Refinement Algebra for Formal Bytecode Generation

Adolfo Duran, Ana Cavalcanti and Augusto Sampaio

Centro de Informática
Universidade Federal de Pernambuco
Po Box 7851 50740-540 Recife PE Brazil
Fax: +55 81 32718438, e-mail:{aad,alcc,acas}@cin.ufpe.br

Abstract. In this paper we propose an strategy for the design of compilers correct by construction for object-oriented languages. The process is formalized within a single and uniform semantic framework of an object-oriented language based on a subset of sequential Java and its algebraic laws. The strategy is to reduce an arbitrary source program to a particular normal form which describes the behavior of the target machine. This behavior is defined by an interpreter written in the same language. From the interpreter we can capture the sequence of generated bytecodes of the target machine. The normal form reduction is formalized as algebraic transformations where the central notion is refinement of programs. Thus, compilation is reduced to program refinement. This avoids translations between semantics as the product of the compilation is a program in the same language.

Keywords: algebraic transformation; refinement calculus; compiler correctness; object-orientation.

1 Introduction

In the literature one can find several approaches that have been used to generate correct compilers for imperative languages [5, 6, 10]. The design of correct compilers for procedural languages is already understood; our main challenge is the development of an approach to deal with object-oriented features.

We propose an algebraic approach to construct a provably correct compiler for an object-oriented language called ROOL (for Refinement Object-oriented Language)[1, 3], which is similar to sequential Java and C++ [9]. This language includes classes, inheritance, dynamic binding, recursion, type casts and tests, and class-based visibility.

We carry out compilation by a series of refinement steps identified with the reduction of an arbitrary source program to a program in a particular normal form: an interpreter executing target code. From the normal form program, we can capture the sequence of bytecodes for a ROOL Virtual Machine (RVM), which is a subset of the Java Virtual Machine (JVM). The immediate correctness criterion is to require that the source program is refined by the interpreter.Thus, the compilation task is reduced to program refinement.

This approach was originally described in [4] and has been further developed in [7]. It characterizes the compilation process within a uniform framework, where comparisons and translations between semantics are avoided. This constitutes the main advantage of our approach since it results in simplicity.

In this paper, we illustrate how compilation is carried out using compilation rules. The compilation process is split into three phases: simplification of expressions, data refinement, and control elimination. Only the code generation phase of a compiler is addressed. We assume that there are no name clashing for variables and attributes in the input program.

This paper is organized as follows. We first give an overview of ROOL, including some of its basic laws. Afterwards we explain the ROOL interpreter structure. Subsequently, we explain how we compile with compilation rules; we present an example to illustrate how the lemmas and basic laws are used to prove the compilation rules. Finally, we summarize the results achieved so far, and consider related work and topics for further research.

2 The ROOL Language and Its Laws

A program in ROOL consists of a sequence of class declarations, followed by a main command ($CDS \bullet c$). A class declaration has the form

$$
\begin{array}{ll}
\textbf{class } N_1 \textbf{ extends } N_2 & \\
\quad \{\textbf{pri } x_1 : T_1;\ \}^* & //\textit{private attributes} \\
\quad \{\textbf{prot } x_2 : T_2;\ \}^* & //\textit{protected attributes} \\
\quad \{\textbf{pub } x_3 : T_3;\ \}^* & //\textit{public attributes} \\
\quad \{\textbf{meth } m \triangleq (pds \bullet S)\ \textbf{end}\}^* & //\textit{public methods} \\
\quad \{\textbf{new} \triangleq x := e\ \textbf{end}\}^* & //\textit{Initializers} \\
\textbf{end} &
\end{array}
$$

The clause **extends** determines the immediate superclass of N_1. If omitted, the built-in class **object** is regarded as the superclass. The visibility mechanism is similar to that of Java: the qualifiers **pri**, **prot**, and **pub** are used for private, protected, and public attributes. The clause **meth** declares a method. For simplicity, all methods are considered to be public. The list of parameters of a method is separated from its body by the symbol '•'. The **new** clause declares initializers: methods called after creating an object of the class.

Data types T are the types of attributes, method parameters, local variables, and expressions. They are either primitive (like boolean or integer) or class names N. For variable identifiers, we use x, whereas f stands for a literal or buit-in function; we also use b for boolean expressions and X for a recursive block identifier. Subscripts are used to extend the set of metavariables. Table 1 describes the rules to generate expressions.

The **self** and **super** references are similar to the *this* and *super* of Java. An update expression has the form $(e_1;\ x : e_2)$ and denotes a fresh object copied from e_1, but with attribute x mapped to a copy of e_2. The expressions that can appear as targets of assignments, method calls, and as result and value-result arguments are called *left expressions* (le).

$e \in Exp ::=$	**self** | **super**	special 'references'
	| **null**	null 'reference'
	| **new** N	object creation
	| $x \mid f(e)$	variable, built-in application
	| e **is** $N \mid (N)e$	type test, type cast
	| $e.x \mid (e;\ x : e)$	attribute selection, update expression

Table 1. Grammar for expressions

$c \in Com ::=$	$le := e$	assignment
	| $c_1;\ c_2$	sequential composition
	| $x : [pre, post]$	specification statement
	| **if** $[]_{\langle 1 \leq i \leq n \rangle} \bullet b_i \rightarrow c_i$ **fi**	conditional
	| **rec** $X \bullet c$ **end** | X	recursion, recursive call
	| **var** $x : T \bullet c$ **end**	local variable block
	| $pc(e)$	parameterized command application

Table 2. Grammar for commands

In addition to method calls, the main command, the body of methods, and the initializers may have imperative constructs. Table 2 describes the imperative constructs of ROOL based on Dijkstra's language of guarded commands. The specification statement $x : [pre, post]$ describes a program that, when executed in a state that satisfies the precondition pre, terminates in a state that satisfies the postcondition $post$, modifying only variables in x. We call x a frame and it stands for a finite sequence of variables identifiers.

The conditional command is composed by guarded commands of the form $b_i \rightarrow c_i$. At least one guard b_i must be true, otherwise the execution aborts. More than one guard can be true at the same time; in this case one of them is non-deterministically chosen and its command is executed.

