HOL. This example specifies the datatype for natural numbers and the primitive recursive function for the addition of natural numbers.

```
datatype
  nat = 0 | Suc nat

primrec
  "add 0 y = y"
  "add (Suc x) y = Suc (add x y)"
```

This program is translated into the following program in Standard ML by the code generation facility. Except for some renaming, this translation is straightforward and, therefore, we think that this translation can be trusted.

```
datatype Code_nat = nat_0 | nat_Suc of Code_nat
```

```
fun add nat_0 y = y
  | add (nat_Suc x) y = nat_Suc (add x y)
```

However, the translation described above has a problem. Types defined in HOL are translated into new types in Standard ML, and there is no way to obtain interoperability with existing datatypes in ML. We used HOL's method, which translates types in HOL to those of ML directly, to overcome this problem. In the following specification, the type *nat*, and the constants *0* and *Suc* are translated into *int* and corresponding operations in Standard ML. This translation is done literally; underscore '_' in the description is substituted by the arguments of the original terms. There is no check in this translation with respect to either syntax or semantics of ML. Therefore, care must be taken when specifying this correspondence.

```
types_code
  "nat"         ("int")

consts_code
  "0"           ("0")
  "Suc"         ("(_ + 1)")
```

This form of translation is essential to obtain interoperability with the parser and the output routine as well as reasonable performance of the generated code.

3 Source Language and Closure Conversion

The specification of our compiler consists of an abstract syntax and compilation functions. To verify correctness of the specified compiler, the semantics of each language must also be specified. We proved that each translation preserves a meaning with respect to their semantics. We now describe the specification of the source language and closure conversion.

3.1 Source Language

Our source language is a pure subset of Scheme, which has the following features: lexical scoping, higher-order functions and list manipulation. As basic data types, we include boolean values, integers and symbols. Features currently not supported include: mutually recursive functions, imperative features and continuation.

We represented our source language as an abstract syntax defined by the following datatype declaration in HOL:

```
datatype exp = NumExp int
             | BoolExp bool
             | SymExp symbol
             | NullExp
             | VarExp variable
```

```
        | UniExp uop exp
        | BinExp bop exp exp
        | IfExp exp exp exp
        | LetExp variable exp exp
        | FunExp variable "variable list" exp exp
        | AppExp exp "exp list"
```

The type `variable` is a synonym of `nat`, which is the built-in datatype to express natural numbers. To represent base types, such as symbols and integers, we introduced new types `symbol` and `int`. Primitive operators are either unary or binary, and expressed by datatypes `uop` and `bop`. Note that, `FunExp` in datatype specifies not only a function definition but also a main expression to be evaluated. The expression below defines a function that adds two arguments, and evaluates its application with arguments 1 and 2.

```
constdefs
  addprog :: exp
  "addprog == FunExp 0 [1, 2]
               (BinExp PlusOp (VarExp 1) (VarExp 2))
               (AppExp (VarExp 0) [(NumExp 1), (NumExp 2)])"
```

The semantics of the source language is specified as big-step natural semantics. We define this as a relation on a value environment, a source expression and a value. We defined datatypes `value` and `env`, to represent values and environments as follows:

```
types  'a env = "(variable * 'a) list"

datatype value = IntVal int
               | BoolVal bool
               | SymVal symbol
               | NullVal
               | ConsVal value value
               | ClsVal variable "value env" "variable list" exp
```

An environment is represented as an association list. A value `ClsVal f env xs exp` represents the closure of a recursive function where `f` is the name of the function, `env` is the environment mapping each free variable to a value and `xs` is the list of formal parameters.

Finally, we define the semantics of this language as an inductively defined relation `eval` as:

```
consts  eval  :: "(value env * exp * value) set"

inductive eval
  intros
  var  : "⟦lookup E n  = Some v⟧ ⟹ (E, VarExp n, v) ∈ eval"
  lete : "⟦(E, e1, v1) ∈ eval;
           ((x, v1)#E, e2, v) ∈ eval⟧
             ⟹ (E, LetExp x e1 e2, v) ∈ eval"
  ...
```

The definition above shows the case of a variable and a `let`-expression. The definition of the semantics is standard. Please refer to the proof scripts for details.

