Under consideration for publication in Formal Aspects of Computing

An Algebraic Approach to the Design of Compilers for Object-oriented Languages

Adolfo Duran[1] and Ana Cavalcanti[2] and Augusto Sampaio[3]

[1]Universidade Federal da Bahia, Centro de Processamento de Dados
MEFES Research Group, CEP 40170-110, Salvador-BA Brazil
[2]University of York, Department of Computer Science,
York YO10 5DD, UK
[3]Universidade Federal de Pernambuco, Centro de Informática
Caixa Postal 7851, CEP 50732-970, Recife-PE Brazil

Abstract. In this paper we describe an algebraic approach to construct provably correct compilers for object-oriented languages; this is illustrated for programs written in a language similar to a sequential subset of Java. It includes recursive classes, inheritance, dynamic binding, recursion, type casts and test, assignment, and class-based visibility, but a copy semantics. In our approach, we tackle the problem of compiler correctness by reducing the task of compilation to that of program refinement. Compilation is identified with the reduction of a source program to a normal form that models the execution of object code. The normal form is generated by a series of correctness-preserving transformations that are proved sound from the basic laws of the language; therefore it is correct by construction. The main advantages of our approach are the characterisation of compilation within a uniform framework, where comparisons and translations between semantics are avoided, and the modularity and extensibility of the resulting compiler.

Keywords: algebraic transformation; refinement; compiler correctness

1. Introduction

Ensuring the correct translation of source into executable code has been a significant research challenge. Correctness proofs performed on programs written in high-level languages cannot ensure correctness of an application. There are various issues involved, and, in particular, a point that needs to be addressed is the correctness of the compiler. Several approaches based on a variety of semantic styles have been proposed. For procedural languages, the design of correct compilers is well understood; examples of successful strategies are [Wat03, Wil02, Sam97, MO97, Pol81, TWW81, MP67]. Provably correct compilation of object-oriented features, however, is still a challenge. There are results based on a formalisation of Java and the JVM, most of them using ASM models [NO98, SSB01, KN06].

The compilation process involves several stages. Even for languages with an elaborate syntax or typing system, including object-oriented languages, there are well understood theories and techniques to handle syntax and type

Correspondence and offprint requests to: Adolfo Duran, Universidade Federal da Bahia, Centro de Processamento de Dados CEP 40170-110, Salvador-BA Brazil. e-mail: adolfo@ufba.br

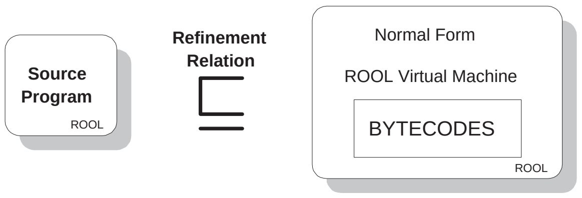

Fig. 1. Compilation approach

checking. In our work, we deal with a language whose syntax, typing system, and well-formedness is completely formalised [CN00]. Therefore, we address the problem of correct code generation of well formed programs only.

Our approach is calculational; it is inspired on that first described in [HHS93], and further developed for imperative programs in [Sam97]. Its main advantage, which accounts for much of its simplicity, is that all the reasoning is carried out within a single framework of an object-oriented language with an algebraic semantics. More precisely, instead of having a semantic model for the source language, a different model for the target language, and a mapping between them to justify the translation, we have just one mathematical theory that is used to model both the source and the target languages, and an accompanying correctness relation that justifies the translation.

In more detail, we tackle compiler correctness by characterising compilation as program refinement into a normal form. The product of the compilation is a normalised program in the source language itself. The normal form is an interpreter-like program that emulates the behaviour of the target machine and the compiled program in the target language. It is a model for a virtual machine inspired on a sequential Java Virtual Machine (JVM) [LY97]. From this interpreter we can capture the sequence of generated instructions.

What we have is a constructive approach in that compilation is carried by the systematic application of rules that are based on basic algebraic laws of the language. Therefore, the correctness of the compiler follows directly from the soundness of the rules. Figure 1 illustrates the correctness process based on program refinement.

Refinement is formalised as an ordering relation on programs: $P_1 \sqsubseteq P_2$ means that P_2 meets every purpose and satisfies every specification satisfied by P_1, that is, P_2 is at least as good as P_1. The relation $\sqsubseteq$ is a partial ordering weaker than equality; the set of algebraic laws that holds for equality is a strict subset of that which holds for $\sqsubseteq$, and therefore refinement is often easier to be established than equality.

Our source language is a subset of sequential Java enriched with specification constructs; it is called ROOL (an acronym for Refinement Object-oriented Language). Although pointers are ubiquitous in practice, ROOL adopts a copy semantics, and so we do not model references or sharing. This does simplify the semantics of the language, but ROOL is sufficiently similar to Java to be used in meaningful case studies and to capture many of the central difficulties that arise on the formalisation of compilation. The laws of ROOL related to object-oriented features are not restricted to the copy semantics; it has been shown in [SSL08] that they also hold in the presence of pointers. This makes ROOL useful for exploring a comprehensive set of laws for applications of program transformation. Refactoring rules justified by the basic algebraic rules of ROOL has already been considered in [Cor04].

In Figure 2 we present a simple program in $ROOL$ to illustrate our compilation approach. This program contains two class declarations and a main command. The class A declares the protected attribute i (whose scope includes A and its subclass B) and the method inc whose body is an assignment that increments i by 2. A reference to an attribute uses **self** to make it explicit that it is an access to an attribute of the current object; so we write **self**.i rather than just i.

The class B extends (inherits from) A, declaring the private attribute j (whose scope is just B) and the methods inc and out. The method inc declared in B is a redefinition of the method of the same name declared in its superclass; it uses the call **super**.inc to invoke the superclass method (to increment i), and then it increments j. The method out has a result parameter l that records the sum of the attributes i and j.

In the main command, a variable x of type A is declared and initialised with an object of type B. Afterwards, there is a call to inc and a call to out with a parameter $result$; this is a global variable, and we use it to represent the output.

There are five phases of compilation. The first phase (Class Pre-compilation) consists of changing the visibility of the attributes i and j to public, and moving the redeclared method inc upwards in the class hierarchy. The resulting program is in Figure 3. The method inc is declared only once, within the class A. The body of this new method inc is a conditional that checks the type of the object associated with the call (**self**) to determine which of the bodies of the original versions of inc is to be executed. Uses of **super** calls are eliminated and all expressions are cast to record their types; this is especially important in the case of **self**, because it is context sensitive. For instance, the assignment **self**.$j :=$ **self**.$j + 1$ in the body of method inc in class B is cast as $(B$ **self**$).j := (B$ **self**$).j + 1$.

