A Refinement Strategy for the Compilation of Classes, Inheritance, and Dynamic Binding

Adolfo Duran[1], Ana Cavalcanti[1] and Augusto Sampaio[2]

[1] Computing Laboratory / University of Kent
Canterbury CT2 7NF United Kingdom
{aad2,A.L.C.Cavalcanti}@ukc.ac.uk
[2] Centro de Informática / Universidade Federal de Pernambuco
Po Box 7851 50740-540 Recife PE Brazil
acas@cin.ufpe.br

Abstract. This paper presents a refinement strategy for the compilation of a subset of Java that includes classes, inheritance, dynamic binding, visibility control, and recursion. We tackle the problem of compiler correctness by reducing the task of compilation to that of program refinement. More specifically, refinement laws are used as compilation rules to reduce the source program to a normal form that models an interpreter running the target code. The compilation process is formalized within a single and uniform semantic framework, where translations or comparisons between semantics are avoided. Each compilation rule can be proved correct with respect to the algebraic laws of the language.

1 Introduction

The concern with correctness of computer programs has increased due to their use in applications where failure could result in unacceptable losses. Correct compilers address the translation of source programs into a target language, with a guarantee that the semantics is preserved. Several approaches have been suggested; the majority focus on procedural languages [12, 14, 17].

Even though object-oriented programming has become very popular in recent years, most applications are developed using informal methods. Approaches to compiler correctness covering object-oriented features are rare in the literature. Recently, there have been works based on verification, focused on the translation between Java programs and the Java Virtual Machine (JVM) [3, 16].

Here, we describe an algebraic approach to construct a provably correct compiler for an object-oriented language called ROOL (for Refinement Object-oriented Language), which is based on a subset of sequential Java. This language includes classes, inheritance, dynamic binding, recursion, type casts and tests, and class-based visibility. It also includes specification constructors like those of Morgan's refinement calculus [13]. Our language was devised as a means to study refinement of object-oriented programs in general, not only compilation. Its weakest precondition and its algebraic semantics have been studied in [4, 2]. In [5], it is used to formalise refactoring as a refinement activity.

Our approach to compilation is inspired on that first described in [10], and further developed for imperative programs in [15]. Its main advantage is the characterisation of the compilation process within a uniform framework, where translations are avoided. Compilation is identified with the reduction of a source program, written in an executable subset of the language, to a normal form.

Our normal form is an interpreter-like program which emulates the behavior of the target machine. From this interpreter we capture the sequence of generated instructions. We define a ROOL Virtual Machine (RVM) as our target. It is based on the Java Virtual Machine (JVM)[11].

The purpose of this paper is to provide a description of the compilation process, and an example. We first presented our approach in [7]; there, however, we do not consider classes, inheritance, recursive method calls, and dynamic binding. In this paper, we consider the compilation of all these constructs; the rules that justify in detail all our reduction steps can be found in [8]. We also present an improved normal form, which provides the same functionality as that in [7], but is more amenable to proof.

The remainder of this paper is structured as follows. In Section 2 we describe ROOL and introduce our case study. In Section 3, we describe the target machine and our normal form. In Section 4, we describe and illustrate the compilation phases and the transformations they impose on our example. Finally, in Section 5, we outline our conclusions and discuss related and future work.

2 ROOL

A program in ROOL consists of a sequence *cds* of Java-like class declarations followed by a main command c, which may contain objects of classes declared in *cds*. A class declaration has the following form.

class N_1 **extends** N_2
  {**pri** $x_1 : T_1$; }*{**prot** $x_2 : T_2$; }*{**pub** $x_3 : T_3$; }* *//attributes*
  {**meth** $m \triangleq (pds \bullet c)$ **end**}* *//methods*
  {**new** $\triangleq (pds \bullet c)$ **end**}* *//Initialiser*
end

First of all, the class name N_1 is introduced. The clause **extends** determines the immediate superclass of N_1; if it is omitted, the built-in empty class **object** is regarded as the superclass. A class can be recursive in that attributes and method parameters of a class N, can have type N itself. Attributes are declared with visibility modifiers similar to those of Java: **pri**, **prot**, and **pub** are used for private, protected, and public attributes.

The clauses **meth** declare methods, which are regarded as public. Methods are seen as parametrised commands in the style of Back [1]; the declaration *pds* of the parameters of a method is separated from its body c by the symbol '•'. The declaration of a parameter x of type T can take the form **val** $x : T$ or **res** $x : T$, corresponding to the traditional conventions of parameter passing known as call-by-value and call-by-result. The **new** clause declares initialisers: methods that are called after creating an object of the class.

In Figure 1, we give an example of an executable program in ROOL that will be used to illustrate our compilation strategy. It simulates a mechanism to keep track of a robot's path. The robot starts in the position $(0, 0)$. Every time it moves, a step of length l is taken towards *north*, *south*, *east*, or *west*. The outcome of this program is the total length of the route described by the robot.

```
class Step
      pri dir, len : int;
      meth setDirection ≙ (val d : Int; • self.dir := d) end
      meth setLength ≙ (val l : Int; • self.len := l) end
      meth getLength ≙ (res l : Int; • l := self.len) end
end
class Path extends Step
      pri previous : Path;
      meth addStep ≙ (val d, l : Int •
         self.previous := self; self.setDirection(d); self.setLength(l);
       ) end
      meth getLength ≙ (res l : Int •
        var aux : Int •
            if (self.previous <> null) → self.previous.getLength(aux)
             [] (self.previous = null) → aux := 0
            fi;
            super.getLength(l); l := l + aux;
        end
       ) end
      new ≙ (val d, l : Int •
        self.setDirection(d); self.setLength(l); self.previous := null
       ) end
end
•     var p : Path •
        p := new Path(north, l0); p.aadStep(north, l1); p.aadStep(east, l2);
        p.aadStep(south, l3); p.aadStep(west, l4); p.getLength(out);
      end
```

Fig. 1. ROOL program for keeping tracking of a robot's path

Two classes are declared in our example, *Step* and *Path*. The first one has two integer attributes: *dir* and *len*, corresponding to the direction and length of a step. The values of these attributes can be set using the methods *setDirection* and *setLength*, whereas the length of a step can be retrieved using the method *getLength*.

The class *Path* extends *Step*, introducing the attribute *previous* to hold the preceding steps that outline the robot's path; *Path* is a recursive class. The method *addStep* introduces a step in the path; it first assigns the current path (**self**) to *previous*, and then invokes the two methods *setDirection* and *setLength* to record the current step. The length of a path is calculated by the method *getLength*, a recursive redefinition of the method with same name declared in

Step. Each recursive invocation of *getLength* visits a step in the path; it traverses the list of steps accumulating the length. The sequence of nested invocations ends when the first step is reached: the value of *previous* is **null**. To get the length of the current step, we use a method call **super**.*getLength* to guarantee that the method declared in *Step*, which is *Path*'s superclass, is invoked.