3.2 Closure Conversion

The first phase of our compiler is closure conversion, which achieves a separation between code and data. This translates a lexically nested function to a code abstracted with an extra environment that contains the free variables occurring in the original function. After this translation, the code becomes closed, and separated from the data. This enables nested functions to be defined at the top level and shared by all closures that are instances of these functions. Consider the following program written in Scheme.

```
(define (f x y) (let ((g (lambda (z) (+ x y z)))) g))
```

The function `g` contains free variables `x` and `y`. Closure conversion eliminates direct references to these variables by replacing them with references to an explicit environment. The above program is translated to the following program.

```
(define (g env z) (let ((x (car env))
                        (y (cdr env)))
                    (+ x y z)))

(define (f x y) (cons g (cons x y)))
```

In this example, the closure and the environment are implemented with cons cells.

Conceptually, closure conversion divides into two phases.

1. The first phase makes all functions closed by abstracting them with an environment.
2. The second phase lifts all functions to the top level.

Most of the previous formal accounts of closure conversion formalize only the first phase [7,11], but compilers usually perform these two phases as one transformation. Because we need to extract an executable compiler from the specification, we formalized closure conversion as a transformation that performs these two phases simultaneously.

The target language of closure conversion differs from the source language in that: it does not include a function definition, but an operation for creating a closure explicitly. A program in the target language is represented as a pair of declaration of top level functions and an expression evaluated under the declaration. The following are datatypes to represent the target language.

```
types  decl  = "variable list * variable list * cexp"
       decls = "decl env"

datatype cexp = ···
              | MkCls variable "variable list"
```

```
primrec
  "clsconv (VarExp x)          = (VarExp x, [])"
  "clsconv (FunExp f as e1 e2) = let (e1', ds1) = clsconv e1 in
                                 let (e2', ds2) = clsconv e2 in
                                 let xs = diff (fv e1) (f#as) in
                                 (LetExp f (MkCls f xs) e2',
                                  ins (f, xs, as, e1')
                                      (union ds1 ds2))"
  "clsconv (AppExp f es)       = let (f',  ds1) = clsconv f in
                                 let (es', ds2) = clsconv_list es in
                                 (AppExp f' es', union ds1 ds2)"
  ...
```

Fig. 2. Closure conversion

The type *decls* represents a declaration of top level functions and is a mapping from variables to definitions of functions. The definition of a function consists of the list of free variables, the list of formal parameters and its body. The expression of the target language, *cexp*, includes the operation *MkCls f xs* that creates a closure of the function *f* with the environment containing values of *xs*.

We formalized closure conversion as a primitive recursive function with the following type.

```
consts
  clsconv :: "exp ⇒ cexp * decls"
```

As we described above, the conversion function translates an expression in the source language into a pair of an expression in the target language and a declaration of top level functions. The main part of the translation is shown in Figure 2. To transform a function definition, *e1* and *e2* are translated into *e1'* and *e2'*, respectively. It also produces declarations *ds1* and *ds2*. To construct the closure of the function, we need to compute the set of free variables that occur in the function body by *diff (fv e1) (f#as)*. Finally, the declaration corresponding to the function *f* is inserted into the union of *ds1* and *ds2*.

In this definition, we represented sets by lists and used the following manipulation functions:

```
consts
  fv :: "exp ⇒ variable list"
  diff    :: "['a list, 'a list] ⇒ 'a list"
  union   :: "['a list, 'a list] ⇒ 'a list"
```

where *fv exp* computes the set of free variables of *exp*. These functions are easily defined as primitive recursive functions. HOL has the built-in type constructor *set* to represent possibly infinite sets, and automated theorem-proving methods work very well for operations on *set*. However, we could not use them for closure conversion because an executable code cannot be extracted for the operations on *set*. Many lemmas on our set operations are required in proving correctness of

the translation. To eliminate unnecessary proof work, we showed that each set operation we defined is compatible with the corresponding built-in set operation. For example, the following lemma shows that the union of two lists is equivalent to the union of two sets. The function *set* used in this example is a conversion function from lists to sets.

```
lemma union_set: "set (union A B) = set A ∪ set B"
```

With these lemmas, most lemmas on our set operations are proved automatically.