In the next compilation phase, Redirection of Method Calls, we introduce a special class L; Figure 4 shows the

```
**class** $A$
   **prot** $i$ : **int**;
   **meth** $inc \triangleq$ ( • **self**.$i$ := **self**.$i + 2$)
**end**
**class** $B$ **extends** $A$
   **pri** $j$ : **int**;
   **meth** $inc \triangleq$ ( • **super**.$inc$; **self**.$j$ := **self**.$j + 1$)
   **meth** $out \triangleq$ (**res** $l$ : **int** • $l$ := **self**.$j$ + **self**.$i$)
**end**
• **var** $x : A$ •
      $x$ := **new** $B$;
      $x.inc$;
      $x.out(result)$
   **end**
```

Fig. 2. Simple example to illustrate our compilation approach

```
**class** $A$
   **pub** $i$ : **int**;
   **meth** $inc \triangleq$ ( •
         if (**self is** $A$) $\rightarrow$ ($A$ **self**).$i$ := ($A$ **self**).$i + 2$
         $\square$ $\neg$(**self is** $A$) $\rightarrow$ ($A$ **self**).$i$ := ($A$ **self**).$i + 2$; ($B$ **self**).$j$ := ($B$ **self**).$j + 1$
         **fi**)
**end**
**class** $B$ **extends** $A$
   **pub** $j$ : **int**;
   **meth** $out \triangleq$ (**res** $l$ : **int** • $l$ := ($B$ **self**).$j$ + ($B$ **self**).$i$)
**end**
• **var** $x : A$ •
      $x$ := **new** $B$;
      $x.inc$;
      $x.out(result)$
   **end**
```

Fig. 3. Class Pre-compilation

result for our example. For each method m declared in A and B, there is an associated method lm in L. For instance, the method out declared in B has its counterpart in L declared as $lout$. A method lm in L has the same behaviour of m, but it is through an operand stack that lm obtains its arguments, its target object, and returns the results of its execution. The idea is to simulate the way method invocation is defined in our stack based target machine.

Since ROOL does not include value-result parameters, each method of L has two parameters: S_{in} is a value parameter, a stack used to hold the arguments of the original method, and S_{out} is a result parameter, used to return the results of the original method. The target of a call to the original method is represented as a local variable o. A method in L simulates the behaviour of the original method, because its body contains the parametrised command that defines the original method, with all occurrences of **self** replaced with the local variable o. In addition, in the case of the method $lout$, for instance, the argument r is passed by result to the parametrised command copied from out, imitating the same behavior of a method call whose target is o and whose result argument is r.

In the main program, calls to inc and out are replaced with calls to $linc$ and $lout$. For each call a variable block is introduced to declare the operand Stack S and a variable V of class L. It is necessary to introduce the variable V because static methods are not allowed in $ROOL$. New objects are created to initialise S and V, and x, the target of the calls, is pushed onto S. Then, we have the call to the method in L. In the case of out, which has a result parameter, after calling the $lout$ method, the value of the result argument is popped from S and assigned to the global variable $result$. For both calls the resulting target object is popped from the stack and assigned to x.

```
class $A$
  pub $i$ : int;
end
class $B$ extends $A$
  pub $j$ : int;
end
class $L$
  meth $linc \triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ •
      var $o$ : object• $Pop(S_{in}, o)$;
        if $(o$ is $A) \rightarrow (A\ o).i := (A\ o).i + 2$
         $\Box\ \neg(o$ is $A) \rightarrow (A\ o).i := (A\ o).i + 2;\ (B\ o).j := (B\ o).j + 1$
        fi
        $Push(S_{in}, o);\ S_{out}; =\ S_{in}$
      end)
  meth $lout \triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ •
      var $o$ : object; $r$ : int • $Pop(S_{in}, o)$;
        (res $l$ : int • $l := (B\ o).j + (B\ o).i)(r)$;
        $Push(S_{in}, r);\ Push(S_{in}, o);\ S_{out}; =\ S_{in}$
      end)
end
• var $x$ : $A$ •
    $x :=$ new $B$;
    var $S$ : $Stack$; $V$ : $L$ • $S :=$ new $Stack$; $V :=$ new $L$;
      $Push(S, x);\ V.linc;\ Pop(S, x)$;
    end
    var $S$ : $Stack$; $V$ : $L$ • $S :=$ new $Stack$; $V :=$ new $L$;
      $Push(S, x);\ V.lout(S, S);\ Pop(S, x);\ Pop_i(S, result)$
    end
  end
```

Fig. 4. Redirection of Method Calls

The redirection of method calls allows the elimination of all methods declared in A and B. The introduction of L is necessary to set the scene to transform all calls in the program to a uniform pattern that uses an operand stack.

In Figure 5, we present the general control structure of our example after the next phase of compilation: Control Elimination. The main command and each method body is placed in a code segment. Each segment corresponds to guarded commands in the body of a loop, with the guards indicating the location of the instructions. In our example, the segment corresponding to $linc$ starts at address k_1, whereas that associated to $lout$ starts at address k_2; finally, k_3 denotes the address where begins the segment corresponding to the main command. At this point the main command of the compiled program is in the same format of the normal form main command.

The set of class declarations cds_{RVM} describes the RVM components. We use cds_{AB} to stand for the declarations of the classes A and B of our example. In the main command, local variables PC, S, and F represent the program counter, an operand stack and a frame stack. The stacks are initialised with fresh objects of the class $Stack$, and PC is initialised with the address s of the first instruction of the target code. The body of the loop is a conditional involving a set of guarded commands in the style of Dijkstra's language [Dij76].

Figure 6 presents the guarded commands corresponding to the main command. The first command pushes a new object of type B onto the operand stack. The second pops a copy of this object and stores it in the variable x. These two commands assign a new object of type B to the variable x. Then, the value in x is pushed onto the operand stack because x is the target object of the call to $linc$. The guard $PC = k_3 + 06$ holds the command to deviate the control flow to k_1, the initial address of the segment corresponding to the method $linc$. After that, the resulting object is stored in x. The interval from $PC = k_3 + 10$ to $PC = k_3 + 16$ corresponds to the call to $lout$. First the target object is pushed onto the operand stack, then the control flow branches to k_2, after that the resulting target object is stored in x, and finally, the result parameter is popped and stored in the global variable result.