A method call $le.m$ is as a parametrised command; m refers to the method associated with the object that is the current value of the expression le. In addition to method calls, the main command, the body of methods, and initialisers are defined using imperative constructs similar to those of Morgan's refinement calculus [13]. They are described in the following grammar.

$c \in Com ::= le := e \mid x : [pre, post]$		assignment, specification
	$\mid c_1 ; c_2$	sequential composition
	$\mid pc(e)$	parametrised command application
	$\mid$ **if** $[]_i \bullet b_i \rightarrow c_i$ **fi**	alternation
	$\mid$ **rec** $X \bullet c$ **end** $\mid X$	recursion, recursive call
	$\mid$ **var** $x : T \bullet c$ **end**	local variable block
	$\mid$ **avar** $x : T \bullet c$ **end**	angelic variable block

Left-expressions le are those that can appear as target of assignments and method calls, and as result and value-result arguments. An assignment has the form $le := e$, where e is an arbitrary expression; the semantics is of copy rather than reference. 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.

A parametrised command can have the form **val** $x : T \bullet c$ or **res** $x : T \bullet c$. The alternation is composed by a collection of guarded command $b_i \rightarrow c_i$, as in Dijkstra's language [6]. The block **rec** $X \bullet c$ **end** introduces a recursive command named X with body c; occurrences of X in c are recursive calls. The difference between the two kinds of variable blocks is the way in which the variables are initialised; in a **var** block, the initial value is arbitrary, whereas in a **avar** block, it is angelically chosen. The variables introduced in a **avar** block are angelic variables, also known as logical constants. ROOL does not include a **while** statement, but it can be defined in the standard way using recursion.

The following grammar describes the ROOL expressions, which are mainly those normally found in an object-oriented language.

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

The reference **self** corresponds to *this* of Java. The expression **new** N creates

an object of class N. A type test allows to check the dynamic type of an object, like the *instanceof* in Java. An update expression $(e_1;\ \ x : e_2)$ denotes a fresh object copied from e_1, but with attribute x mapped to a copy of e_2.

3 Target Machine

In our approach to compilation, the behavior of the target machine is given by an interpreter written in ROOL itself. Each executable command is translated to a sequence of bytecode instructions encoded as ROOL commands. The ROOL interpreter models a cyclic mechanism which executes one instruction at a time.

The interpreter represents the target machine components using the variables PC (program counter), S (the operand stack), F (the frame stack), M (store for variables), *Cls* (symbol table for the classes declared in the source program), and CP (constant pool). The execution of an instruction changes these components, updating the machine state. The PC register contains the address of the bytecode instruction currently being executed. Because the virtual machine has no registers for storing temporary values, everything must be pushed onto the operand stack S before it can be used in a calculation. The frame stack F is composed by objects of a class *FrameInfo* that hold the state of execution of a method. When a method is invoked, a new frame is placed on the top of the frame stack; when the method completes, the frame is discarded.

The observable data space of our interpreter is a store for variables M: the concrete counterpart of the variables of the source program. It is a map from the address of the variables to their values; the model is a sequence of objects. Each method invocation frame possesses a store for variables, which records the values of the variables visible during the invocation.

The component *Cls* is a symbol table holding the essential information about the class declarations in the source program; it is represented by a sequence of objects of the class *ClassInfo*. Each of them has the following attributes: *name*, which records the class name; *super*, the superclass name; *subcls*, the sequence of immediate subclasses names; *mtds*, the sequence of methods (objects of a class of *MethodInfo*) declared in the class; and *attrib*, the sequence of attributes (objects of a class of *AttribInfo*) declared in the class. An object of *MethodInfo* holds the following attributes: *name*, which records the method name; and *descriptor*, the list of the type and mechanism used to pass each of the parameters. Attributes are recorded using objects of the class *AttribInfo*; their attributes are also *name* and *descriptor*; the latter contains the type and visibility of the attribute.

The constant pool is a heterogeneous list of references to classes, attributes, methods, and constants; these references are entries obtained through an index. Entries are objects of four different classes. A class entry is an object of a class *CpEntryClass*, which contains only a class name; from that class we can obtain the corresponding object of *ClassInfo* in *Cls*. An attribute entry contains an object of *CpEntryAttrib*, which holds an attribute name and the class where the attribute is defined. A method entry holds an object of *CpEntryMtd* containing the method name and the name of the class where the method is defined. Finally,

an entry containing an integer constant is an object of a class *DataInfoInt*; integer values are encapsulated in objects.

3.1 Normal Form

Our normal form is an interpreter-like program modelling a cyclic mechanism that executes one instruction at a time. Every cycle fetches the next instruction to be executed and simulates its effect on the internal data structures of the interpreter. The normal form consists of a sequence of class declarations (cds_{RVM}) followed by a main command named I, as shown in Figure 2. The class declarations define the classes mentioned in the description of the *RVM* components. The main command describes the behavior of our target machine executing a compiled program: an iterated execution of a sequence of bytecodes represented by the set of guarded commands *GCS*.

The main command is a **var** block declaration that introduces three local variables, PC, S, and F: the program counter, the operand stack, and the frame stack. The first two commands in the variable block are assignments to create a new operand stack and a new frame stack.

$$
\begin{array}{l}
cds_{RVM} \bullet \textbf{var } PC, S, F : Int, Stack, FrameStack \quad \bullet \\
\qquad S := \textbf{new } Stack;\ \ F := \textbf{new } FrameSTack; \\
\qquad PC := s; \\
\qquad \textbf{while } PC \ \geq\ s \wedge PC \ <\ f \ \rightarrow\ \textbf{if } GCS \textbf{ fi end} \\
\quad \textbf{end}
\end{array}
$$

Fig. 2. The ROOL Interpreter ($cds_{RVM} \bullet I$)

The variable PC is used for scheduling the selection and sequencing of instructions. The abbreviation *GCS* depicts the stored program as a set of guarded commands of the form $(PC = k) \rightarrow q$, where q is a machine instruction that is executed when PC is k. The initial value of PC is the address s of the first instruction to be executed; the final address is f. The **while** statement is executed until PC reaches a value beyond the interval determined by s and f. The body of the while tests the PC value and selects the instruction to be executed. All instructions modify PC. The set of guarded commands is an abstract representation of the target code. The design of a compiler in our approach is actually an abstract design of a code generator.

For convenience, we define some abbreviations.

Definition 1 (Abbreviations for the interpreter)

$$
\begin{array}{ll}
\mathit{Init} & \stackrel{def}{=} S := \textbf{new } Stack;\ \ F := \textbf{new } FrameStack; \\
v : [s, GCS, f] & \stackrel{def}{=} PC := s; \\
 & \qquad \textbf{while } PC \ \geq\ s \wedge PC \ <\ f \ \rightarrow\ \textbf{if } GCS \textbf{ fi end}
\end{array}
$$

where v is the list PC, S, F.