3.3 Verification of Closure Conversion

We proved correctness of closure conversion in HOL. To formalize correctness of the transformation, we first introduce the following datatype representing observable values.

```
datatype ovalue = IntVal int
                | BoolVal bool
                | SymVal symbol
                | NullVal
                | ConsVal
                | ClsVal
```

We consider integers, boolean values, symbols and null as observable values, but the detailed structures of cons cells and closures are ignored. Values of the source and target languages are coerced into observable values by the following functions.

```
consts
  value2obs :: "value ⇒ ovalue"
  cvalue2obs :: "cvalue ⇒ ovalue"
```

The following is the correctness theorem verified in HOL. The theorem says that if *M* is evaluated to *v* under the empty environment *[]*, the translated expression *M'* is evaluated to *v'* under the empty environment *[]* and the declaration *D* generated by closure conversion. Furthermore, the values *v* and *v'* correspond to the same observable value.

```
theorem assumes "([], M, v) ∈ Lang.eval "  "fv M = []"
  shows "let (M',D) = clsconv M in func D ⟶
         (∃v'. (D, [], M', v') ∈ CLang.eval ∧
                value2obs v = cvalue2obs v')"
```

Note an extra condition *func D* in the theorem. This condition is necessary to lift function definitions to the top level.

```
constdefs
  func :: "('a, 'b) list ⇒ bool"
  "func D == (∀f x y. elem (f, x) D ⟶ elem (f, y) D ⟶ x = y)"
```

The function `elem` is the membership predicate on a list. This condition is satisfied if function names are distinct. In our hand-written parser, each variable is renamed to a fresh integer value to satisfy this condition.

To prove the theorem above, we needed to prove a generalized statement, which was proved by induction on the derivation of $(E, M, v) \in eval$. The proof consists of approximately 750 lines.

4 Code Generation

The code generation phase of our compiler translates a closure-converted program into a Java virtual machine code. We chose the Java virtual machine (JVM) as the target of our compiler for the following reasons. First, it is easier to translate our source language into JVM than a conventional machine language, because its instructions are stack-based and we can assume memory management of JVM. Secondly, a formalization of JVM has been studied extensively in several works for type soundness of JVM and there are several works that have already formalized it using theorem provers [8,9].

For the verification of our compiler, we formalized JVM from the beginning because existing formalizations do not include sufficient instructions to generate an executable code for our compiler and we can also simplify our verification by restricting features of the virtual machine.

The first step of our verification is to clarify what should be verified. Our compiler consists of the code generated from the verified specification and the small quantity of hand-written code. The hand-written code contains a small library written in Java to implement some primitives of the source language. In our verification, we focused on the verification of the core of the compiler and did not verify correctness of the library. For real programming languages, implementation of some primitives are complicated and their correctness proofs will be non-trivial. We think that a verification of the library is a different issue.

The rest of our verification is similar to the verification of closure conversion, but it was much more difficult than that of closure conversion. The specification of code generation and its verification is approximately 2400 lines.

4.1 Formalization of a Virtual Machine

We formalized a small subset of the Java virtual machine, so that it is sufficient to generate the code of the source language. The instruction set is restricted to a small subset and the form of an object is also restricted. The following summarizes the restrictions of our virtual machine.

- classes have only one method
- branching instructions do not jump backward
- no interface, no exception, no thread and no inheritance

Our compiler translates a closure into an object with one method and each function is translated into a class. To simplify the virtual machine, we did not

introduce a class with multiple methods. Branch instructions only jump forward because our source language does not have a loop construct.

Instructions of our virtual machine are formalized by the following datatype:

```
datatype
  'a instr = ALoad nat
           | IAdd
           ...
           | Dup
           | New cname
           | Invoke nat
           | PrimCall 'a
```

where the type variable `'a` is used to parameterize the set of instructions with primitives provided by the library. We briefly describe the instructions defined above. `ALoad` instruction loads an object reference from local variables. `New f` makes a new instance of class `f`. `Invoke` instruction invokes a method of the class corresponding to closures. It only takes a number of arguments, since an object can be determined from the operand stack and each object has only one method. `PrimCall` instruction is a pseudo instruction introduced to call primitives.

The following is a part of the primitives implemented in the library.

```
datatype
  prim = PrimMkCons
       | PrimCar
       | PrimCdr
       ...
```

We split the specification of semantics of these primitives from the specification of the virtual machine. The semantics of each of these primitives is specified as a relation describing their effects on a state of the virtual machine. The specification below, defines the semantics of the primitives.

```
consts
  prim_spec :: "(prim heap * vmVal list * prim * prim heap *
vmVal) set"

inductive prim_spec
intros
  mkcons : "l ∉ dom H ⟹
     (H, [v2,v1], PrimMkCons,
      H(l ↦ PrimObj PrimMkCons [v1, v2]), VmAddr l) ∈ prim_spec"
  car : "H(l) = Some (PrimObj PrimMkCons [v,v'])
     ⟹ (H, [VmAddr l], PrimCar, H, v) ∈ prim_spec"
   ...
```