Methods are seen as parameterized commands, which can be applied to a list of arguments to yield a command. A parameterized command can have the form **val** $x : T \bullet c$, **res** $x : T \bullet c$, or **vres** $x : T \bullet c$, corresponding to the conventions of parameter passing known as call-by-value, call-by-result, and call-by-value-result. Table 3 describes the syntax of parameterized commands. In [3] we have further details of ROOL and its formal semantics based on weakest preconditions.

The laws of ROOL are an algebraic semantics for this language and establish a sound basis for the design of correct compilers. In [1] many laws have been proved correct with respect to the weakest precondition of ROOL. Here we introduce a subset of the basic laws we use later on. We say that $CDS, x : T, N \triangleright c_1 = c_2$, when the commands c_1 and c_2 are equal, in the context of a sequence of class declarations CDS, visible attributes, parameters, and local variables $x : T$, and class N. In what follows, we remove the context, when it is not relevant for the law.

$pc \in PCom$	$::=$	$pds \bullet c$	parameterization
		$\mid le.m \mid m$	method calls
$pds \in Pds$	$::=$	$\varnothing \mid pd \mid pd;\ pds$	parameter declarations
$pd \in Pd$	$::=$	$\mathbf{val}\ x : T \mid \mathbf{res}\ x : T \mid \mathbf{vres}\ x : T$	

Table 3. Grammar for parameterized commands

We use the abbreviation **skip** for the command that does nothing; its execution always terminates and leaves the state unchanged. This command can be defined as: [*true*, *true*]. To precede or follow a command by **skip** does not change its effect.

Law 1 $(\mathbf{skip};\ c) \ = \ c \ = \ (c;\ \mathbf{skip})$

The assignment of a variable to itself does not change anything.

Law 2 $(\mathrm{v} \ := \ \mathrm{v}) \ = \ \mathbf{skip}$

If exactly one of the guards of a conditional is true, the corresponding command is selected for execution.

Law 3 If i and j range over $1..n$ and $\neg\ (b_i \wedge b_j)\ \mathit{with}\ (i \neq j)$, then
$[b_j];\ \mathbf{if}\ []_{\langle 1\leq\ i\ \leq\ n\rangle}\ b_i\ \rightarrow\ c_i\ \mathbf{fi}\ =\ [b_j];\ \ c_j$

The expression $[b_i]$ above is an assertion. Whenever the flow of control reaches an assertion, it is checked: if false, the program behaves like *miracle*; if true, execution continues normally. The command miracle can serve any purpose; it has the most refined behavior: It is infeasible and cannot be implemented, but constitutes a useful theoretical concept for reasoning.

Although the **while** statement does not appear in the ROOL grammar, we use it in our interpreter. This is merely a syntactic sugar to improve readability; **while** is easily defined using recursion and conditional statements, as follows.

Definition 1 (While Statement)
$\mathbf{while}\, b\ \bullet\ c\ \mathbf{end} \overset{def}{=} \mathbf{rec}\ X\ \bullet\ \mathbf{if}\, b\ \rightarrow\ (c;\ \ X)\ []\neg\ b\ \rightarrow\ \mathbf{skip};\ \ \mathbf{fi}\ \mathbf{end}$

A while can be eliminated if an assertion before it implies that its condition does not hold initially.

Law 4 If $(b_1 \Rightarrow \neg b_2)$, then $[b_1];\ \mathbf{while}\ b_2\ \bullet\ c\ \mathbf{end}\ =\ [b_1]$

If, on the contrary, the assertion before the **while** implies that its condition holds, it behaves like c followed by the whole iteration.

Law 5 If $(b_1 \Rightarrow b_2)$, then $[b_1];\ \mathbf{while}\ b_2 \bullet c\ \mathbf{end}\ =\ [b_1];\ \ c;\ \ \mathbf{while}\ b_2 \bullet c\ \mathbf{end}$

The symbol $\sqsubseteq$ denotes the refinement ordering on programs: $q \sqsubseteq r$ means that r is at least as good as q in the sense that the substitution of r for q in any context is an improvement or will leave things unchanged.

An assertion $[b]$ refines **skip**, as it behaves like miracle if b does not hold.

Law 6 $\textbf{skip} \quad \sqsubseteq \quad [b]$

If a declared variable is never used, its declaration has no effect.

Law 7 If x is not free in c, then $\quad \textbf{var}\ x : T \bullet c\ \textbf{end} \ = \ c$

Assigning to a variable just before the end of its scope is irrelevant.

Law 8 $(\textbf{var}\, x : T \bullet c;\ \ x := e\ \textbf{end}) \ = \ (\textbf{var}\, x : T \bullet\ c\ \textbf{end})$

Invoking a method is insignificant if the affected variables are just before the end of their scope.

Law 9 $\textbf{var}\ y : T_1,\ z : T_2\ \bullet\ p;\ \ y.m(z);\ \ \textbf{end} = \textbf{var}\ y : T_1,\ z : T_2\ \bullet\ p\ \textbf{end}$

This is only a small fragment of the algebraic laws of ROOL. See [1] for a comprehensive set of laws.

3 Interpreter Structure

The RVM is characterized by a normal form, which is an interpreter-like program modeling a cyclic mechanism that executes one bytecode instruction at a time (Figure 1). More specifically, it describes the behavior of our virtual machine executing a stored program, in an iterated execution of a sequence of bytecodes stored in the global variable *Cprog*, which represents the compiled program. Another global variable named *Memory* stores the initial value of the program variables. At the end of execution, the final values are copied back to *Memory*. Therefore, from the point of view of the interpreter execution, the observable data space is *Memory*. This represents the concrete counterpart of the variables of the source program.

As a valid program in ROOL, the normal form consists of a sequence of class declarations (CDS_{RVM}) followed by a main command named I_{RVM}. It is basically a single flat loop, where every cycle fetches the next instruction to be executed and then invokes an associated method to simulate its effect on the internal data structures of the virtual machine.