The last phase of compilation, Data Refinement, replaces references to variables with explicit accesses to their

```
$cds_{RVM}\ cds_{AB}$ •  var $PC$ : int; $S, F : Stack, x_1, x_2 : Boolean;\ r$ : int; $o_1, o_2 : Object;\ x : A$ •
                           $S :=$ new $Stack$;
                           $F :=$ new $Stack$;
                           $PC := s$;
                           while $PC \geq s \wedge PC < f \rightarrow$
                             if  $PC = s \rightarrow PC := k_5$
                               $\square\ PC = k_1 \rightarrow\ \ linc$
                               ⋮
                               $\square\ PC = k_2 \rightarrow\ \ lout$
                               ⋮
                               $\square\ PC = k_3 \rightarrow\ \ Main\ Command$
                             fi
                           end
                         end
```

Fig. 5. General structure of our example after the Control Elimination phase

```
⋮
$\square\ PC = k_3$          $\rightarrow Load_{CE}(\mathbf{new}\, B)$
$\square\ PC = k_3 + 02$     $\rightarrow Store_{CE}((A\ x))$
$\square\ PC = k_3 + 04$     $\rightarrow Load_{CE}((A\ x))$
$\square\ PC = k_3 + 06$     $\rightarrow invoke(k_1)$
$\square\ PC = k_3 + 08$     $\rightarrow Store_{CE}((A\ x))$
$\square\ PC = k_3 + 10$     $\rightarrow Load_{CE}((A\ x))$
$\square\ PC = k_3 + 12$     $\rightarrow invoke(k_2)$
$\square\ PC = k_3 + 14$     $\rightarrow Store_{CE}((A\ x))$
$\square\ PC = k_3 + 16$     $\rightarrow iStore_{CE}(result)$
```

Fig. 6. Sequence of guarded commands corresponding to the main command

positions in the memory. Figure 7 presents the set of guarded commands associated to the main command, obtained in the end of the compilation process. The guarded commands preserve the same structure shown in Figure 7, but due to the chance of data representation, the abstract space of the source program is replaced with the concrete state of the target machine. The term Ψ_x indicates the location in the storage where the object in x is stored.

In [DCS02], we detail some of the compilation rules for the imperative subset of ROOL, based on the approach of [HHS93, Sam97]. In [DCS03] we consider the compilation of the object-oriented features of ROOL; we present a systematic strategy, but do not provide the algebraic rules that support a constructive proof of correctness of the compilation process. The current paper builds on these preliminary results to consolidate a complete characterisa-

```
⋮
$\square\ PC = k_3$          $\rightarrow load(\Psi_x)$
$\square\ PC = k_3 + 02$     $\rightarrow store(\Psi_x))$
$\square\ PC = k_3 + 04$     $\rightarrow load(\Psi_x)$
$\square\ PC = k_3 + 06$     $\rightarrow invoke(k_1)$
$\square\ PC = k_3 + 08$     $\rightarrow store(\Psi_x)$
$\square\ PC = k_3 + 10$     $\rightarrow Load(\Psi_x)$
$\square\ PC = k_3 + 12$     $\rightarrow invoke(k_2)$
$\square\ PC = k_3 + 14$     $\rightarrow store(\Psi_x)$
$\square\ PC = k_3 + 16$     $\rightarrow store(\Psi_{result})$
```

Fig. 7. Sequence of guarded commands corresponding to the main command after the Data Refinement

tion of our approach to provably correct compilation of object-oriented languages. Particularly, based on the laws of ROOL [BSCC04], the presentation here is purely algebraic. Our strategy is justified by 36 novel rules; we discuss them and their soundness argument here. The complete list of rules and their proofs can be found in [Dur05]. We illustrate the entire compilation through a more elaborate example than the one considered in the introduction.

The remainder of this paper is structured as follows. In the next section we give an overview of ROOL and its laws. In Section 3 we describe the target machine, and in Section 4 we present the compilation process. Finally, in Section 5 we report on related work, summarize our results and consider topics for further research.

2. The language and its algebraic laws

In this section we present an informal description of ROOL. Essentially, it is an object-oriented language in the style of Java [GJSB00]. In [BSCC04] we have provided an algebraic semantics for ROOL which is sound with respect to the weakest precondition semantics of [CN00]. We have also formalised a comprehensive set of refactorings as program transformations justified by the algebraic laws [CCS02, Cor04].

ROOL includes programming and specification constructs, in the style of Morgan's refinement calculus [Mor94]. It was designed to support reasoning about object-oriented programming. Similarly to Java, ROOL includes classes, inheritance, dynamic binding, recursion, assignment, type casts and tests, class-based visibility, and many other imperative features. As already discussed, however, it has a copy semantics.

A program in ROOL consists of a sequence of class declarations (cds), 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 $x_1 : T_1;\ \}^*$                  $//private\ attributes$
   $\{$prot $x_2 : T_2;\ \}^*$                 $//protected\ attributes$
   $\{$pub $x_3 : T_3;\ \}^*$                  $//public\ attributes$
   $\{$meth $m \triangleq (pds \bullet c)\}^*$    $//public\ methods$
   $\{$new $\triangleq c$ end$;\ \}^*$             $//Initialisers$
end
```

Classes can be recursive: attributes and method parameters of a class N, for example, can have type N. Subclassing is defined through the clause **extends**; it determines the immediate superclass of the declared class. Whenever the clause **extends** is omitted, the built-in class **object** is regarded as the superclass.

Attributes are declared with visibility modifiers similar to those of Java: **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 initialisers: a syntactic sugar for methods that are called after creating objects. Differently from Java, they have no parameters. Since our methods are parametrised, however, it is easy to extend our work to cater for parametrised constructors. The compilation process is already sorted out to cover such cases.

Types T are those of attributes, parameters, local variables, and expressions. They are either primitive (like boolean or integer) or class names. To refer to variable identifiers, we use x, whereas f stands for a literal or built-in function; we also use b to stand for boolean expressions, and X for a recursive block identifier.

ROOL expressions are those typically found in object-oriented languages; Table 1 defines the syntax. 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. The **self** and **super** expressions are similar to *this* and *super* of Java. The use of **self** is not optional when accessing an attribute a of the current class; we write **self**.a. The type test expression **is** corresponds to *instanceof* in Java and does not require exact type matching. A type cast is written $(N)e$; its value is the object denoted e, if it belongs to class N. Whenever a value of e is not of the dynamic type N, $(N)e$ is an error. The expression null is the default value of a non-initialised object variable. Attribute selection $e.x$ can be a run-time error if the value of e is **null**. Expressions such as **null**.x and (**null**; $x : e$) cannot be successfully evaluated; they yield **error**, a special value that can only be used in predicates and lead the commands in which they appear to **abort**. The left expressions are those that can appear as targets of assignments and method calls, and as result arguments. In Table 2, they are the subset Le of Exp.

The imperative constructs of ROOL are based on the language of Morgan's refinement calculus, which extends Dijkstra's language of guarded commands [Dij76]. The body of methods and constructors are commands similar to those described in [Mor94]; Table 3 describes their syntax. Following the approach adopted in [Sam97], we also have constructors for dynamic declaration (**dvar** and **dend**) explained in the sequel.

$e \in Exp ::=$	$(e;\ x : e)$	update expression
	$\mid$ **self**	the current object that the method operates on
	$\mid$ **super**	reference to the superclass
	$\mid$ **null**	null value
	$\mid$ **error**	error value
	$\mid$ **new** N	object creation
	$\mid x$	variable
	$\mid f(e)$	built-in application
	$\mid e$ **is** N	type test
	$\mid (N)e$	type cast
	$\mid e.x$	attribute selection

Table 1. Grammar for expressions

$le \in Le$	$::=$	$le1 \mid$ **self**$.le1 \mid ((N)le).le1$
$le1 \in Le1$	$::=$	$x \mid le1.x$

Table 2. Grammar for left expressions

For convenience, we use x, le, e, and T to represent lists of identifiers, left expressions, expressions and types, as well as individual elements of these syntactic categories. An assignment has the form $le := e$. 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 the frame x.

The usual sequential composition of commands c_1 and c_2 is written $c_1;\ c_2$. The conditional is composed by a collection of guarded commands $b_i \rightarrow c_i$ separated by [], as in Dijkstra's language [Dij76].

For reasoning purposes, it is useful to have independent constructors to introduce a variable and to end its scope, especially in the Data Refinement phase. The construct **dvar** introduces x with a dynamic scope. Operationally, an unbounded stack is associated with each variable, so that rather than creating a new variable, **dvar** x has the effect of pushing the current value of x onto the stack, assigning to x an arbitrary value. The dynamic scope of x extends up to the end of the static scope of x or the execution of the command **dend** x. The effect of **dend** x is to pop the stack, assigning the popped value to x. When the stack is empty, this value is arbitrary.

Methods are parametrised commands pc in the style of Back; they can be applied to arguments to yield a command. Since **self** is not optional in our language, a call to a method m on the current object is written **self**$.m$. Furthermore, we can call the method m declared by the superclass by writing **super**$.m$.

A parametrised command can have the form **val** $x : T \bullet c$ or **res** $x : T \bullet c$, corresponding to the traditional conventions of parameter passing known as call-by-value and call-by-result. Table 4 describes the syntax. It is possible to combine an arbitrary number value and result parameter declarations.

The block **rec** $X \bullet c$ **end** introduces a recursive command named X with body c; occurrences of X in c are recursive calls. A **while** command can be defined in the standard way in terms of recursion.

Definition 1. (While definition)

$$\textbf{while } b \ \rightarrow\ c \textbf{ end} \quad \overset{def}{=} \quad \textbf{rec } X \bullet \textbf{if } b \rightarrow\ c;\ X \ [] \ \neg b \rightarrow \textbf{skip fi end}$$

$\diamond$

As a reasoning language, ROOL yields a framework where both programming and specification operators have the same status: they are both modelled as predicate transformers. Specifications are abstract programs; they are

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

Table 3. Grammar for commands

$pc \in PCom$	$::=$	$pds \bullet c$	parametrisation
		$\mid le.m \mid m$	method calls
$pds \in Pds$	$::=$	$\varnothing \mid pd \mid pd;\ pds$	parameter declarations
$pd \in Pd$	$::=$	**val** $x : T$ $\mid$ **res** $x : T$	

Table 4. Grammar for parametrised commands