For example, the primitive `PrimMkCons` creates a fresh cons cell `PrimObj PrimMkCons [v1, v2]` on the heap for the arguments `v1` and `v2`. The heap of the virtual machine is also abstracted with primitives to represent abstract objects with `PrimObj`. We assume that the primitives implemented in the library satisfy the above specification.

```
consts
  cgExp :: "variable list ⇒ cexp ⇒ prim instr list"
primrec
  "cgExp E (NumExp n) = [Ldc n, PrimCall PrimMkInt]"
  "cgExp E (VarExp x) = [ALoad (the (assign E x))]"
  "cgExp E (MkCls f xs) = (New f) # cgCls E f xs"
  "cgExp E (AppExp f es) = (cgExp E f) @ [CheckCls] @
                           (cgExps E es) @ [Invoke (length es)]"
  ...
```

Fig. 3. Code generation function

The semantics of the virtual machine is based on the semantics of primitives, and specified as small-step natural semantics. The semantics is defined as an inductively defined relation with the following type:

```
consts
  exec :: "(class * prim heap * frame * prim codes *
                  prim heap * frame) set"
```

Type `frame` used above, is a record of an operand stack and a local variable environment. `(D * H * F * C * H' * F') ∈ exec` means, the heap `H` and the frame `F` are transformed to `H'` and `F'` respectively by the code `C` under the class declaration `D`. We write $D \vdash < H, F, C > \rightsquigarrow < H', F' >$ to describe this relation.

4.2 Code Generation

The code generation of our compiler is similar to that of other compilers from functional languages into JVM [3,1]. We explain the major section of the code generation phase and the main theorem we verified. The main part of the code generator is the function that translates an expression into a list of instructions as shown in Figure 3. In the definition, integer expression `NumExp n` is translated into the instruction sequence: `Ldc n` instruction puts an integer constant `n` on the operand stack, and `PrimMkInt` primitive creates a new integer object from it.

A variable is translated to the load instruction of the local variable obtained by an assignment function. The assignment function translates a variable into a local variable according to a list of variables `E`. It assigns the n-th variable from the tail of the list to the n-th local variable. This assignment is specified by the following primitive recursive function.

```
consts
  assign :: "'a list ⇒ 'a ⇒ nat option"
primrec
  "assign [] n     = None"
  "assign (x#xs) n = (if n = x then Some (length xs)
                               else assign xs n)"
```

This translation of a variable is correct only if the list of variables used in the translation satisfies some condition: intuitively, variables in the list must be distinct, but a weaker condition is sufficient. This is guaranteed by the well-formedness of the declaration we discuss later in this section.

A function application is translated to a sequence of: a code for the function part, codes for arguments and method invocation. `CheckCls` pseudo instruction interleaved in this sequence checks that the function translated is an instance of a closure class. This check is required so that a generated code is type-checked by the bytecode verifier of JVM and translated into `checkcast` instruction of JVM.

The most complicated part of the code generation phase is the handling of closures. A function is represented by a class with one "apply" method. This class may contain instance variables to keep values for the free variables. When compiling a function, instructions to create a new instance of the corresponding class and to store all free variables into the closure is inserted. Storing all free variables into the closure is accomplished by the following function.

```
consts
  cgCls :: "variable list ⇒ cname ⇒ variable list ⇒ prim
codes"
primrec
  "cgCls E f [] = []"
  "cgCls E f (x#xs) = (cgCls E f xs) @
                      [Dup,ALoad (the (assign E x)),PutField f x]"
```

It can be seen that each value is loaded from the corresponding local variable and stored to a class instance by `PutField`.

The following is the correctness theorem we verified in HOL.

```
theorem assumes "(D, [], e, v) ∈ eval" "wf_decls D"
  shows "∃H' F'.cgDecls D⊢<newheap,newframe,cgExp[]e>⇝< H′, F′ >
                    ∧ (D, H', v, hd (opstack F')) ∈ valeq"
```

The generated code `cgExp [] e` is executed under the empty heap `newheap` and empty frame `newframe`. The theorem says, if expression `e` is evaluated to a value `v` under a declaration `D`, the execution of the generated code transforms the empty heap and frame into heap `H'` and frame `F'`. Furthermore, the value that is on the top of operand stack `F'` corresponds to the value `v` under the class declaration `D` and the obtained heap `H'`.