More specifically, I_{RVM} is a **var** block declaration that introduces three local variables. The main variable is *rvm*; the other two, *op* and c, are auxiliary: they are used to obtain the control information stored in *rvm* in order to guide the execution flow of the interpreter. The *op* variable is an integer whose value is the operation code that indicates the next bytecode instruction to be executed. The c variable is an instance of the *Control* class declared in CDS_{RVM}; its attributes are the value of the *pc* register, *initial* and *final*. The last two attributes denote

```
$CDS_{RVM}$ • var $op, c, rvm$ :   $Int, Control, RoolVM$   •
        $rvm :=$ new $RoolVM$;
        $rvm.SetClasses(Cprog)$; $rvm.SetVariables(Memory)$;
        $rvm.GetControl(c)$;
        while $c.pc \geq c.initial \wedge c.pc < c.final$   •
          $rvm.GetNextBytecode(op)$;
          if $[]_{\langle 0 \leq i \leq n \rangle}\ op = i \rightarrow rvm.inst_i$ fi;
          $rvm.GetControl(c)$;
        end;
        $rvm.GetVariables(Memory)$;
      end
```

Fig. 1. The ROOL Interpreter ($CDS_{RVM} \bullet I_{RVM}$)

the interval comprising the bytecode stream of the current executing method. We assume that the value of *initial* is always 1. The value of *final* depends on the size of the bytecode stream.

The **while** statement is executed until the program counter reaches a value beyond the interval that comprises the current executing bytecode stream. In the body of the while statement, first the operation code (*op*) of the next instruction to be executed is determined and then a conditional statement selects the corresponding method that implements that instruction. The instructions available for the interpreter are implemented as methods defined in the class *RoolVM*. All instructions modify the program counter; even the *initial* and *final* can be modified when instructions that handle method calls are executed. The last statement in the while body is necessary to retrieve the current program counter and the current bytecode interval to be tested in the next while iteration.

As already said, the internal data structure of the interpreter is represented by the local variable *rvm*, an instance of the class *RoolVM*. This structure is basically a sequence of instances of *FrameInfo* implementing a stack of frames. This stack is named F and records the order in which the called methods were invoked. When a method is invoked, the virtual machine creates a new frame onto that stack. When the method completes, the frame is discarded.

An instance of *FrameInfo* (Figure 2) holds the state of one ROOL method invocation. It includes its own *pc* register; its operand stack s; its current class *cl*, which is an instance of *ClassInfo*; its currently executing method *mtd*; and its list of local variables v. When the ROOL interpreter starts running, the *pc* value in the initial frame points to the first bytecode of the main command.

An instance of *ClassInfo* contains the following attributes: *name*, the class name; *super*, its superclass; *mtds*, a reference to the list of methods declared in the class; *fds*, a reference to the list of attributes declared in the class; *cp*, a reference to the constant pool. This last attribute is an heterogeneous list of references, and provides much of the essential information needed by a class. It contains entries for the names of referenced classes, methods and attributes,

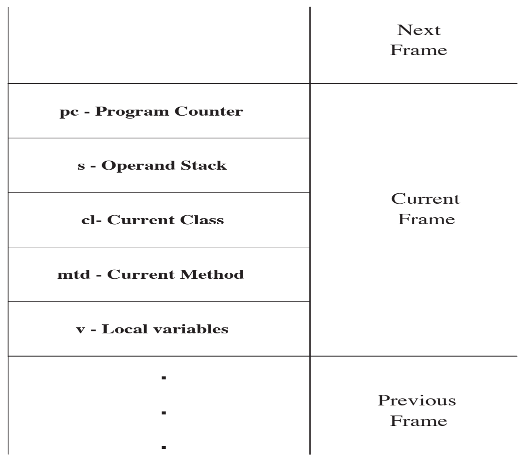

Fig. 2. A frame in the ROOL Virtual Machine

and for the integer constants. From the constant pool, the interpreter can reach instances of *ClassInfo* bound to any class referenced by the current class.

The computation in the virtual machine is centralized on the operand stack (Figure 2). Because the virtual machine has no registers for storing arbitrary values, everything must be pushed onto the stack before it can be used in a calculation.

Most of the data structures used to implement our interpreter employs sequences of objects. We assume the following operators to deal with sequences: $Y \frown X$, the concatenation of sequence Y with sequence Z; *head* y, the leftmost element of sequence Y; *last* Y, the rightmost element of sequence Y; *front* Y, the sequence without the last element of Y; *tail* Y, the sequence without the head element of Y; $\#Y$, the number of elements of Y.

In order to refer to some components of our interpreter, we adopt the following abbreviations.

Definition 2 (Target Machine Components)

$$\begin{array}{ll} S & \overset{def}{=} (last\ rvm.F).s \\ V[n] & \overset{def}{=} (last\ rvm.F).v[n] \\ PC & \overset{def}{=} (last\ rvm.F).pc \\ CP[n] & \overset{def}{=} (last\ rvm.F).cp[n] \end{array}$$

We denote by S the operand stack in the current frame; $V[n]$ designates the local variable whose location is n in the current frame, PC represents the current program counter; finally $CP[n]$ stands for the object stored in the constant pool of the current class, whose entry is given by the index n.

We now introduce abbreviations for update expressions over the operand stack.

Definition 3 (Abbreviations over the operand stack)

$$
\begin{array}{ll}
S \uparrow [e] & \overset{def}{=} (rvm;\ \ F : (front\ F) \frown \langle (last\ F;\ \ S : S\ \frown \langle\, e \,\rangle\,) \rangle) \\
S \downarrow & \overset{def}{=} (rvm;\ \ F : (front\ F) \frown \langle (last\ F;\ \ S : front\ S) \rangle\,) \\
S|| & \overset{def}{=} (last\ rvm.F).(last\ S) \\
S_{uop} & \overset{def}{=} (rvm;\ \ F : (front\ F) \frown \langle (last\ F;\ \ S : (front\ S) \frown \\
 & \quad \langle\, (\mathbf{new}\, DataInt;\ \ Info : \mathbf{uop}\,(last\ S).Info)\,\rangle\,)\rangle) \\
S_{bop} & \overset{def}{=} (rvm;\ \ F : (front\ F) \frown \langle (last\ F;\ \ S : front(front\ S) \frown \\
 & \quad \langle\, (\mathbf{new}\, DataInt;\ \ Info : (last\ (front\ S)).Info\,\mathbf{bop}\,(last\ S).Info)\,\rangle\,)\rangle)
\end{array}
$$

The abbreviation $S \uparrow [e]$ stands for an update expression in which e is pushed onto S. Similarly, $S \downarrow$ represents the update expression in which a value from the top of S is popped; $S||$ depicts a copy of the value on the top of the stack; S_{uop} represents the update expression in which one integer value is popped from the top of the stack, the unary operator **uop** is applied to it, and the result is pushed back onto S. Finally, S_{bop} handles binary operators *bop*.