Together, *Init* and $v : [s, GCS, f]$ compose the body of variable block in the main command.

3.2 Machine Instructions

In this section, we give some examples of how the instructions of our virtual machine are defined. We assume that n stands for an address in M, and k for an address in the sequence of bytecodes.

Definition 2 (Instructions definition)

$$
\begin{array}{lll}
load(n) & \stackrel{def}{=} & S := \langle M[n] \rangle \frown S; \quad PC := PC + 2 \\
bop & \stackrel{def}{=} & S := \langle head(tail(S)) \, \mathbf{bop} \, head(S) \rangle \frown tail(tail(S)); \quad PC := PC + 1 \\
goto(k) & \stackrel{def}{=} & PC := k \\
new(j) & \stackrel{def}{=} & \mathbf{var} \; o : \; ObjectInfo \quad \bullet \\
 & & \quad o := \mathbf{new} \, ObjectInfo; \quad o.create(Cls, CP, j); \\
 & & \quad S := \langle o \rangle \frown S; \; ; \; PC := PC + 2 \\
 & & \mathbf{end}
\end{array}
$$

Pushing a local variable onto the operand stack is done by the instruction *load*, and involves moving a value from the store of variable M to the operand stack S. To deal with operators, we group them so that *bop* stand for binary operators. The instruction *goto* always branches: the integer argument k is assigned to PC. The instruction *new* builds an object of class *ObjectInfo* to hold the representation of an object whose type is indicated by the argument j. This is an index to a class entry in the constant pool. The call $o.create(Cls, CP, j)$ traverses *Cls*, the representation of the source program class hierarchy, determining and recording in o the attributes of the class indicated by j, and of its superclasses. The resulting object o is pushed into the operand stack S.

4 Strategy for Compilation

In this section we give an overview of our reduction strategy for compilation. There are five phases: class pre-compilation, redirection of method calls, simplification of expressions, data refinement, and control elimination. Class pre-compilation records the structure of the classes in the source program and introduces the class L, which includes just one method, *lookUp*, to resolve dynamic binding. The redirection of method calls consists of rewriting each method call to a call to *lookUp*. The simplification of expressions eliminates nested expressions in assignments and guards. In the data refinement phase, the abstract space of the source program is substituted by the the concrete space of the target machine. Finally, the purpose of control elimination is to reduce the nested control structure of the source program to the single flat iteration of the normal form. The idea is to progressively change the structure of the source program to get to the normal form.

The compilation process is justified by reduction theorems; for each phase, a theorem asserts the expected outcome, and a main theorem links the intermediate steps and establishes the outcome for the entire process. Since the reduction

theorems are proved from the basic laws of ROOL, they corroborate the correctness of the compilation process. We anticipate the main theorem.

Theorem 1 (Compilation Process) *Let* $cds \bullet c$ *be an executable source program. Given the symbol tables* Φ *and* Ψ*, then there are* s*,* GCS*, and* f *such that*

$$\overline{\Psi}(cds \bullet c) \quad \sqsubseteq \quad cds_{RVM} \bullet Init;\ v : [s, GSC, f]$$

The symbol $\sqsubseteq$ represents the refinement relation; this theorem guarantees that the normal form embeds a correct implementation of the source program. Its constructive proof characterises the compilation process, discussed in the sequel.

Since the source program operates on a data space different from that of the normal form, it does not make sense to compare them directly. A function $\overline{\Psi}$ performs the necessary change of data representation. The symbol table Ψ maps the variables declared in the source program to addresses in the store M, in such a way that $M[\Psi_x]$ holds the value of x.

Before we describe our compilation strategy, we explicit the following restrictions, which we assume to hold for the source programs; there is no name clashing for attributes in the set of class declarations *cds*; name clashing for methods are allowed only in the case of redefinitions; and all references to an attribute have the form **self**.a. These conditions are necessary to ensure the applicability of the compilation rules and the convergence of the overall process. They do not impose any semantic restrictions, and can be satisfied with simple syntactic changes to an arbitrary source program. We further discuss the roles of these conditions during the detailed description of the reduction steps for compilation.

4.1 Class Pre-compilation

The outcome of this phase is summarised by the theorem below. It establishes that the compilation rules applied in this phase are sufficient to end up with a program in a form where all method declarations of the source program are copied to the *lookUp* method of a new class L.

Theorem 2 (Class Pre-compilation) *Let* $cds \bullet c$ *be an executable source program, then there is a program* cds_{RVM}, cds', $L \bullet c'$ *such that*

$$cds \bullet c \quad \sqsubseteq \quad cds_{RVM},\ L,\ cds' \bullet c'$$

where the main command c' *differs from* c *only by trivial casts;* cds' *has the same structure of* cds*, but all the attributes are public; and the class* L *has only a declaration of a method lookUp.*

Initially, the source program does not refer to the set of class declarations cds_{RVM}. The commands that will be introduced in L and in the main command are built using methods, attributes, and types defined in cds_{RVM}. For that reason, when we start the process of restructuring the code, introducing the class L, the need to introduce cds_{RVM} arises. Throughout the compilation process, the class L plays a fundamental role in our strategy: it establishes the basic conditions that will allow the elimination of *cds*, and all source program references.

In order to define *lookUp*, we need to copy all method bodies declared in *cds*. The idea is to transform all method calls to a unique pattern, where the invoked method is always *lookUp*. Once *lookUp* is invoked, the method body associated with the original method call should be selected, and then executed. As a consequence of such transformations, the method declarations in *cds* become useless and can be eliminated. Since the association of a method call with its corresponding method body is affected by dynamic binding and by the use of **super**, it is necessary to address the treatment of these issues.

This compilation phase comprise three steps; the first two change the visibility of the attributes in *cds*, and introduce trivial casts. Both are necessary to avoid syntactic errors that can be originated by the introduction of *lookUp*. The last step in this phase deals with the *lookUp* creation.

Changing visibility of attributes. We need to guarantee that the bodies of the methods do not contain references to private and protected attributes; otherwise, an error can arise when we copy them to the *lookUp* method. Therefore, we change the declarations of the attributes to make them all public. Even though this is not a good idea from a software engineering point of view, this does not change the behaviour of a complete program.

To perform these transformations, we use the laws presented in [2]. Due to space restrictions, we omit most of the compilation rules; our objective in this paper is to present and illustrate the compilation strategy. In our example, we change the declarations of *dir* and *len* in *Step*, and *previous* in *Path*.

Introducing trivial casts. We introduce type casts to produce an uniform program text in which all targets are cast with its static type. The purpose of this step is to explicitly annotate in the program text the declared type of each target. The casts have no effect.