In this correctness theorem, well-formedness of the declaration `D` is essential. To satisfy this condition, we defined a well-formedness predicate on the source language as:

- all function names must be distinct.
- the variable names of the arguments of a function must be distinct.
- a function name and the variable names of its arguments must be different.

If a source program is well-formed, we can show that the closure conversion produces a well-formed declaration. In addition, we can show the condition of

closure conversion shown in Section 3.3 can be satisfied by this predicate. Therefore, the entire compilation process is correct assuming that the source program is well-formed.

Note that our verification of the code generator has several limitations. Our specification of the compiler does not calculate the size of the operand stack and the number of local variables used in each method. It is not verified that the compiler produces a code that is compliant with the bytecode verifier of JVM.

5 The Generated Compiler and Its Evaluation

The specification and correctness proof of the compiler is approximately 5000 lines of HOL proof scripts. From this specification, 300 lines of an executable Standard ML code is generated. To make the compiler executable, we provided a parser and an output routine written in ML and some library code written in Java. Codes written in ML and Java are approximately 700 lines and 120 lines respectively. In this section, we describe integration with hand-written codes and then present some preliminary experiments with the resulting compiler.

5.1 Integration with Hand-Written Codes

From the specification, we obtain the abstract syntax of the source language. Types that appears in the abstract syntax are translated to types of ML, according to the following declaration:

```
types_code
  "variable"    ("int")
  "int"         ("int")
  "symbol"      ("string")
```

Among these declarations, translations of `int` and `symbol` are essential to coordinate with the parser. The type `variable`, which is a type synonym of `nat`, can be used in our compiler. However, it is obviously inefficient to take `nat` as a datatype for the variables and compilation time became from two times to eight times more inefficient in our experiment. Therefore, we applied this translation. In the parser, each identifier that occurs in source code is replaced by a fresh integer value, so that the well-formedness condition required by the correctness proof is satisfied.

The output routine produces an assembly code in the syntax of the Jasmin bytecode assembler [10] and this assembler is used to generate a Java bytecode. Because the instructions of our virtual machine almost directly correspond to those of the Java virtual machine, writing the output routine for the compiler is straightforward. One twist was needed in the output routine because a branch instruction of our virtual machine takes a relative address and Jasmin uses a label to specify the target address of a branch instruction.

We also formalized a simple dead code elimination. This eliminates a function definition that is not used in the main expression. The verification of this

Table 1. Compilation time (in seconds)

program	#lines	vc total	vc w/o jasmin	bigloo	kawa
symbdiff	376	1.87	0.08	0.46	2.90
boyer	552	2.13	0.11	0.32	2.82
sets	349	1.74	0.04	0.55	2.62
sk	181	1.59	0.05	0.25	2.34
art	3008	54.44	51.56	4.22	3.82

optimization was relatively simple compared to the remainder of our verification and completed in approximately one day. This optimization works in combination with the other part of our compiler and eliminates unused primitives and library procedures.

5.2 Experimental Results

We tested the generated compiler on several programs: a symbolic differentiation program (symbdiff), a tautology checker (boyer), a set module implemented with a balanced binary tree (sets) and a translator from the lambda calculus to combinatory logic (sk). These examples were selected from the Scheme repository and successfully compiled/executed with few modifications.

We compared compilation times of these programs with the existing Scheme to Java compilers Bigloo and Kawa[1]. A 650MHz Ultra SPARC IIi processor with 256MB of memory, using: Poly ML 4.1.3, Sun J2SDK 1.4.1, Bigloo 2.6a and Kawa 1.7 was used for the experiment. Through this comparison, we refer our compiler as vc (verified compiler). Table 1 shows compilation times. The column "vc total" shows the compilation time including the time spent by Jasmin, and the column "vc w/o jasmin" shows the compilation time without Jasmin. The program "art" is an artificial large example with a large number of variable definitions. Compilation time of this program is more than ten times slower than the time for the other two compilers. We consider this problem is caused by the naive implementation of various data structures: we represented sets by lists and used association lists to handle identifiers. We think refining the implementation of data structures used in the compiler is one of the first tasks to make the compiler realistic.