Below, we give examples of how the instructions of our virtual machine are defined as commands on RVM.

Definition 4 (Instructions definition)

$$
\begin{array}{ll}
nop & \overset{def}{=} PC := PC + 1 \\
ldc\ i & \overset{def}{=} rvm :=\ S \uparrow [C.CP[i]];\ \ PC :=\ PC + 2 \\
load\ n & \overset{def}{=} rvm :=\ S \uparrow [V[n]];\ \ PC :=\ PC + 2 \\
store\ n & \overset{def}{=} V[n] :=\ \ S||;\ \ rvm :=\ S \downarrow;\ \ PC :=\ PC + 2 \\
uop & \overset{def}{=} rvm := S_{uop};\ \ PC := PC + 1 \\
bop & \overset{def}{=} rvm := S_{bop};\ \ PC := PC + 1 \\
goto\ k & \overset{def}{=} PC := PC + k \\
new\ j & \overset{def}{=} \mathbf{var}\ o :\ ObjectInfo\ \ \bullet \\
 & \quad\quad o := \mathbf{new}\, ObjectInfo;\ \ o.create(CP[j]); \\
 & \quad\quad rvm := S \uparrow [o];\ PC :=\ PC + 2 \\
 & \mathbf{end}
\end{array}
$$

The instruction with opcode *nop* (no operation) has no effect, except for the program counter (PC) increment. The instruction with opcode *ldc* (load constant) has one argument (i) and pushes an integer constant onto the operand stack (S). Its argument i follows the opcode *ldc* in the bytecode stream and represents a constant pool index to the location where the constant is stored.

Pushing a local variable onto the operand stack is done by the instruction *load*, and involves moving a value from the local variables list to the operand stack. All local variables are instances of the class **object**, as well as the elements of the operand stack. So they can hold any object reference, including an object that encapsulates values, the integer or any other primitive type. To pop a value of any type from the the stack to a local variable, the virtual machine uses the instruction *store*. The argument n is a index to a local variable.

To deal with operators, we group them so that *uop* and *bop* stand for arbitrary unary and binary operators, respectively. The instruction with *goto* opcode always branches: the offset k can be a positive or negative integer value.

The instruction *new* builds an instance of *ObjectInfo* to hold the RVM's representation of an object whose type is indicated by the argument j. The j is an index to a class entry in the constant pool. The call $o.create(CP[j])$ traverses the representation of the source program class hierarchy, determining and recording in o the attributes of the class indicated by j. The resulting object o is pushed into the operand stack S.

4 Compiling with Theorems

The compilation process consists in reducing by algebraic transformation an arbitrary program to the above normal form. The reduction theorems (stated here as rules), which justify the compilation process, can be proved correct from the basic algebraic laws of ROOL. The correctness of the compiler follows from the correctness of each compilation rule. It is sufficient to show how each primitive command can be written in the normal form and, by structural induction, how each operator of the language, when applied to operands in the normal form, yields a result expressible in the normal form.

The compilation process involves three phases: simplification of expressions, data refinement, and control elimination, in this order. The data refinement converts the abstract space of the source program to the concrete space of the RVM. A permutation in the order of the phases may require to repeat a phase already accomplished.

In this section we give an overview of our approach to compilation. Moreover, we list the compilation rules to give an idea of how we can compile methods, classes, and imperative control structures.

4.1 Simplification of Expressions

The first task of the compilation process is the elimination of nested expressions. The expected outcome of this phase is a program formed of a sequence of assignments where each assignment operates through the operand stack. To do so, we need to refer to the variable *rvm*, which contains the data structure of our interpreter.

Basically, the task of eliminating nested expressions in a source program involves the rewriting of assignments. The outcome is a program involving assignments of the form described by the following patterns. They are closely related to those used to define bytecode instructions of our stack-based machine.

Definition 5 (Patterns for the operand stack)

$$\begin{array}{lll} \mathit{load}_{se}(i) & \overset{def}{=} & \mathit{rvm} := S \uparrow [(\mathbf{new}\mathit{DataInt};\ \mathit{Info} : i)] \\ \mathit{store}_{se}(i) & \overset{def}{=} & i :=\ S||.\mathit{Info};\ \ \mathit{rvm} :=\ S\downarrow \\ \mathit{uop}_{se} & \overset{def}{=} & \mathit{rvm} := S_{uop} \\ \mathit{bop}_{se} & \overset{def}{=} & \mathit{rvm} :=\ S_{bop} \\ \mathit{load}_{o}(o) & \overset{def}{=} & \mathit{rvm} := S \uparrow [o] \\ \mathit{store}_{o}(o) & \overset{def}{=} & o := S||;\ \ \mathit{rvm} :=\ S\downarrow \end{array}$$

The pattern load_{se} pushes an integer value onto the operand stack, whereas store_{se} pops an integer value from the operand stack and assigns it to an integer variable. Since the operand stack consists of a sequence of objects, it is necessary to encapsulate integer values using instances of *DataInt*. The pattens uop_{se} and bop_{se} represent a group of patterns that implement the effect of the unary and binary operators, respectively. The patterns $\mathit{load}_o(o)$ and $\mathit{store}_o(o)$ are similar to those for integer values, except for the type of its arguments: objects, instead of integers or primitive types.

The following rules rely on the context in which they will be applied. We use the form bellow to present the compilation rules.

We recall that we use the notation $\mathit{CDS}_{RVM}, \mathit{rvm} : \mathit{RoolVM}, N \triangleright c \sqsubseteq c'$ to mean that the refinement step $c \sqsubseteq c'$ holds in the context of class declarations CDS_{RVM} and local variable $\mathit{rvm} : \mathit{RoolVM}$. Furthermore, the command c is assumed to be inside the class N which denotes the main command or a class in the sequence of class declaration (CDS) of the source program.

An exhaustive application of the following rules simplifies arbitrarily nested expressions. The next rule deals with the simplification of an assignment to an integer variable. The assignment is refined to a sequence of patterns that operates primarily over the operand stack S.

Rule 1 (Assignment of an integer variable)

$$\mathit{CDS}_{RVM}, \mathit{rvm} : \mathit{RoolVM}, N \triangleright (x := e) \quad \sqsubseteq \quad \mathit{load}_{se}(e);\ \ \mathit{store}_{se}(x);$$

The expression e above may be arbitrarily nested; we need to further simplify it to achieve a simpler form. We must proceed until the resulting expressions consist only of a single variable or constant. For simplicity, we omit the context in the following rules.