```
class $Step$
  pri $dir, len$ : int;
  meth $getLength \triangleq$ (res $l$ : int; • $l :=$ self.$len$)
end
class $Path$ extends $Step$
  pri $previous$ : $Path$;
  meth $addStep \triangleq$ (val $d, l$ : int •
      self.$previous :=$ self; self.$dir := d$; self.$len := l$)
  meth $getLength \triangleq$ (res $l$ : int •
    var $aux$ : int •
        if (self.$previous =$ null) $\rightarrow aux := 0$
          [] (self.$previous <>$ null) $\rightarrow$ self.$previous.getLength(aux)$
        fi;
        super.$getLength(l)$; $l := l + aux$;
    end)
end
• var $p$ : $Path$ • $p :=$ new $Path$;
    $p.addStep(north, l_1)$; $p.addStep(east, l_2)$; $p.addStep(south, l_3)$; $p.addStep(west, l_4)$;
    $p.getLength(out)$;
  end
```

Fig. 8. ROOL program for keeping track of a robot's path

not executable, but they aid in reasoning. We can start from an abstract specification of a program's behaviour and gradually refine it, obtaining a mix of code and specifications, until we get a program with executable constructs only.

Since some constructs are not implementable, the compilation is restricted to the executable subset of ROOL. More specifically, the following constructs are used only for reasoning purposes: multiple assignments, specification statements, angelic variable block, and the dynamic scope constructs. They are not covered by our compilation strategy.

In Figure 8, we give an example of an executable program in ROOL that we use to illustrate our compilation strategy. It keeps 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 is the total length of the route described by the robot.

Two classes are declared in Figure 8, $Step$ and $Path$. The class $Step$ has two integer attributes: dir and len, corresponding to the direction and length of a step of the robot. 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 assigns the direction and length to the attributes dir and len to record the current step. The length of a path is calculated by $getLength$, a recursive redefinition of the method with the same name in $Step$. Each recursive call to $getLength$ visits a step in the path; it traverses the list of steps. 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 in $Step$, which is $Path$'s superclass, is invoked.

In the main command, the free variables l_1 to l_4 represent the length of each robot's step, the free variables $north$, $south$, $east$ and $west$ denote the direction of each robot's step, and out represents the output of the program. The main command is a variable block in which an object of type $Path$ is created and several steps are added to this path. The last method call retrieves the total length of the path. Input and output are represented by free variables.

For simplicity of reasoning, we assume that names of attributes, methods, variables and so on, are not reused. It is simple to check or enforce this restriction by a syntactic preprocessing of the program.

Basic programming laws of ROOL, for both commands and classes, are presented in [BSC03, BSCC04]. Class laws are expressed as equations $cds_1 =_{cds,c} cds_2$ to state that the sequences of class declarations cds_1 and cds_2 are

Law 1. (move redefined method to superclass)

$$
\boxed{\begin{array}{l}
\textbf{class } B \textbf{ extends } A\\
\quad ads\\
\quad \textbf{meth } m \triangleq (pds \bullet b)\\
\quad ops\\
\textbf{end}\\
\textbf{class } C \textbf{ extends } B\\
\quad ads'\\
\quad \textbf{meth } m \triangleq (pds \bullet b')\\
\quad ops'\\
\textbf{end}
\end{array}}
=_{cds,c}
\boxed{\begin{array}{l}
\textbf{class } B \textbf{ extends } A\\
\quad ads\\
\quad \textbf{meth } m \triangleq (pds \bullet\\
\qquad \textbf{if } \neg(\textbf{self is } C) \rightarrow b\\
\qquad\quad [\,]\ \textbf{self is } C \rightarrow b'\\
\qquad \textbf{fi})\\
\quad ops\\
\textbf{end}\\
\textbf{class } C \textbf{ extends } B\\
\quad ads'\\
\quad ops'\\
\textbf{end}
\end{array}}
$$

provided

($\leftrightarrow$) (1) **super** and private attributes do not appear in b'; (2) **super**.m does not appear in ops';
($\rightarrow$) b' does not contain uncast occurrences of **self**;
($\leftarrow$) m is not declared in ops'.

Fig. 9. Example of an algebraic law related to object-oriented features

equivalent in the context of the class declarations in cds, and the main command c. Thus, both ($cds\ cds_1$) and ($cds\ cds_2$) are valid sequences of class declarations; juxtaposition of class declarations represents their concatenation. The above equation is just an abbreviation for $cds_1\ cds \bullet c = cds_2\ cds \bullet c$.

In order to make explicit which conditions must be satisfied for the application of a law, we adopt the following notation: '($\rightarrow$)' indicates conditions that are necessary when applying the law from left to right; '($\leftarrow$)' indicates conditions that are necessary when applying the law from right to left; and, finally, '($\leftrightarrow$)' indicates conditions that are necessary for applications in both directions. As an example, Figure 9 presents a law that establishes that a method declaration and its redefinition can be merged into a single declaration in the superclass. The appropriate behaviour is determined by type tests in the resulting method. This law is used later on as part of our compilation strategy.

The provisos about **super** are necessary when moving m from a subclass to a superclass, or vice-versa, since this can potentially change the meaning of **super**.m. The other provisos concern the validity of the programs involved. The body of m can only be moved up when it does not refer to private elements of the class and all uses of **self** are cast, so that any references to particular attributes or methods of the superclass remain valid. Finally, part of a method definition can only be moved down if the subclass does not declare a method of the same name already.

We present in Figure 10 a law of command. Due to dynamic binding, the use of the copy rule is complex since replacing the call by a method body depends on the type of the target object. We would have to inspect the type of the object to determine the correct class from which we can obtain the body of the method. Instead, we use Law 1 (move redefined method to superclass) to handle dynamic binding separately. Surprisingly, maybe, in doing so, we need just the simple law in Figure 10, which is like the copy rule, to characterise method call elimination.

Extra notation is used in Figure 10. Hereafter, we write $cds, N \rhd e : C$ to indicate that in the class N declared in cds, the expression e has static type C. To indicate that the typing holds inside the main command we use cds, **main** $\rhd e : C$. We also write $cds, N \rhd c = d$ to denote that the equation $c = d$ holds inside the class named N, in a context defined by the set of class declaration cds. The notation $pc[le/$**self**$]$ represents the parametrised command pc where every occurrence of **self** is replaced with le. We need an assumption $\{le \neq \textbf{null} \wedge le \neq \textbf{error}\}$ on the right hand side of Law 2 because, in case le is **null** or **error**, a method call $le.m(e)$ aborts. If the condition in the assumption holds it behaves like **skip**, otherwise as **abort**. Formally, $\{b\} \mathrel{\widehat{=}} \textbf{if } b \rightarrow \textbf{skip} \ [\,]\ \neg b \rightarrow \textbf{abort fi}$.

The semantics of ROOL is defined in [CN00] using weakest preconditions. All its laws, including those that we use here, have been proved sound with respect to this model [CN00]. Work reported in [Cor04, Dur05] and Appendix A provide an extensive set of laws. Relative completeness for the laws of ROOL has been addressed in [BSCC04].

Law 2. (method call elimination)
Consider that the following class declaration