We also conducted preliminary experiments regarding execution times. we chose the programs: Fibonnatti function, Takeuchi's function, eight queen program and tautology checker. Results are shown in Table 2. The column "bigloo opt" shows the execution time of the code generated by Bigloo with optimization. Note that the comparison is disadvantageous to Bigloo and Kawa because our compiler supports only a small subset of Scheme and that simplifies a compilation. Regardless of that, execution times for Bigloo with optimization are much

[1] Bigloo is written in C, and Kawa is written in Java. Both compilers were tested without any command line options

Table 2. Execution time (in seconds)

program	vc	bigloo	bigloo opt	kawa
fib	32.12	33.14	1.31	42.72
tak	0.48	0.74	0.62	1.58
queens	2.14	2.13	0.98	12.47
boyer	69.37	6.33	3.62	262.52

faster than those of our compiler. To generate more efficient code, we need to improve the translations in our compiler and incorporate standard optimizations such as inlining and constant folding.

6 Related Work

Various system components including language implementations were verified using the Boyer-Moore theorem prover [4,12,19]. The Boyer-Moore theorem prover is first-order and its specification language is Lisp based. Therefore, their specification of the language implementations can be executed. Although it is possible to verify compilers in theorem provers based on a first-order logic as their works demonstrated, expressiveness of a higher-order logic was of considerable assistance in verifying the correctness of our compiler.

Oliva, Ramsdell and Wand verified a specification of the VLISP compiler [14]. VLISP compiles a dialect of Scheme designed for system programming. The semantics of the language is described in denotational semantics. They wrote a correctness proof of the compiler specification by hand. The compiler is written based on this specification, but there is no verification of the implementation.

Stepney wrote a specification of a compiler in the Z specification language and proved correctness of the compiler specification by hand [16]. The specification was translated into an executable Prolog program by a syntax-based hand-translation. Stringer-Calvert translated the specification of this compiler into PVS [15] and verified its correctness [17]. The approach of this work is closely related to ours in sense that the executable compiler is obtained from the specification. However, we believe that the code generation of HOL is more trustworthy and we obtain a more realistic compiler because the translation is more direct.

The same approach of generating a compiler from the specification was taken by Curzon [5]. He verified the correctness of an assembler for the high-level assembly language Vista with the HOL theorem prover [6] and generated an executable assembler in ML with an automated tool. However, issues concerning the resulting compiler and its evaluation are not described in detail.

7 Conclusions and Future Work

We have verified a specification of a compiler from a functional language to the Java virtual machine in Isabelle/HOL. An executable compiler was derived from the specification with the code generation facility of Isabelle/HOL. In our development, most time was spent in proving correctness of the compiler. We think recent progress in theorem proving have made verification of compilers more feasible. Especially, the human-readable structured proof language of Isabelle[18] made our verification easier.

The compiler was tested for several Scheme programs and the compile times were acceptable. However, the compilation of an artificial large program revealed inefficiency in the compiler.

We are planning to refine and extend our compiler in various respects to make it more realistic. To compile large examples, we will need to extend the features supported by the source language and improve data structures used in the compiler. With respect to performance of code generated by the compiler, first we wish to refine the algorithms used in the current specification. Second, we intend to introduce several basic optimizations into our compiler specification, such as inlining.

References

1. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.
2. S. Berghofer and T. Nipkow. Executing higher order logic. In *Proceedings of International Workshop on Types for Proofs and Programs*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer-Verlag, 2002.
3. P. Bothner. Kawa—compiling dynamic languages to the Java VM. In *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track*, New Orleans, LA, 1998. USENIX Association.
4. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
5. P. Curzon. A verified Vista implementation final report. Technical Report 311, University of Cambridge Computer Laboratory, 1993.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge, 1993.
7. J. Hannan. A type system for closure conversion. In *Proceedings of the Workshop on Types for Program Analysis*, pages 48–62, 1995.
8. G. Klein and T. Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13:1133–1151, 2001.
9. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298:583–626, 2003.
10. J. Meyer. Jasmin home page. `http://mrl.nyu.edu/~meyer/jasmin/`.
11. Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *Proceedings of Symposium on Principles of Programming Languages*, pages 271–283, 1996.

12. J. S. Moore. A mechanically verified language implementation. Technical Report 30, Computational Logic Inc., 1988.
13. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
14. D. P. Oliva, J. D. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.
15. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.
16. S. Stepney. *High Integrity Compilation : a case study*. Prentice-Hall, 1993.
17. D. W. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, Department of Computer Science, University of York, Mar. 1998.
18. M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proceedings of International Conference on Theorem Proving in Higher Order Logics*, pages 167–184, 1999.
19. W. D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic Inc., 1988.