The next rule handles a pattern whose argument is an application of binary operators.

Rule 2 (Binary operator) If S does not occur in e or f

$$\mathit{load}_{se}(e\ \mathbf{bop}\ f) \quad \sqsubseteq \quad \mathit{load}_{se}(e);\ \ \mathit{load}_{se}(f);\ \ \mathit{bop}_{se}$$

where bop represents an arbitrary binary operator.

The nested expression in $\mathit{load}_{se}(e\ \mathbf{bop}\ f)$ is replaced with a sequence of patterns which first load e, then load f, and finally perform the **bop** operation.

The boolean expressions appearing in **while** and **if** commands may also be arbitrarily nested, and therefore need to be simplified. The following rule considers the **while** command.

Rule 3 (Condition of **while**) If x does not occur in b nor in p;

$$\begin{array}{lll} \mathbf{while}\ b\ \bullet\ p\ \mathbf{end} & \sqsubseteq & \mathbf{var}\ x : \mathit{boolean}\ \bullet \\ & & \quad x := b; \\ & & \quad \mathbf{while}\ x\ \bullet\ p;\ \ x := b\ \mathbf{end}; \\ & & \mathbf{end} \end{array}$$

The condition becomes a single variable whose value is given by the assignment. The expression b can now be simplified using the rules related to assignment.

Object creation is just another instance of assignment.

Rule 4 (Object Creation - Simplification)
$x\ :=\ \mathbf{new}\ C \quad \sqsubseteq\ \mathit{load}_o(\mathbf{new}\ C);\ \ \mathit{store}_o(x);$

The expression **new** C gives rise to a new object of type C. In this case we need to use pattern $\mathit{load}_o(\mathbf{new}\ C)$ to push this object onto the operand stack. Observe that $\mathit{load}_o(\mathbf{new}\ C)$ does not need to be simplified anymore; the simplification here takes only one step.

4.2 Data Refinement

Data refinement is the replacement of the abstract space of the source program by the concrete state of the target machine. This means that all references to variables, methods, attributes, and classes declared in the source program must be replaced with the corresponding ones in the interpreter.

The function Ψ is the symbol table which maps each variable of the source program to addresses in the local variables sequence, in such a way that $V[\Psi_x]$ holds the value of x. Similarly, we assume that Φ is the class table, which holds all instances of *ClassInfo* corresponding to the compiled class declarations. From these tables, we can build the function $\overline{\Psi\Phi}$ which carries out the data refinement on commands.

The treatment of class declarations is handled by the function $\overline{\varpi}$, which associates a class declaration with an instance of *ClassInfo* recording the declarations of attributes and methods occurring in the class. The representation resulting from the compilation by $\overline{\varpi}$ is incorporated in the class table Φ by the function $\overline{\Psi\Phi}$. The rather lengthy definition of $\overline{\varpi}$ is by the induction on sequences of class declarations, and on the structure of classes.

We present below our main rule which states the correctness of the compilation process for a program $\mathit{CDS}\ \bullet\ c$.

Rule 5 (Compilation Process)
$\mathit{CDS}_{RVM}\ \mathit{CDS}\ \bullet\ \overline{\Psi\Phi}(c) \quad \sqsubseteq\ \mathit{CDS}_{RVM} \bullet\ \overline{\Psi\Phi \cup\ \overline{\varpi}(\mathit{CDS})}(c)$

Since the above rule is a refinement between programs, it is valid in any context. The source program $CDS \bullet c$ operates on a data space different from the data space of our normal form. Therefore, it does not make sense to compare them directly: $\overline{\Psi\Phi}$ performs the necessary change of data representation. Observe that $\Phi \cup \varpi(CDS)$ incorporates the compiled class declarations CDS in Φ. The symbol table Ψ includes the global variables and is extended by local variables declarations.

The function $\overline{\Psi\Phi}$ acts only over the class declarations belonging to the source program (CDS). RVM's class declarations (CDS_{RVM}) are not reduced. Similarly, the function $\overline{\Psi\Phi}$ does not affect the variables declared in our interpreter.

In order to carry out the change of class references and data representation in a systematic way, we need to use the distributivity properties of the function $\overline{\Psi\Phi}$. Commands which have no reference to variables or classes are not affected by the function $\overline{\Psi\Phi}$. For instance, the **skip** command is not affected.

Rule 6 $\quad \overline{\Psi\Phi}(\mathbf{skip}) \quad \sqsubseteq \quad \mathbf{skip}$

In the following, we introduce simple patterns over the operand stack introduced by the data refinement phase.

Definition 6 (Patterns introduced by the data refinement phase)

$$\begin{array}{lll}
load_{dr}(\Psi_x) & \overset{def}{=} & rvm := S \uparrow [V[\Psi_x]] \\
store_{dr}(\Psi_x) & \overset{def}{=} & V[\Psi_x] := S||; \;\; rvm := S \downarrow \\
ldc_{dr}(\Phi_a) & \overset{def}{=} & rvm := S \uparrow [C.CP[\Phi_a]] \\
new_o(\Phi_C) & \overset{def}{=} & \mathbf{var}\ o : ObjectInfo \;\; \bullet \\
 & & \quad o := \mathbf{new}\, ObjectInfo; \;\; o.create(CP[\Phi_C]); \\
 & & \quad rvm := S \uparrow [o]; \\
 & & \mathbf{end}
\end{array}$$

These patterns have no reference to variables or constants of the source program. The pattern $load_{dr}(\Psi_x)$ pushes onto the stack S the local variable whose location is given by Ψ_x. Similarly, the method $store_{dr}(\Psi_x)$ pops a value from S and stores it in the local variable indicated by the index Ψ_x. The pattern $ldc_{dr}(\Phi_a)$ pushes the constant a onto S; Φ_a indicates the location of the object holding a in the constant pool (CP) of the current class. The pattern $new_o(\Phi_C)$ creates an object which is a representation of an instance of the class C; the new object is pushed onto S; Φ_C indicates the location in the constant pool of an instance of *ClassInfo* which represents the class C. The patterns above have the same behavior as the instructions presented in the Definition 4, except for the absence of the PC increment.

The next rules deal with patterns possibly introduced in the previous phase. The rule below shows how we deal with object creation.