```
class $C$ extends $D$
        $ads$
        meth $m \triangleq pc$
        $ops$
end
```

is included in cds, and $cds, A \rhd le : C$. Then
$cds, A \rhd le.m(e) = \{le \neq \textbf{null} \wedge le \neq \textbf{error}\};\ pc[le/\textbf{self}](e)$

provided

($\leftrightarrow$) (1) m is not redefined in cds and pc does not contain references to **super**; (2) all attributes which appear in the body pc of m are public.

Fig. 10. Example of a law of command

```
$cds_{RVM}$ •   var $PC$ : int;  $S, F$ : $Stack$   •
                    $S :=$ new $Stack$;  $F :=$ new $Stack$;
                    $PC := s$;
                    while $PC \geq s \wedge PC < f \rightarrow$ if $GCS$ fi end
                end
```

Fig. 11. The ROOL Interpreter ($cds_{RVM} \bullet I$)

3. Target machine

Here, we define a ROOL Virtual Machine (RVM) based on the Java Virtual Machine (JVM) [LY97]. Our definition is an interpreter-like ROOL program; it models a cyclic mechanism executing one instruction at a time (Figure 11).

As a valid program, the RVM consists of a sequence of class declarations cds_{RVM} followed by a main command I. The class declarations describe the RVM components. In every cycle, if the value of the program counter PC is still within the range of addresses from s to f where the program is stored, the bytecode instruction indicated by the value of PC is executed; it is a sequence of ROOL commands that simulates the effect of the instruction on the internal data structure. An example was presented in Section 1 and the general structure is described in Figure 11.

The abbreviation GCS depicts the stored program as a set of guarded commands $(PC = k) \rightarrow q$, where q is a bytecode instruction that is executed when PC takes the value k. The initial value of PC is the address s of the first instruction to be executed; the final address is f. The **while** is executed until PC reaches a value out of the interval determined by s and f. The body of the **while** uses the value of PC to select the instruction to be executed. All instructions modify PC. The set of guarded commands is an abstract representation of the target code.

The machine components are represented by the variables: PC (program counter), S (operand stack), M (store for variables), F (frame stack), *Cls* (data structure that records the hierarchy of classes), and CP (constant pool). The execution of an instruction changes (some of) these variables. The memory of the machine is implicitly represented by S, F, *Cls*, CP, and by the association of addresses in the range from s to f to the bytecodes.

Although most machines use relative branching to implement control flow, for simplicity we do not consider this issue here. The possibility of using relative addresses has been already discussed in [Sam97] and involves additional proof obligations. Clearly, our model of target machine could be more complex, nevertheless we would rather concentrate on the central contribution of our approach: the overall strategy to reduce the problem of designing a correct compiler for an object-oriented language to one of program refinement. Besides, when dealing with more realistic compilers, incremental extensions can be made to deal with additional features, since reuse of design and proofs arises as a consequence of abstraction and modularity, which are distinguishing characteristics of our reasoning framework.

The components *Cls*, M and CP are represented as global variables. *Cls* holds the essential information about class declarations in the source program. This variable has class type $CdsHierarchy$. Basically, it has one attribute: a sequence of objects of the class *ClassInfo*. Each of these objects has the following attributes: *cl*, which codifies the class name, and *subcls*, the sequence of immediate subclasses. The method $Instanceof$ declared in $CdsHierarchy$ is used to check whether an object in a given position in the sequence has a certain type.

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 addresses of the variables to their values. The symbol table Ψ maps the variables declared in the source program to addresses in the store M, so that $M[\Psi_x]$ holds the value of x. In the RVM, M is a free variable; it is a sequence of objects of type $Data$, a class that we describe later on.

The constant pool CP is a heterogeneous list of objects of three different types. An entry containing an integer is an object of type $DataInt$; boolean values are encapsulated in instances of $DataBool$; an entry containing an object of type $objData$ holds the RVM representation of a fresh object of a given class. The symbol table Φ maps elements declared in the source program to addresses in the constant pool CP, such that $CP[\Phi_C]$ holds an object of class C.

All values stored in the RVM are instances of the class $Data$. This class has two immediate subclasses: $PriData$ and $ObjData$. The class $PriData$ specialises into subclasses whose instances hold primitive values. For instance, an integer value is stored in an instance of $DataInt$, whose immediate superclass is $PriData$. The class $ObjData$ is used to represent objects. In this representation, the object attributes are either instances of $PriData$ or $ObjData$.

For simplicity, the current instruction set has support only for two primitive types: Integer and Boolean. To consider other primitive types, we have to expand the set of immediate subclasses of $PriData$, and extend the instruction set with the associated instructions that manipulate the corresponding primitive types.

Instruction set. The computation in the virtual machine is based on the operand stack, hence the majority of the bytecode instructions involves the operand stack S. 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. Since each compilation rule preserves semantics, we guarantee that the operands on the stack will be of the correct type for the operation.

Below, we give some examples of how the instructions of our virtual machine are defined. We assume that n stands for an address in M, k for an address in the sequence of bytecodes, and j for an index to an entry in CP.

Definition 2. (Instruction set - Miscellaneous)

$$\begin{array}{lcl}
nop & \overset{def}{=} & PC := PC + 1 \\
cmpeq & \overset{def}{=} & S.s := \langle(\textbf{new}\, DataBool;\ \ Info : (head(S.s) = head(tail(S.s))))\rangle ^\frown tail(tail(S.s)); \\
 & & PC := PC + 1 \\
ldc(j) & \overset{def}{=} & S.s := \langle CP[j]\rangle ^\frown S.s;\ \ PC := PC + 2 \\
load(n) & \overset{def}{=} & S.s := \langle M[n]\rangle ^\frown S.s;\ \ PC := PC + 2 \\
store(n) & \overset{def}{=} & M[n] := head(S.s);\ \ S.s := tail(S.s);\ \ PC := PC + 2 \\
aconstnull & \overset{def}{=} & S.s := \langle \textbf{null}\rangle ^\frown S.s;\ \ PC := PC + 1
\end{array}$$

$\diamond$

The instruction with opcode nop (no operation) has no effect, except for the program counter (PC) increment. The definition of $cmpeq$ uses an update expression to change the field $Info$. Two values are popped from the operand stack and compared for equality. If they are equal, then $true$ is pushed onto the stack, otherwise $false$ is pushed. The $head$ and $tail$ operators are standard, except that, when applied to the empty sequence, they give an arbitrary result.

The instruction with opcode ldc (load constant) has one argument (j) and pushes an integer constant onto the operand stack (S). Its argument j immediately follows the instruction with opcode ldc in the bytecode stream and represents a constant pool index to the location where the corresponding constant is stored.

Pushing a local variable onto the operand stack is done by the instruction with opcode $load$, and involves moving a value from the local variables list to the operand stack. Analogously, to pop a value of any type from the top of the stack to a local variable, the virtual machine uses the instruction with opcode $store$. The argument n is an index to a local variable. The instruction with opcode $aconstnull$ pushes a **null** value onto the operand stack.

The PC increment depends on the number of operands of the instruction. The more operands are needed, the more bytecodes are necessary to codify the instruction. For instance, when the instruction needs no operand, as in nop, the PC is incremented by 1 because one bytecode is enough to codify the opcode, whereas in the case of $load(n)$, an additional bytecode is necessary to represent the operand n, thus the PC is incremented by 2.

To deal with operators, we group them so that **uop** and **bop** stand for arbitrary unary and binary operators, respectively. They abstract arithmetic and relational operators. All of them use as operands objects of type $DataInt$ stored at the top of the operand stack S, but their outcome is expressed differently: when an arithmetic operation is performed, an object of type $DataInt$ is pushed onto the operand stack S, whereas in the case of relational operators, the pushed object has type $DataBool$. The definition for arithmetic operators is given below.

Definition 3. (Binary and Unary Operators)

$$
\begin{array}{lcl}
bop & \overset{def}{=} & S.s := \langle(\mathbf{new}\ DataInt;\ Info : (DataInt\, head(S.s)).Info\ \mathbf{bop} \\
 & & (DataInt\, head(tail(S.s))).Info)\rangle \frown tail(tail(S.s));\ \ PC := PC+1 \\
uop & \overset{def}{=} & S.s := \langle(\mathbf{new}\ DataInt;\ Info : \mathbf{uop}\ (DataInt\, head(S.s)).Info)\rangle \frown tail(S.s);\ \ PC := PC+1
\end{array}
$$

◇

Here, bop is an abbreviation for a sequence of ROOL commands that removes two values from the operand stack S, operates on them according to the corresponding binary operand **bop**, and places the result at the top of S. Since bop has no parameter, the PC is incremented by one. An instance of a binary operator is add, which is used in Figure 23.

Similarly to the bop abbreviation, uop pops one value from the operand stack S and applies the unary operand **uop** on it, pushing the resulting value on the top of S. Also, the PC is incremented by one.

A $goto$ instruction always branches. It causes execution to jump to the next instruction determined by the parameter k. The $cgoto$ is a conditional $goto$. If the value stored at the top of the operand stack is the boolean $true$, execution passes to the next instruction, otherwise execution continues at the specified instruction.

Definition 4. (Branch Instructions)

$$
\begin{array}{lcl}
goto(k) & \overset{def}{=} & PC := k \\
cgoto(k) & \overset{def}{=} & \mathbf{var}\ x : Boolean\ \ \bullet \\
 & & \quad x := (DataBool\ head(S.s)).Info;\ \ S.s := tail(S.s); \\
 & & \quad \mathbf{if}\ (x) \rightarrow PC := PC+2\ [] \neg(x) \rightarrow PC := k\ \mathbf{fi} \\
 & & \mathbf{end}
\end{array}
$$

◇

For creation and manipulation of objects, there are three instructions.

Definition 5. (Object creation and manipulation)

$$
\begin{array}{lcl}
new(j) & \overset{def}{=} & S.s := \langle CP[j]\rangle \frown S.s;\ \ PC := PC+2 \\
putfield(j) & \overset{def}{=} & S.s := \langle((ObjData\ head(S.s));\ att[j] : head(tail(S.s)))\rangle \frown tail(tail(S.s)); \\