We need a data structure that maps the elements declared in the source program to indexes in the constant pool *CP*. From these indexes, the objects representing the classes, methods, or attributes can be obtained from *Cls*; the constants are represented by objects in *CP*. We assume that this data structure built as part of the syntactic analysis and type checking of the program.

***lookUp* creation.** Basically, *lookUp* consists of a sequence of two conditionals. The first conditional treats the dynamic binding and the use of **super**, whereas the second implements the method body selection. The general format of the *lookUp* declaration is as follows.

meth *lookUp* $\hat{=}$ (**val** S : *Stack*; **res** *Sres* : *Stack*•
  var *mtd* : *Int*; o : *Object*; w : T; •
    $S.Pop(db)$; $S.Pop(mtd)$; $S.Pop(o)$;
    $conditional_1$; $conditional_2$;
    $S.Push(o)$; *Sres* := S;
  end
) **end**

The formal parameters are the operand stacks S and *Sres*. When *lookUp* starts, it pops two values from S: *mtd* and o. The former indicates the method to

be executed, and the latter, the target of the method call. When the dynamic binding is considered in the $conditional_1$, the value of *mtd* may be modified. The $conditional_2$ selects the method body denoted by *mtd*; arguments are handled through variables declared in the list w. The two last commands pushes a copy of o back onto S and assigns S to *Sout*.

Associated with each method m, we have three indexes, ι, σ and δ. The first, $\iota_{C.m}$, identifies the declaration of the method m occurring in the class C. In our robot example, we use the following values: $\iota_{Step.setDirection} \mapsto 1$, $\iota_{Step.setLength} \mapsto 2$, $\iota_{Step.getLength} \mapsto 3$, $\iota_{Path.addStep} \mapsto 4$, $\iota_{Path.getLength} \mapsto 5$, and $\iota_{Path.initialiser} \mapsto 6$. The initialiser declared in *Path* is treated like an ordinary method, and is denoted by 6. There is no ι index for the *setDirection* method of *Path*, because this class does not include a declaration for this method

The second index, $\sigma_{C.m}$, is used to identify references to the method m of the class C in calls of the form **super**.m, which do not require dynamic binding. If a declaration of a method m is shared (through inheritance) by two classes D and E, then $\sigma_{D.m}$ and $\sigma_{E.m}$ have the same value. In our example, we chose the following values for this index: $\sigma_{Step.setDirection} \mapsto 1$, $\sigma_{Path.setDirection} \mapsto 1$, $\sigma_{Step.setLength} \mapsto 2$, $\sigma_{Path.setLength} \mapsto 2$, $\sigma_{Step.getLength} \mapsto 3$, $\sigma_{Path.getLength} \mapsto 5$, $\sigma_{Path.addStep} \mapsto 4$, $\sigma_{Path.initializer} \mapsto 6$. It is important to observe that, since *Path* inherits *setDirection* from *Step*, this method is available in both classes, and so the values of $\sigma_{Step.setDirection}$ and $\sigma_{Path.setDirection}$ are both 1.

The last index, $\delta.m$, identifies references to m in calls that may require dynamic binding: those of the form $le.m$. When m has just one definition, the values of $\sigma_{C.m}$ and δ_m are identical for all classes C in which m is available. Otherwise, δ_m has a value that is not associated with any method declaration by ι and by σ. In our example, the values chosen are as follows: $\delta_{setDirection} \mapsto 1$, $\delta_{setLength} \mapsto 2$, $\delta_{setLength} \mapsto 2$, $\delta_{getLength} \mapsto 0$, $\delta_{addStep} \mapsto 4$, $\delta_{initializer_{Path}} \mapsto 6$. In the case of the method *getLength*, it is defined in *Step* and redefined in *Path*; for this reason, it is associated with 0, an index not used by ι or by σ. This indicates that calls to *getLength* require dynamic binding. For *setLength*, on the other hand, we use 2, which is the value of σ for *setLength* in *Step* and *Path*. This indicates that calls to *setLength* do not require dynamic binding.

A function creates the $conditional_1$ based on the class hierarchy in *Cls* and the indexes above. Starting from the bottom of the hierarchy, for each redefined method m, a nest of conditionals is created. The outermost conditional addresses the class C at the bottom of the hierarchy of classes where m is available. In this conditional, one first guard tests if C is the dynamic type of the object o. If so, the command associated with this guard is an assignment of $\sigma_{C.m}$ to the variable *mtd*. One other guard tests if C is not the dynamic type of the o. In this case, the command associated with this guard is a similar conditional, addressing the immediate superclass of C. For instance, in our example, the only redefined method is *getLength*. We chose 0 as the value of $\delta_{getLength}$. When the $conditional_1$ is created, a test is introduced to check if the value in *mtd* is 0. In this case, the type of o is tested, and 3 or 5, corresponding to $\sigma_{Step.getLength}$ and

$\sigma_{Path.getLength}$, is assigned to *mtd* to indicate which version has to be executed. The resulting conditional is as follows.

$$
\begin{array}{l}
\textbf{if } mtd = \delta_{getLength} \ \rightarrow \\
\quad \textbf{if } o \textbf{ is } Path \ \rightarrow \ mtd := \sigma_{Path.getLength} \\
\quad [] \neg(o \textbf{ is } Path) \ \rightarrow \ \textbf{if } o \textbf{ is } Step \ \rightarrow \ mtd := \Phi_{Step.getLength} \\
\qquad\qquad\qquad\qquad [] \neg(o \textbf{ is } Step) \ \rightarrow \ \textbf{skip} \\
\qquad\qquad\qquad\qquad \textbf{fi} \\
\quad \textbf{fi} \\
[] \neg(mtd = \delta_{getLength}) \ \rightarrow \textbf{skip} \\
\textbf{fi}
\end{array}
$$

In summary, the first conditional tests the dynamic type of o. If C is the current type of o, an assignment of $\sigma_{C.m}$ to the variable *mtd* is done. The value of *mtd* is tested in $conditional_2$ to select the method body that has to be executed.

The creation of $conditional_2$ is based on the class hierarchy described in *Cls*. Each guard in the conditional tests the value in *mtd* to identify a method declaration. The general form of the $conditional_2$ is as follows.

$$
\begin{array}{l}
\textbf{if } []_{\langle 0 \leq \ i \ \leq \ k \rangle}(mtd = \iota_{C.m}) \ \rightarrow \ In(\iota_{C.m}, v); \\
\qquad\qquad\qquad\qquad\qquad\quad pc_{\iota_{C.m}}[o/\textbf{self}](w); \\
\qquad\qquad\qquad\qquad\qquad\quad Out(\iota_{C.m}, r) \\
\textbf{fi}
\end{array}
$$

The notation $pc_{\iota_{C.m}}[o/\textbf{self}]$ expresses the substitution of o for every occurrence of **self** in the parametrised command $pc_{\iota_{C.m}}$: the body of the method declaration identified by $\iota_{C.m}$. It is applied to the list of variables w, which is formed from two other lists v and r. The input arguments are popped from S and stored in v, whereas the result arguments are placed in r. The function $In(\iota_{C.m}, v)$ inspects the signature of m, and creates the list of commands that pop the input values from the operand stack S, initialising the list of variables v. Similarly, $Out(\iota_{C.m}, r)$ creates the list of commands that push the result values on S.