Rule 7 (Object Creation - Data Refinement)

$\overline{\Psi\Phi}(load_o(\mathbf{new}\ C)) \quad \sqsubseteq \quad new_o(\Phi_C)$

After the simplification phase, all object creations are refined to a pattern $load_o(\mathbf{new}\ C)$, which retains a reference to C, a class of the source program. The function $\overline{\Psi\Phi}$ eliminates this reference, introducing another pattern whose parameter is an index Φ_C to an entry in the constant pool CP, corresponding to the type C.

The following rule deals with the loading of an integer variable.

Rule 8 (Load Integer - Data Refinement)
$\overline{\Psi\Phi}(load_{se}(x)) \quad \sqsubseteq \quad load_{dr}(\Psi_x)$

The pattern $load_{dr}(\Psi_x)$ uses Ψ_x to refer to location that holds the value of x in V. In other words, $V[\Psi_x]$ is an object holding the value of x.

The next rule shows the effect of $\overline{\Psi\Phi}$ over the pattern $store_{se}(x)$.

Rule 9 (store Integer - Data Refinement)
$\overline{\Psi\Phi}(store_{se}(x)) \quad \sqsubseteq \quad store_{dr}(\Psi_x)$

Again $\overline{\Psi\Phi}$ removes the source reference to the variable x. The Ψ_x refers to the location allocated in V to store the object popped from the operand stack.

The next rule addresses the pattern used to push a constant onto S.

Rule 10 (Load Constant - Data Refinement)
$\overline{\Psi\Phi}(load_{se}(a)) \quad \sqsubseteq \quad ldc_{dr}(\Phi_a)$

When $\overline{\Psi\Phi}$ is applied, the constant a is placed in an entry of the constant pool whose location is indicated by Φ_a.

The following rule deals with the pattern $store_o(o)$.

Rule 11 (Store object - Data Refinement)
$\overline{\Psi\Phi}(store_o(o)) \quad \sqsubseteq \quad store_{dr}(\Psi_o)$

Note that $\overline{\Psi\Phi}$ eliminates the source reference to the variable o. In this case, Ψ_o has the same role of Ψ_x in the last rule.

In the case of sequential composition of commands p and q, $\overline{\Psi\Phi}$ distributes over each command.

Rule 12 (Sequential Composition)
$\overline{\Psi\Phi}(p;\ q) \quad \sqsubseteq \quad \overline{\Psi\Phi}(p);\ \overline{\Psi\Phi}(q)$

In the next section we tackle the control elimination phase of the compilation process.

4.3 Control Elimination

Control elimination consists of reducing the nested control structure of the source program to a single flat iteration. The outcome is a program in our normal form.

Recall that each frame instance of our interpreter is equipped with the program counter PC used for scheduling and selection of bytecode instructions. More precisely, PC is the pointer which indicates the location in the bytecode stream of the next instruction to be executed. The addresses of the bytecodes is implicit in the sequence we generate here. We introduce the following abbreviation.

Definition 7 (Compiled Program)

$$\mathit{Cprog}\ (\varrho, \beta, s) = (\mathbf{new}\ \mathit{ClassInfo};\ \ \mathit{CP} : \varrho_\beta;\ \ \mathit{mtds} : \langle(\mathbf{new}\ \mathit{MethodInfo};\ \ \mathit{bytecode} : \beta;\ \ \mathit{size} : s)\rangle)$$

This is an instance of *ClassInfo*, which contains only one *MethodInfo* that holds the bytecode stream β. The parameter s is the size of the bytecode stream. Moreover, there is the symbol table ϱ which is the constant pool possibly referenced by the bytecode instructions in β. We also use $I_{RVM}^{Cprog(\varrho,\beta,s)}$ to refer to the ROOL interpreter executing the compiled program stored in $\mathit{Cprog}(\varrho, \beta, s)$.

The reduction of **skip** states that its only effect is the PC increment.

Rule 13 (Skip)

$$\mathbf{skip} \ \sqsubseteq\ I_{RVM}^{Cprog(\varrho,\beta,s)}$$
$$\mathit{where}\ \beta\ =\ [\mathit{nop}],\ \ s = 1,\ \ \mathit{and}\ \varrho\ = \varnothing$$

The above β holds the bytecode stream containing the singleton operation *nop*. To illustrate the proofs of the compilation rules, the proof of the above rule is presented in Section 4.6.

The next rules deal with patterns introduced in the previous phases. The following rule considers the pattern that stores an integer variable.

Rule 14 (Store Integer)

$$\mathit{store}_{dr}(\Psi_x) \ \sqsubseteq\ I_{RVM}^{Cprog(\varrho,\beta,s)}$$
$$\mathit{where}\ \beta\ =\ [\mathit{store}, \Psi_x],\ \ s = 2,\ \ \mathit{and}\ \varrho\ = \varnothing$$

The first bytecode in β is the instruction *store*, followed by the index Ψ_x, representing the location of x in V.

The following rule deals with object creation.

Rule 15 (Object Creation)

$$\mathit{new}_{dr}(\Phi_C) \ \sqsubseteq\ I_{RVM}^{Cprog(\varrho,\beta,s)}$$
$$\mathit{where}\ \beta\ =\ [\mathit{new}, \Phi_C],\ \ s = 2,\ \ \mathit{and}\ \varrho\ = \{\Phi_C \mapsto \Phi.C\}$$

The first bytecode is the instruction *new*, followed by the argument, the index in the constant pool of the class.

The reduction of sequential composition assumes that both arguments are already in the normal form. The resulting normal form combines the original bytecode streams.

Rule 16 (Sequential composition — $p;\ q$)

$$I_{RVM}^{Cprog(\varrho_p,\beta_p,s_p)};\ I_{RVM}^{Cprog(\varrho_q,\beta_q,s_q)} \ \sqsubseteq\ I_{RVM}^{Cprog(\varrho,\beta,s)}$$
$$\mathit{where}\ \beta = \beta_p \frown \beta_q,\ \ s = s_p + s_q,\ \ \mathit{and}\ \varrho = \varrho_p\ \cup \varrho_q$$

We concatenate the bytecode streams of p and q given by β_p and β_q, respectively. We also join the symbol tables ($\varrho_p\ \cup \varrho_q$).

In the next section, two examples illustrate how the rules presented so far can be used effectively to carry out the compilation process.