 & & PC := PC+2 \\
getfield(j) & \overset{def}{=} & S.s := \langle(ObjData\ head(S.s)).att[j]\rangle \frown tail(S.s);\ \ PC := PC+2
\end{array}
$$

◇

The instruction $new(j)$ creates a new class instance. In the RVM, instead of traversing the representation of the source program class hierarchy, determining and recording the attributes present in the class indicated by j, our new instruction simply gets a copy of an object in the constant pool. This object is an instance of $Data$ and holds the RVM representation of the intended object whose type is indicated by the argument j. This argument is an index to a class entry in the constant pool holding an object that belongs to the concrete data space of the target machine. Once this object is obtained, it is pushed onto the operand stack. Since ROOL has a copy semantics, the object operations involve possible copying of complex structures. We further discuss how the issue of pointers can be handled in Section 5.

The instruction $putfield(j)$ selects and modifies the value of the attribute indicated by the entry of the constant pool indexed by j. When it is executed, two values are popped from the operand stack; the first is the object whose attribute is to be set, and the second is the value to be assigned. A copy of the modified object is pushed onto the operand stack. The execution of $getfield(j)$ pops from the operand stack the object whose attribute value is to be obtained. The argument j is an index into the constant pool that identifies the attribute. The instruction $getfield(j)$ gets the value of the attribute in the RVM representation of the object, and pushes it on the stack.

The instruction $instanceof$ tests if an object belongs to a type. The argument j is an integer given by the function Φ that identifies a class. The global variable Cls holds an image of the class hierarchy. From it, we can determine if the object belongs to a class C or any subclass of C using the method $instanceof$ of the class $CdsHierarchy$.

Definition 6. (Type testing)

$$\begin{array}{lll} instanceof(j) & \overset{def}{=} & \textbf{var}\ obj : objData, r : DataBool\ \bullet \\ & & \quad obj := head(S.s);\ S.s := tail(S.s); \\ & & \quad Cls.Instanceof(Obj, j, r); \\ & & \quad S.s := \langle r \rangle ^\frown S.s;\ PC := PC + 2 \\ & & \textbf{end} \end{array}$$

◇

Finally, we have four instructions to handle method calls. The instruction $save(o)$ pushes an object onto the frame stack F, whereas $restore(o)$ pops an object from F and assigns it to o. The instruction $invoke$ pushes the returning value of PC onto the frame stack F; afterwards a new value is assigned to PC, making the execution flow deviate to a position where the method body begins. When $return$ is executed, the value of PC is popped from the frame stack F. The execution flow returns to the next instruction after the invocation that originated the method call.

Definition 7. (Method invocation)

$$\begin{array}{lll} save(o) & \overset{def}{=} & F.s := \langle o \rangle ^\frown F.s \\ restore(o) & \overset{def}{=} & o := head(F.s);\ F.s := tail(F.s) \\ invoke(m) & \overset{def}{=} & PC := PC + 2;\ F.s := \langle (\textbf{new}\ DataInt;\ Info : PC) \rangle ^\frown F.s;\ PC := m \\ return & \overset{def}{=} & PC := (DataInt\ head(F.s)).Info;\ F.s := tail(F.s) \end{array}$$

◇

The RVM definition characterises the normal form adopted as the target of compilation: program is in normal form if it in the form shown in Figure 11. In the next section, we describe our strategy to reduce programs to normal form.

4. Compilation Process

In our approach, the design of a compiler is a constructive proof that an arbitrary program can be refined by a program in normal form. To carry out the compilation, we use provably correct reduction theorems as rewrite rules.

As already said, the compilation process is divided into five phases: Class Pre-compilation, Redirection of Method Calls, Simplification of Expressions, Control Elimination, and Data Refinement. The compilation follows this sequence; a change in this order is disallowed by side conditions on the reduction theorems of each phase. For each phase, a theorem establishes the expected outcome, and the main compilation theorem links the intermediate steps.

The reduction theorems are proved correct from the basic algebraic laws of ROOL and the compilation rules introduced in this section. We make a distinction between laws and rules. The basic laws assert general properties of the language constructors, whereas rules serve the special purpose of compilation. The correctness of the compiler follows from the correctness of each compilation rule (which relies on the soundness of the laws used in their proofs).

In Sections 4.1 to 4.5 we introduce the reduction theorem for a compilation phase and its associated rules. In the last section we present the proof of the main theorem. The proofs of all the compilation rules are in [Dur05]; the ROOL program introduced in Figure 8 is used as a running example to illustrate the compilation process.

4.1. Class Pre-Compilation

The outcome of this phase is summarised by the theorem below, proved in [Dur05]. It establishes that the basic laws applied in this phase are sufficient to end up with a program where method redefinitions are eliminated, all attributes are public, some attributes are moved up in the class hierarchy to allow elimination of method redefinitions, and every possible cast is introduced in each method body and in the main command.

Theorem 1. (Class Pre-compilation) Let $cds \bullet c$ be a program containing only executable constructs, as defined in Section 2, correct with respect to syntax and its static typing of expressions and statements. Then, there is a program $cds_{pc} \bullet c_{pc}$ such that $cds \bullet c \quad \sqsubseteq \quad cds_{pc} \bullet c_{pc}$ where (1) the main command c_{pc} differs from c only by the introduction of casts in all expressions; (2) the class declarations cds_{pc} are obtained from cds by changing the visibility of all attributes to public, and by eliminating all method redefinitions; and (3) all targets of calls and attribute accesses are cast with their static types. ◇

This compilation phase comprises three steps. The first one changes the visibility of the attributes, and the second introduces trivial casts. Both steps are necessary to avoid syntactic errors that could be originated later, when performing the next compilation phases. Finally, in the last step, if there is a redefined method in cds, laws of classes are applied to eliminate redefinitions; as a consequence, dynamic binding of methods and references to **super** are eliminated.

This is not the approach typically used in implementing dynamic binding. The advantage of this approach is that dynamic binding is handled in a purely algebraic way; this is an original contribution of our work. It allows us to separate concerns since we have separate laws to capture dynamic binding, calls via super, and method call elimination. Our proofs using the laws have given evidence that this is extremely convenient for reasoning.

4.1.1. Changing visibility of attributes

We need to guarantee that the body of a method in cds does not contain references to private or protected attributes, since otherwise an error can arise when we move the method declaration upwards in the class hierarchy. Therefore, we change the declarations of the attributes to make them all public. This is not a good practice from a software engineering point of view, but this does not change the behaviour of a complete program, and our aim is compilation rather than, for instance, refactoring [Opd92, Fow99, Cor04]. To perform these transformations, we use Laws 4 (change visibility: from protected to public) and 5 (change visibility: from private to public). These and all the other laws referenced here are listed in Appendix A and proved in [Cor04]. We discuss and exemplify the proofs in Section 4.4.

In order to illustrate the compilation process, we reduce the example program in Figure 8 to normal form, showing step by step which transformations are imposed by each compilation phase. Applying Laws 4 (change visibility: from protected to public) and 5 (change visibility: from private to public), we change the declarations of the attributes dir and len in the class $Step$, and of the attribute $previous$ in $Path$ to make them public (see Figure 12).

4.1.2. Introducing trivial casts

Applying Law 9 (cast introduction in expressions), we introduce type casts to produce a uniform program text in which all targets of calls and attribute accesses are cast with their static types. Introducing casts has no effect. During the elimination of method redefinitions, however, casts play a fundamental role because, in the presence of **self**, when we move methods upwards, the behaviour of the program can be modified, if **self** is not properly cast.

In Figure 12, we present the intermediate program obtained from the source program, after the application of the transformations to change the visibility of the attributes and introduce trivial casts. In our example, observe that the occurrence of **super** in $getLength$ as defined in $Path$ is cast to $Step$.

4.1.3. Elimination of method redefinition

ROOL supports method redefinition but, as opposed to Java, not overloading. This is exploited in Law 1 (move redefined method to superclass), which states that we can merge a method declaration and its redefinition into a single declaration in the superclass. In order to choose the appropriate behaviour, the resulting method uses type tests. Before moving a method up in the class hierarchy in this way, we have to eliminate references to **super**, otherwise we could end up referencing nonexistent methods. The elimination of **super** relies on Law 6 (introduce method redefinition), which is basically a version of the copy rule for ordinary procedures.

As an example, consider the method $getLength$ (Figure 12), which is first declared in the class $Step$ and redeclared in the class $Path$. Applying Law 6 (introduce method redefinition), we obtain the class $Path$ described in Figure 13; in the definition of $getLength$ the **super** call is replaced by the body of this method in $Step$.

To merge a redefined method into a single declaration in the superclass, we apply Law 1 (move redefined method to superclass). Law 7 (move original method to superclass) allows us to move up in the class hierarchy a method declaration that is not a redefinition. Hence, we can transform the classes $Path$ and $Step$, moving the methods in $Path$ to the class $Step$, as shown in Figure 14. Now, we have a single declaration for $getLength$.

4.2. Redirection of Method Calls

In this phase, we introduce a class L, which includes elements that later allow the elimination of the source method declarations. The introduction of L allows arguments of method calls to be passed using the operand stack S, adapting