In our example, the $conditional_2$ is formed by six guards, one for each method declaration in the source program. To select the parametrised command corresponding to the method body that has to be executed, the value of *mtd* is used. For our example, the first guard in the $conditional_2$ is as follows.

$$\textbf{if } mtd = 1 \ \rightarrow \ S.Pop(x_1); \ (\textbf{val } d : Int; \ \bullet \ (Step)o.dir := d)(x_1)[] \ \ldots$$

It tests if the value in *mtd* is equal 1. If so, the body of the method *setDirection* declared in *Step* is executed, with the element at the top of the operand stack as argument.

4.2 Redirecting method calls

In this phase all method calls are redirected to *lookUp*, so that the method declarations in *cds* become useless and can, therefore, be eliminated. The outcome is summarised by the Theorem 3: the compilation rules applied in this phase are sufficient to end up with a program where all calls are to the *lookUp* method.

Theorem 3 (Redirection of method calls) *Let* $cds_{RVM}, cds, L \bullet c$ *be an executable program where all attributes in cds are public, and the class L and its lookUp method are as above, then there is a program* cds_{RVM}, cds', $L' \bullet c'$ *such that* $cds_{RVM}, cds, L \bullet c \sqsubseteq cds_{RVM}$, L', $cds' \bullet c'$, *where cds′ contains only the attribute declarations of cds, the main command c′ has the same functionality of c, but neither c′ nor lookUp refer to the methods declared in cds.*

In order to transform each method call that appears in the program into a method call to *lookUp*, we need to simplify the targets. In calls $le.m(e)$ and $\mathbf{super}.m(e)$, both le and e can have nested expressions. The idea is to reduce all possible method calls to a simpler form, suitable to be manipulated by the subsequent steps. The laws needed can be found in [7, 8].

We also use rules that rely on the type of parameter passing used to to determine which arguments have to be pushed onto and popped from S. To illustrate these transformations, we present the rule that addresses calls-by-result. We use the notation $cds_{RVM}, cds, N \triangleright c \sqsubseteq c'$ to mean that the refinement $c \sqsubseteq c'$ holds in the context of the sequence of class declarations of cds_{RVM}, cds, for a command c inside N, which denotes the main command or a class in cds_{RVM} or cds.

Rule 1 (Result parameter)

$cds_{RVM}, cds'', L' \triangleright$

$$
\begin{array}{lll}
(C)le.m(x) & \sqsubseteq & \mathbf{var}\ o : Object;\ \ S : Stack;\ \ V : L \bullet \\
& & \quad S := \mathbf{new}\ Stack;\ \ V := \mathbf{new}\ L;\ \ o := le; \\
& & \quad S.Push(o);\ \ S.Push(\delta_{C.m});\ \ V.lookUp(S); \\
& & \quad S.Pop(o);\ \ S.Pop(x);\ \ le := o; \\
& & \mathbf{end}
\end{array}
$$

provided the definition of m in C has one result parameter, and o, S and V are fresh names.

For the compilation of a call-by-result, a variable block is introduced, declaring an object o, the operand Stack S, and a variable V of class L. New objects are created to initialise S and V, whereas o receives a copy of the object denoted by le. Then, o and $\delta_{C.m}$ are pushed onto S. After the invocation of the *lookUp* method, the value of the parameter is popped from S and assigned to x; the resulting object is assigned to le.

As an example, we consider the method call $p.getLength(out)$ in the main command of the robot program. This method has a result parameter. The result of applying Rule 1 is as follows.

$$
\begin{array}{l}
\mathbf{var}\ o : Object;\ \ S : Stack;\ \ V : L \bullet \\
\quad S := \mathbf{new}\ Stack;\ \ V := \mathbf{new}\ L; \\
\quad o := p;\ \ S.Push(o); \\
\quad S.Push(0);\ \ V.lookUp(S); \\
\quad S.Pop(o);\ \ S.Pop(out);\ \ p := o; \\
\mathbf{end}
\end{array}
$$

The indication that dynamic binding must be performed is done by $S.Push(0)$, where 0 is the value of $\delta_{getLength}$. In *lookUp*, the value associated to *mtd* is modified, based on the type of o. In this case, o is an instance of the class *Path*,

thus, 5 is the value that is assigned to *mtd*, indicating that the method body declared in *Path* is the one that has to be executed. When *lookUp* completes, the value of *out* is popped from S. Finally, the returning value of o is also popped from S, and assigned to p.

Applying compilation rules like that above introduces several variable blocks. To accomplish the result in Theorem 3, we need to apply laws that expand the scope of variable blocks. These laws are standard [8].

When **super** appears as target in a method call, the check of dynamic binding is not necessary. Therefore, we use a slightly different version of the above rules. Instead of δ_m, $\sigma_{(C.super).m}$ is used. This modification prevents the value of *mtd* from being changed in *lookUp*, because $\sigma_{(C.super).m}$ denotes an specific method body, the one declared as m and associated with the immediate superclass of C.

$$
\begin{array}{l}
\textbf{var } p : \mathit{Path}\ o : \mathit{Object};\ \ S : \mathit{Stack};\ \ V : L \bullet \\
\quad p := \textbf{new } \mathit{Path};\ \ S := \textbf{new } \mathit{Stack};\ \ V := \textbf{new } L; \\
\quad o := p;\ \ S.\mathit{Push}(l_0);\ \ S.\mathit{Push}(\mathit{north});\ \ S.\mathit{Push}(o); \\
\quad S.\mathit{Push}(6);\ \ V.\mathit{lookUp}(S);\ \ S.\mathit{Pop}(o);\ \ p := o; \\
\quad o := p;\ \ S.\mathit{Push}(l_1);\ \ S.\mathit{Push}(\mathit{north});\ \ S.\mathit{Push}(o); \\
\quad S.\mathit{Push}(4);\ \ V.\mathit{lookUp}(S);\ \ S.\mathit{Pop}(o);\ \ p := o; \\
\quad o := p;\ \ S.\mathit{Push}(l_2);\ \ S.\mathit{Push}(\mathit{east});\ \ S.\mathit{Push}(o); \\
\quad S.\mathit{Push}(4);\ \ V.\mathit{lookUp}(S);\ \ S.\mathit{Pop}(o);\ \ p := o; \\
\quad S.\mathit{Pop}(o);\ \ p := o;\ o := p;\ \ S.\mathit{Push}(l_3); \\
\quad S.\mathit{Push}(\mathit{south});\ \ S.\mathit{Push}(o);\ \ S.\mathit{Push}(4);\ \ V.\mathit{lookUp}(S); \\
\quad S.\mathit{Pop}(o);\ \ p := o;\ \ o := p;\ \ S.\mathit{Push}(l_4);\ \ S.\mathit{Push}(\mathit{west});\ \ S.\mathit{Push}(o); \\
\quad S.\mathit{Push}(4);\ \ V.\mathit{lookUp}(S);\ \ S.\mathit{Pop}(o);\ \ p := o;\ o := p;\ \ S.\mathit{Push}(o); \\
\quad S.\mathit{Push}(0);\ \ V.\mathit{lookUp}(S);\ S.\mathit{Pop}(o);\ \ S.\mathit{Pop}(\mathit{out});\ \ p := o; \\
\textbf{end}
\end{array}
$$