4.4 The Approach through an Example

To illustrate the compilation process, we present the following example.

Example 1. Consider the simplification of the assignment $\overline{\Psi\Phi}(x := \mathbf{new}\, C)$, where x is a variable of type C.

Using Rule 4, we can transform the above assignment into

$$\sqsubseteq\ \overline{\Psi\Phi}(\mathit{load}_o(\mathbf{new}\, C);\ \ \mathit{store}_o(x));$$

At this point, observe that we have a hybrid data space including the abstract space of the source program and the concrete state of the target machine. We need to replace the source references by the corresponding ones in the RVM. Using the distributive properties of the simulation function $\overline{\Psi\Phi}$ results in

$$\sqsubseteq\ \overline{\Psi\Phi}(\mathit{load}_o(\mathbf{new}\, C));\ \ \overline{\Psi\Phi}(\mathit{store}_o(x));$$

Applying Rules 7 and 11, it becomes

$$\sqsubseteq\ \mathit{new}_o(\Phi_C);\ \ \mathit{store}_{dr}(\Psi_x);$$

Applying the Rule 15 to the pattern $\mathit{new}_o(\Phi_C)$ we obtain

$$\mathit{new}_o(\Phi_C)\quad \sqsubseteq\ I_{RVM}^{Cprog(\varrho_1,[new,\Phi_C],2)}$$

The remaining pattern $\mathit{store}_{dr}(\Psi_x)$ is then transformed by Rule 14, yielding

$$\mathit{store}_{dr}(\Psi_x)\quad \sqsubseteq\ I_{RVM}^{Cprog(\varrho_2,[store,\Psi_x],2)}$$

Using the Rule 16 for sequential composition, we combine these simple forms.

$$\sqsubseteq\ I_{RVM}^{Cprog(\varrho_1\cup\varrho_2,[new,\Phi_C,store,\Psi_x],4)}$$

The resulting *Cprog* has 2 instructions in its bytecode stream. The first creates an object based on the RVM's internal representation of class C. The second stores this object in the local variable list, in the location corresponding to the variable x.

□

4.5 Lemmas

The basic rules of ROOL and the following lemmas are necessary to prove the compilation rules above. The introduction of assertions reveals the effect of the execution of a command, making explicit the the contents of the internal data structure of our ROOL interpreter.

For convenience, we use $\ll PC : n;\ cl : p;\ mtd : m;\ V : l \gg$ as an abbreviation for the following update expression:

$$(\textbf{new}\ \mathit{RoolVM};\ (\mathit{last}\ F) : (\textbf{new}\ \mathit{FrameInfo};\ \ PC : n;\ \ cl : p;\ \ mtd : m;\ \ \ V : l))$$

The above update expression denotes an object of class *RoolVM* whose top frame is an instance of *FrameInfo*; its program count PC has the value n; p represents the current class cl; m depicts the current executing method mtd; l is the list of local variables V.

The following lemma is related to the initialization of the *rvm* variable with the compiled program stored in *Cprog*.

Lemma 1 (Initializing the local variable *rvm*)
$rvm := \textbf{new}\ \mathit{RoolVM};\ \ rvm.\mathit{SetClasses}(\mathit{Cprog})$
$=$
$rvm := \textbf{new}\ \mathit{RoolVM};\ \ rvm.\mathit{SetClasses}(\mathit{Cprog});$
$[rvm =\ll PC : 1;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{null} \gg]$

The assertion introduced here makes explicit the value of the attributes in the *rvm*'s initial frame.

The next lemma shows the effect of the method call *rvm.SetVariables(Memory).*

Lemma 2 (Loading Initial Memory)
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{null} \gg];$
$rvm.\mathit{SetVariables}(\mathit{Memory})$
$=$
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{null} \gg];$
$rvm.\mathit{SetVariables}(\mathit{Memory});$
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{Memory} \gg]$

The last assertion shows the value of the list of local variables (V) as a copy of the global variable *Memory*.

The following lemma makes explicit the effect of the method call *rvm.GetControl(c)* over the auxiliary variable *c*.

Lemma 3 (Loading Control)
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{Memory} \gg];$
$rvm.\mathit{GetControl}(c)$
$=$
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{Memory} \gg];$
$rvm.\mathit{GetControl}(c);$
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{Memory} \gg];$
$[c.pc = i;\ c.\mathit{Initial} = v_1;\ c.\mathit{Final} = v_2;\ ]$
Where $v_1 = 1$ *and* $v_2 = (v_1 + (\mathit{head}\ \mathit{Cprog.mtds}).\mathit{Size})$.

The assertion introduced here shows that the value of the variable c is updated with the current program counter, and the interval denoted by v_1 and v_2.

The next lemma makes clear the value of the operation code of the next instruction to be executed.

Lemma 4 (Getting Operation Code)
$[rvm =\ll PC : i;\ cl : \mathit{Cprog};\ mtd : (\mathit{head}\ \mathit{Cprog.mtds});\ V : \mathit{Memory} \gg];$

$rvm.GetNextBytecode(op)$
$=$
$[rvm = \ll PC : i;\ cl : Cprog;\ mtd : (head\ Cprog.mtds);\ V : Memory \gg];$
$rvm.GetNextBytecode(op);$
$[rvm = \ll PC : i;\ cl : Cprog;\ mtd : (head\ Cprog.mtds);\ V : Memory \gg];\ \ [op = n]$
Where n denotes the operation code of the next instruction to be executed.

The last assertion makes explicit the value of the opcode.

The following lemma shows the effect of the *nop* instruction.

Lemma 5 (*Nop Effect*)
$[rvm = \ll PC : i;\ cl : Cprog;\ mtd : (head\ Cprog.mtds);\ V : Memory \gg];\ \ rvm.nop$
$=$
$[rvm = \ll PC : i;\ cl : Cprog;\ mtd : (head\ Cprog.mtds);\ V : Memory \gg];\ \ rvm.nop;$
$[rvm = \ll PC : i+1;\ Cl : Cprog;\ mtd : (head\ Cprog.mtds);\ V : Memory \gg]$

When *rvm.nop* is executed, the program counter is incremented by 1.

The next lemma replaces the method call *rvm.GetVariables(Memory)* by an assignment corresponding to its effect.