```
class  Step
       pub $dir, len$ : int;
       meth $setDirection \triangleq$ (val $d$ : int; • $(Step$ self$).dir := d$ )
       meth $setLength \triangleq$ (val $l$ : int; • $(Step$ self$).len := l$ )
       meth $getLength \triangleq$ (res $l$ : int; • $l := (Step$ self$).len$ )
end
class  $Path$ extends $Step$
       pub $previous : Path$;
       meth $addStep \triangleq$ (val $d, l$ : int •
          $(Path$ self$).previous := (Path$ self$)$; $(Path$ self$).setDirection(d)$; $(Path$ self$).setLength(l)$ )
       meth $getLength \triangleq$ (res $l$ : int •
          var $aux$ : int •
               if  $((Path$ self$).previous <>$ null$)$ $\rightarrow (Path$ self$).previous.getLength(aux)$
                 [] $((Path$ self$).previous =$ null$)$ $\rightarrow aux := 0$
               fi; $(Step$ super$).getLength(l)$; $l := l + aux$;
          end)
end
•      var $p$ : $Path$ • $p :=$ new $Path$;
          $p.addStep(north, l_1)$; $p.addStep(east, l_2)$; $p.addStep(south, l_3)$; $p.addStep(west, l_4)$;
          $p.getLength(out)$;
       end
```

Fig. 12. Program obtained after changing the visibility of the attributes and introducing trivial casts