Fig. 3. Main command obtained after redirecting method calls

The method *getLength* uses recursion to find the length of a robot's path. The redirection of a recursive method call does not imply in any overhead. Any recursion is automatically embedded in recursive calls to *lookUp*. For instance, the recursive call $(\mathit{Path})o.\mathit{previous}.\mathit{getLenth}(\mathit{aux})$ appears in *lookUp*, in the method body related to the *getLength* method of *Path*. When we rewrite this method call, we obtain the following.

$$
\begin{array}{l}
\textbf{var } o_1 : \mathit{Object};\ \ S : \mathit{Stack};\ \ V : L \bullet \\
\quad S := \textbf{new } \mathit{Stack};\ \ V := \textbf{new } L; \\
\quad o_1 := o.\mathit{previous};\ \ S.\mathit{Push}(o_1); \\
\quad S.\mathit{Push}(0);\ \ V.\mathit{lookUp}(S);\ \ S.\mathit{Pop}(o_1); \\
\quad S.\mathit{Pop}(\mathit{out});\ \ o.\mathit{previous} := o_1; \\
\textbf{end}
\end{array}
$$

Recursion arises because we use 0 again to identify the method to be called.

In order to eliminate a parametrised commands, we use the standard definitions in the literature [1]. For each parametrised command, a variable block is introduced. We can combine them using standard laws of ROOL.

This phase considers method calls in *lookUp* and in the main command. In Figure 3, we show the main command resulting from the compilation of our example. For each call in the main command, a variable block is introduced and further manipulated to expand its scope. The first call to *lookUp* has 6 as an argument, indicating that the body of the initialiser declared in *Path* has to be executed to give p its first value. Then, the next four *lookUp* invocations correspond to calls to *addStep*. Finally, the last one refers to a call to *getLength*.

4.3 Simplification of Expressions

This phase eliminates nested expressions that appear in assignments and guards. The expected outcome is stated by Theorem 4.

Theorem 4 (Simplification of Expressions) *Let* cds_{RVM}, cds, $L \bullet c$ *be an executable source program, then there is a program* cds_{RVM}, cds', $L' \bullet c'$ *such that* cds_{RVM}, cds, $L \bullet c \sqsubseteq cds_{RVM}$, cds', $L' \bullet c'$ *where each assignment in* c', cds' *and,* L' *operates through the operand stack, and each boolean expression is a variable.*

Variables that represent the components of *RVM*, and the auxiliary variable V used to invoke *lookUp* are not affected by the transformations of this phase. Basically, the task of eliminating nested expressions in a source program involves rewriting assignments and boolean expressions in the L and in main command. Since new variables are introduced, we apply extra laws to expand the scope of variable blocks.

Before we proceed to the Data Refinement phase, we need to change the parameters of the method *lookUp*. This change is due to the need to access the class hierarchy information and constants stored in the global variables *Cls* and *CP*. Then, the refactoring rule that allows us to add a parameter is applied twice to introduce the desired parameters in *lookUp*.

In Figure 4, we show the result of applying these laws to the class L of our example. The simplification of the conditionals required the introduction of 11 boolean variables. Every boolean expression is assigned to a boolean variable, and these variables replace the corresponding expression in the guards, so that, each guard now consists of a simple boolean variable. Each assignment is rewritten to operate exclusively through the operand stack S. Since the pair $(S.Pop(x),\ S.Push(x))$ is a simulation $(S.Pop(x);\ S.Push(x) \sqsubseteq \mathbf{skip})$. Dispensable sequences of push and pops are eliminated.

4.4 Data Refinement

The data refinement phase replaces the abstract space of the source program with 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 target machine.

The following theorem summarizes the outcome of this phase of compilation.

```
class L
  meth lookUp ≙
    (val CP, Cls, S : Seq Object, Seq ClassInfo, Stack;  res Sres : Stack•
    var x1, x2 : Int;  mtd, aux : Int;  o : Object
        b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11 : Boolean •
      S.Pop(mtd);  S.Pop(o);  S.Push(0);  S.Push(mtd);  S.Equal;
      S.Pop(b1);  S.Push(o is Path);  S.Pop(b2);
      S.Push(o is Path);  S.Neg;  S.Pop(b3);
      if b1 →
        if b2 →  S.Load(5);  S.Pop(mtd);
          []b3 →  if o is Step →  S.Load(3);  S.Pop(mtd);  fi
        fi
      fi;
      S.Push(1);  S.Push(mtd);  S.Equal;  S.Pop(b4);
      S.Push(2);  S.Push(mtd);  S.Equal;  S.Pop(b5);
      S.Push(3);  S.Push(mtd);  S.Equal;  S.Pop(b6);
      S.Push(4);  S.Push(mtd);  S.Equal;  S.Pop(b7);
      S.Push(5);  S.Push(mtd);  S.Equal;  S.Pop(b8);
      S.Push(null);  S.Push(o.previous);  S.NEqual;  S.Pop(b9);
      S.Push(null);  S.Push(o.previous);  S.Equal;  S.Pop(b10);
      S.Push(6);  S.Push(mtd);  S.Equal;  S.Pop(b11);
      if b4 → S.Pop(o.dir)  []b5 → S.Pop(o.len);  []b6 → Push(o.len);
        []b7 → S.Pop(x1); S.Pop(x2); S.Push(o);  S.Pop(o.previous);
              S.Push(o);  S.Push(x1);  S.Push(1);  V.lookUp(S, S);
              S.Push(x2);  S.Push(2);  V.lookUp(S, S);  S.Pop(o);
        []b8 → if  b9 → S.Push(o.previous); S.Push(0);
                    V.lookUp(S, S); S.Pop(o.previous); S.Pop(aux);
                  [] b10 → S.Push(0); S.Pop(aux)
               fi; S.Push(o);  S.Push(3);  V.lookUp(S, S);
              S.Pop(o);  S.Push(aux);  S.Add
        []b11 → S.Pop(x1); S.Pop(x2); S.Push(o);  S.Push(1);
               V.lookUp(S, S); S.Push(x2); S.Push(2);
               V.lookUp(S, S);  S.Pop(o); S.Push(null);  S.Pop(o.previous)
      fi
      S.Push(o);  S.Push(S);  S.Pop(Sres);
    end
  ) end
end
```