Lemma 6 (*Getvariables Effect*)
$[rvm = \ll PC : i;\ cl : Cprog;\ mtd : (head\ Cprog.mtds);\ V : M \gg];$
$rvm.GetVariables(Memory)$
$=$
$Memory := M$

Memory receives the list of local variables stored in the attribute *V*.

4.6 Proof Example

As an example of a proof of a compilation rule, we show how the lemmas and the basic laws are used to prove the Rule 13. We start from $I_{RVM}^{Cprog(\varnothing,nop,1)}$, the right hand side (RHS) of the rule inequation. First, the Lemma 1 is applied to show the effect of initializing the variable *rvm* with the compiled program stored in *Cprog*. Then, the Lemma 2 is applied to evidence the result of copying the global variable *Memory* into the *rvm*'s current frame. After that, the Lemma 3 shows the values of the variables used in the **while** condition. Since the assertion before the **while** implies the satisfaction of the **while** condition, it can be unfold. Then, the Lemma 4 can be used to make explicit the value of the operation code of the current executing instruction.

RHS = {Lemma 1} {Lemma 2} {Lemma 3} {Law 2.5} {Lemma 4.1.4}

```
var $op,\ c,\ rvm:\ \ Int,\ Control,\ RoolVM\ \ \bullet$
    $rvm := \textbf{new}\ RoolVM;\ rvm.SetClasses(Cprog);$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:null \gg];$
    $rvm.SetVariables(Memory);$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $rvm.GetControl(c);\ [c.pc = 1;\ c.Initial = 1;\ c.Final = 2;\ ];$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $rvm.GetNextBytecode(op);$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $[op = 0];$
    if $[]_{\langle 0\leq\ i\ \leq\ n\rangle}\ op = i \rightarrow rvm.inst_i$ fi $rvm.GetControl(c);$
    while $c.pc \geq c.Initial \wedge c.pc < c.Final \bullet$
        $rvm.GetNextBytecode(op);$ if $[]_{\langle 0\leq\ i\ \leq\ n\rangle}\ op = i \rightarrow rvm.inst_i$ fi; $rvm.GetControl(c);$
    end;
    $rvm.GetVariables(Memory);$
end
```

The assertion introduced just before the first **if** statement allows the selection of one command (Law 3); in this case, *rvm.nop*. Then, in order to show the effect of the *rvm.nop*, the Lemma 5 is used. Then, the Lemma 3 makes explicit the effect of the *rvm.GetControl(c)*, by showing the updated value of the variable c. At this point, the assertion before the **while** does not satisfy the **while** condition. Using the Law 4, we can eliminate the **while**. The Lemma 6 shows that the effect of the method call *rvm.GetVariables(Memory)* is just the useless assignment of the global variable *Memory* to itself, which can be eliminated (Law 2).

= {Law 3} {Lemma 5} {Lemma 3} {Law 4} {Lemma 6} {Law 2}

```
var $op,\ c,\ rvm:\ \ Int,\ Control,\ RoolVM\ \ \bullet$
    $rvm := newRoolVM;\ rvm.SetClasses(Cprog);$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:null \gg];$
    $rvm.SetVariables(Memory);$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $rvm.GetControl(c);\ [c.pc = 1;\ c.Initial = 1;\ c.Final = 2;\ ];$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $rvm.GetNextBytecode(op);$
    $[rvm =\ll PC:1;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $rvm.nop;$
    $[rvm =\ll PC:2;\ cl:Cprog;\ mtd:(head\ cprog.mtds);\ V:Memory \gg];$
    $rvm.GetControl(c);$
end
```

Repeatedly applying the following laws we can eliminate every command that appears just before the end of the **var** block.

$\sqsupseteq$ {Law 6} {Law 8} {Law 9} **var** $op,\ c,\ rvm:\ Int,\ Control,\ RoolVM\ \bullet$ **end**
Using the Law 7 we can eliminate an empty **var** block.

= {Law 7} **skip**
This concludes our proof.

5 Conclusions

We have presented a framework that can be used to build a correct compiler for a Java-like language extending the approach described in [7]. Both the source (ROOL) and the target (a subset of JVM bytecodes) languages adopted here are far more complex than those described in [7].

We illustrate how compilation is carried out using compilation rules in order to generate bytecodes for an RVM, which can be viewed as a sequential subset of JVM. Each transformation performed by the compilation rules brings the source program closer to our particular normal form, from which we can capture the sequence of generated bytecodes of the target machine.

Our strategy of proof based on assertions allows us to keep the object-oriented design of the interpreter. We carry out the proof of the rules at a more abstract level, without needing to expand the definition of the interpreter and its associated methods. This is necessary only when proving the lemmas.

In the literature, one can find several approaches related to the design of correct compilers. The majority deals with procedural languages, as the algebraic approach described in [5]. In [2], using Abstract State Machines (ASMs), a compilation scheme of Java programs to JVM code is presented; this is a case study for mechanical verification of a compiler correctness proof. Recently, in [8] a description of the ASM models of Java and JVM are given and properties of JVM verification and execution of compiled Java programs are proved. The approach is based on verification instead of on calculation, as here.

A further topic for investigation is the mechanization of our approach. Due to its algebraic nature, a term rewrite system can be used to verify the compilation rules (reduction theorems). Furthermore, the compilation rules can be taken as rewrite rules to carry out compilation automatically. In this way, a prototype compiler can be obtained as a by-product of its own proof of correctness.

Acknowledgments

The research reported in this paper benefitted from discussions with our collaborators David Naumann and Paulo Borba. The authors are partly supported by CAPES and CNPq, grants 520763/98-0 (Ana Cavalcanti), 521039/95-9 (Augusto Sampaio), 680032/99-1 (CO-OP project, jointly funded by PROTEM-CC and the National Science Foundation), and PICDT/UFBa (Adolfo Duran).

References

1. Paulo Borba and Augusto Sampaio. Basic laws of rool: an object-oriented language. *Revista de Informática Teórica e Aplicada*, 7(I):49–68, 2000.
2. E. Börger and W. Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *MFCS'98.*, number 1450, pages 17–35. Springer LNCS, 1998.

3. Ana Cavalcanti and David Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Enginnering*, 26(08):713–728, 2000.
4. C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
5. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS*. Springer-Verlag, Heidelberg, Germany, 1997.
6. W. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
7. Augusto Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.
8. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.
9. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
10. J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, (15):223–249, 1981.