Fig. 4. Program generated by the simplification of expressions

Theorem 5 (Data Refinement) *Consider a program of the form* cds_{RVM}, cds, $L \bullet$ (**var** $S, V, w : Stack, L, T \bullet r$ **end**)*, where in* r *there are no local declarations, all assignments are through the operand stack, and all boolean conditions are boolean variables. In addition, the class* L *includes only a declaration of a method called lookUp, whose format is as follows*

meth $lookUp \triangleq$
(**val** $CP, Cls, S : Seq\ Object, Seq\ ClassInfo, Stack$; **res** $Sres : Stack \bullet$
var $o : Object$; $V : L$; $x : T \bullet l$ **end**
)**end**

where l *satisfies the same restrictions as* r*. Then, there are programs* q *and* u *such that*

$\overline{\Psi}(\overline{\omega}(cds_{RVM}, cds, L \bullet$ (**var** $S, V, w : Stack, L, T \bullet r$ **end**)))
$\sqsubseteq$ $cds_{RVM}, L' \bullet$ **var** $S, V : Stack, L' \bullet q$ **end**

and the method lookUp declared in L' *has the following form.*

meth $lookUp \triangleq$
(**val** $CP, Cls, S : Seq\ Object, Seq\ ClassInfo, Stack$; **res** $Sres : Stack \bullet$
var $V : L'$; $M : Memory \bullet u$ **end**
)**end**

where q *and* u *preserve the control structure of* r *and* l*, respectively, but operate mainly on the concrete space.*

Only in the next phase, after introducing the stack of frames F, we can eliminate the local variables V and M, and join the code in the *lookUp* method with the code in the main command.

To carry out the change of data representation, we use the distributivity properties of the function $\overline{\Psi}$ as in [15, 7]. It is a polymorphic function that applies to programs and commands, and distributes over the commands in the class declarations and main command, applying a function with the same name. The function $\overline{\Psi}$ does not affect the classes used to define our interpreter (cds_{RVM}), the components of our target machine, and commands that have no reference to variables or classes of the source program.

For example, after the simplification of expressions, objects are created by $S.Push($**new** $C)$, where the expression **new** C references a class C declared in the source program. The function $\overline{\Psi}$ eliminates this reference, introducing a method call whose parameter is an index in CP corresponding to C. When applied to constructors that deal with control, like the conditional and iteration commands, $\overline{\Psi}$ distributes over the components of these commands. For illustration, Figure 5 presents the initial segment of the class L. The classes and variables declared in the source program are eliminated. The program operates exclusively on the concrete space.

4.5 Control Elimination

In this phase, the nested control structure of the source program is reduced to a single flat iteration. The result is a program in the normal form described in Figure 2. The next theorem summarises the outcome of this phase of compilation.

```
class L
  meth lookUp ≙
    val Cls, CP, S : Seq ClassInfo, Seq Object, Stack; res Sres : Stack•
      S.Store(Ψ_mtd); S.Store(M[Ψ_o])
      S.Load(CP[Φ_0]); S.Load(M[Ψ_mtd]); S.Equal; S.Store(M[Ψ_b1]);
      S.Load(M[Ψ_o]); S.Instanceof(Cls, CP, Φ_Path); S.Store(M[Ψ_b2]);
      S.Load(M[Ψ_o]); S.Instanceof(Cls, CP, Φ_Path); S.Neg; S.Store(M[Ψ_b3]);
      if M[Ψ_b1] →
        if M[Ψ_b2] → S.Load(CP[Φ_5]); S.Store(M[Ψ_mtd]);
          []M[Ψ_b3] → if o is Step → S.Load(CP[Φ_3]); S.Store(M[Ψ_mtd]); fi
        fi
      fi
      ...
end
```

Fig. 5. Class L after the the data refinement

Theorem 6 (Control Elimination) *Consider a program* cds_{RVM}, $L \bullet q$, *which operates mainly on the concrete space, with the method lookUp of L declared in the following form.*

meth *lookUp* $\triangleq$
(**val** CP, Cls, S : *Seq Object, Seq ClassInfo, Stack*; **res** *Sres* : *Stack* •
var o : *object*; $V : L$; M : *Memory* • u **end**
) **end**

Then, there is a normal form program such that $cds_{RVM}, L \bullet q \sqsubseteq cds_{RVM} \bullet I$.

To accomplish the goal established by this theorem, we apply to the commands in the main command and in the body of *lookUp* rules that create the corresponding series of guarded commands. Eventually, we produce a program $v : [s+1, GCS_m, i-1]$ in *lookUp*, and $v : [i+1, GCS_c, f]$ in the main command. In the program below, we present the general form our example at this stage.

```
class L
  meth lookUp ≙ (
    val Cls, CP, S : Seq ClassInfo, Seq Object, Stack; res Sres : Stack •
    var A : N; M : Seq object; • V := new N; v : [s+1, GCS_m, i-1] end )
  end
end
  • var A : N; S : Stack •
      V := newN; S := newStack; v : [i+1, GCS_c, f]
    end
```

To reduce this program to our normal form, we need to eliminate the class L. The only obstacle resides in the method calls to *lookUp* that may exist in GCS_m and GCS_c. Both correspond to conditionals, in which the guarded commands

are closely related to the definition of the behaviour of the machine. To produce the desired normal form, it is necessary to join them. To achieve this goal we need to expand GCS_c using the guards presented in GCS_m. Using basic laws of ROOL, we can extend a conditional by introducing new guarded commands. This leads to a refinement, because the resulting program is more deterministic.

We modify the program above, to obtain the program in the normal form, as shown below. The first action is to deviate the execution flow to the address $i+1$, where the instructions corresponding to the main command start. When a method invocation occurs, the execution flow is deviated to $s+1$. Executing the instructions in GCS_m, the PC eventually reaches the address i. Then, the saved values of PC and M are popped from F, and the execution flow is deviated to just after the invocation. The program ends when PC gets the value f.

$$
\begin{array}{l}
\textbf{class } L \\
\quad \textbf{meth } \mathit{lookUp} \ \hat{=} \ (\\
\quad\quad \textbf{val } \mathit{Cls}, \mathit{CP}, S : \mathit{Seq\ ClassInfo}, \mathit{Seq\ Object}, \mathit{Stack}; \ \textbf{res } \mathit{Sres} : \mathit{Stack} \bullet \\
\quad\quad \textbf{var } A : N; \ M : \mathit{Seq\ object}; \bullet \ V := \textbf{new } N; \ \ v : [s+1, \mathit{GCS}_c, i] \ \textbf{end }) \\
\quad \textbf{end} \\
\textbf{end} \\
\quad \bullet \ \textbf{var } A : N; \ S : \mathit{Stack} \bullet \\
\qquad V := \textbf{new} N; \ S := \textbf{new} \mathit{Stack}; \\
\qquad v : [s, (PC = s) \ \rightarrow \ PC := i+1 \\
\qquad\qquad [] \ \mathit{GCS}_m \\
\qquad\qquad [] \ (PC = i) \ \rightarrow \ F.\mathit{Pop}(PC); \ F.\mathit{Pop}(M) \\
\qquad\qquad [] \ \mathit{GCS}_c, f] \\
\quad\quad \textbf{end}
\end{array}
$$

Using the next rule we can eliminate method calls to *lookUp*, and afterwards, we can eliminate the auxiliary class L.

Rule 2 (Eliminating method calls)

$$
\begin{array}{l}
\mathit{cds}_{RVM} \ L \ \triangleright \\
V.\mathit{lookUp}(\mathit{Cls}, \mathit{CP}, S) \quad \sqsubseteq \quad F.\mathit{Push}(PC); \ \ F.\mathit{Push}(M); \ \ PC := s+1;
\end{array}
$$

This rule compiles a call to *lookUp* by pushing the value of the PC and M onto F, and the assigning of the value $s+1$ to PC. In this address, the code relative to *lookUp* is stored. Once the frame stack F is introduced, we are able to eliminate all method calls, because F plays the same role of the implicit stack used when a method is called. Therefore, we can reduce the whole program to a flat iteration.

4.6 The Compilation Process

Here, we sketch the proof to the Theorem 1. Basically, we want to transform $\overline{\Psi}(\mathit{cds} \ \bullet \ c)$ into $\mathit{cds}_{RVM} \bullet \ \mathit{Init}; \ \ v : [s, \mathit{GSC}, f]$. From Theorem 2 (Class Precompilation), we can transform $\mathit{cds} \ \bullet \ c$, and obtain cds_{RVM}, cds', $L \bullet \ c'$. At this point, we can refer to the Theorem 3 (Redirection of method calls) which establishes that $\mathit{cds}_{RVM}, \mathit{cds}, L \ \bullet \ c \quad \sqsubseteq \quad \mathit{cds}_{RVM}, \ L, \ \mathit{cds}' \ \bullet \ c'$. Then Theorem 4

(Simplification of Expressions), states that we can obtain cds_{RVM}, cds', $L' \bullet c'$ from cds_{RVM}, cds, $L \bullet c$. Using monotonicity of $\overline{\Psi}$, we can conclude that $\overline{\Psi}(cds \bullet c) \sqsubseteq \overline{\Psi}(cds_{RVM}, cds', L' \bullet c')$. At this point, based on the Theorem 5 (Data Refinement), cds is eliminated and the outcome program operates over the concrete space. Finally, from Theorem 6 we achieve a program in our normal form, $cds_{RVM} \bullet Init;\ v : [s, GSC, f]$.

5 Final Considerations

As an attempt to address the correct implementation of object-oriented programs, we have proposed a refinement strategy for the compilation of ROOL, a language that includes classes, inheritance, dynamic binding, and recursion. This language is sufficiently similar to Java to be used in meaningful case studies; our result represents significant advance on previous work. In [7], we detail the compilation rules for the phases of simplification of expressions, data refinement, and control elimination. Here, we focus on the overall strategy for the compilation, illustrating the whole process through a case study.

The classes declared in the source program have to be eliminated during the compilation process. In order to remove them, we developed a strategy based on the introduction of an auxiliary class L that allows us to eliminate the references to methods of the source program. Inheritance is treated through the generation of a data structure *Cls* resembling the original class hierarchy. Dynamic binding is handled with the use of a function to construct a conditional to check the type of the target object at run time.

The main difference between our work and those in [3, 16] resides in the fact that their approach is based on verification, instead of on calculation. Recently, a case study in verified program compilation from imperative program to assembler code was presented in [17]. The compiled code is data refined by calculation. That case study, however, does not comprise object-oriented features.

In [2], a similar strategy for normal form reduction was adopted as a measure of completeness of the set of proposed laws for ROOL. Here, a similar set of laws, together with specific compilation rules, are used to carry out the design of a provably correct compiler. Obviously, due to the nature of our application, our normal form and our strategy need to be different.

We are currently working on the proof soundness of the compilation rules. Initial results are reported in [9]. The proofs are based on basic laws of rule, that are sound with respect to its weakest precondition semantics [2].

Further work is needed towards the mechanisation of this approach. Its algebraic nature makes the mechanisation easier, allowing the use of a term rewrite system as a tool for specification, verification, and prototype implementation. We already have initial results in this direction.

6 Acknowledgments

Adolfo Duran is supported by UFBA (Universidade Federal da Bahia, Brazil) and CAPES: grant BEX0786/02-0. The other authors are partially suported

by CNPq: grants 520763/98-0 and 472204/01-7 (Ana Cavalcanti), 521039/95-9 (Augusto Sampaio), and 680032/99-1.

References

1. R. J. R. Back. Procedural abstraction in the refinement calculus. Technical Report Ser. A No. 55, Department of Computer Science, Abo - Finland, 1987.
2. P. Borba, A. Sampaio, and M. Cornélio. A refinement algebra for object-oriented programming. In *To Appear in the Proceedings of ECOOP 2003*, 2003.
3. 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.
4. A. Cavalcanti and D. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Enginnering*, 26(08):713–728, 2000.
5. M. Cornélio, A. Cavalcanti, and Augusto Sampaio. Refactoring by transformation. In Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science*, 2002.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Engewood Cliffs, 1976.
7. A. Duran, A. Cavalcanti, and A. Sampaio. Refinement algebra for formal bytecode generation. In *ICFEM 2002 - 4th International Conference on Formal Engineering Methods*, pages 347–358, Shanghai, China, October 2002. Springer-Verlag.
8. A. Duran, A. Cavalcanti, and A. Sampaio. A refinement strategy for the compilation of classes, inheritance, and dynamic binding (extended version). Technical report, Computing Laboratory, University of Kent at Canterbury, 2003.
9. A. Duran, A. Sampaio, and A. Cavalcanti. Formal bytecode generation for rool virtual machine. In *IV WMF— Workshop on Formal Methods.* PUC—Rio de Janeiro/Brazil, October 2001.
10. C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
11. Tim Lindholm and Frank Yellin. *The java Virtual Machine Specification.* Addison-Wesley, 1997.
12. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Symposium on Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.
13. C. Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.
14. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS.* Springer-Verlag, Heidelberg, Germany, 1997.
15. A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing.* World Scientific, 1997.
16. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer-Verlag, 2001.
17. L. Wildman. A formal basis for a program compilation proof tool. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME2002: Formal Methods – Getting IT Right*, Copenhagen, Denmark, July 2002. International Symposium of Formal Methods Europe, Springer.