```
class $Path$ extends $Step$
   pub $previous : Path$;      meth $addStep \triangleq$ (val $d, l$ : int •
      $(Path$ self$).previous := (Path$ self$)$; $(Path$ self$).dir := d$; $(Path$ self$).len := l$)
   meth $getLength \triangleq$ (res $l$ : int •
      var $aux$ : int •
           if $((Path$ self$).previous =$ null$)$ $\rightarrow aux := 0$
             [] $(Path$ self$).previous <>$ null$)$ $\rightarrow (Path$ self$).previous.getLength(aux)$
           fi; (res $l$ : int; • $l := (Step$ self$).len$ )$(l)$; $l := l + aux$
      end)
end
```

Fig. 13. Class $Path$ without references to **super**

the invocation of methods to work in the way it is defined in the RVM. As explained in Section 3, our RVM is a stack-based machine; instructions receive values and yield results using an operand stack.

Each method lm declared in L plays the role of a method m declared in cds; the body of lm contains a copy of the body of m. At this stage, redefinitions have already been eliminated. When a method lm is created in L, it contains the parametrised command pc that defines the method m, declared in cds. Therefore, before we apply the rules to redirect the method calls, the body of a method lm might have calls to methods declared in cds, including recursive calls. For instance, in Figure 17 observe that there is a call to the method $getLength$ within the body of the method $lgetLength$, declared in L. After introducing the class L, all method calls are redirected to the corresponding methods declared in class L, so that the method declarations in cds become useless and can, therefore, be eliminated.

In this step, we also introduce the class declarations cds_{RVM}. They not referenced by the source program, but some commands that are introduced in L and in the main command c reference methods, attributes, and classes defined in cds_{RVM}. For this reason, when we introduce the class L, the need to introduce cds_{RVM} arises.

The outcome of this phase is summarised by Theorem 2. It establishes that the compilation rules applied in this phase are sufficient to end up with a program where all calls are to methods in L. The proof can be found in [Dur05].

Theorem 2. (Redirection of method calls) Let $cds_{pc} \bullet c_{pc}$ be a program where all attributes in cds_{pc} are public, every

```
class $Step$
  pub $dir$, $len$ : int;
  meth $getLength \triangleq$ (res $l$ : int; •
      if (self is $Path$) $\rightarrow$ $l := (Step$ self$).len$
          var $aux$ : int •
              if $((Path$ self$).previous <>$ null$)$ $\rightarrow$ $(Path$ self$).previous.getLength(aux)$
                  [] $((Path$ self$).previous =$ null$)$ $\rightarrow aux := 0$
              fi; (res $l$ : int; • $l := (Step$ self$).len\ )(l)$; $l := l + aux$
          end
      [] $\neg$(self is $Path$) $\rightarrow$ $l := (Step$ self$).len$
      fi)
end
class $Path$ extends $Step$
  pub $previous$ : $Path$;
  meth $addStep \triangleq$ (val $d, l$ : int •
    $(Path$ self$).previous := (Path$ self$)$; $(Path$ self$).dir := d$; $(Path$ self$).len := l)$
end
```

Fig. 14. Method *getLength* as a method without redefinitions

```
meth $lm \triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ •
                 var $o$ : $Object$; $v$ : $T$ •
                   $Pop(S_{in}, o)$; $Pop(S_{in}, v)$; $pc[o/$self$](v)$; $Push(S_{in}, o)$; $S_{out} := S_{in}$
                 end)
```

Fig. 15. The pattern of a method with value parameters in the class L

possible casts are present, and no methods have redefinitions, then there is a program cds_{RVM} cds_{rmc} $L \bullet$ c_{rmc}, such that $cds_{pc} \bullet c_{pc} \sqsubseteq cds_{RVM}$ cds_{rmc} $L \bullet c_{rmc}$, where cds_{rmc} contains only the attribute declarations of cds, and the calls in c_{pc} to methods in cds are replaced in c_{rmc} with calls to the corresponding methods in L. $\diamond$

The previous compilation phase establishes the conditions that allow the definition of L. The way the class L is organised is explained in Section 4.2.1. In Section 4.2.2, we introduce rules for the Redirection of Method Calls. These rules rely on the type of parameter passing used. We modify method calls so that their arguments are pushed onto and popped from the operand stack. When a redirected method is invoked, a copy of its target object is also pushed onto the stack. When the execution is completed, the results are reflected in the operand stack, including a copy of the target object, possibly modified as a result of the execution of the method body.

4.2.1. The class L

The introduction of the class L is justified by Law 3 (class elimination). In principle, any class that is not referenced in the program can be introduced. For each method m declared in cds, there is a corresponding method lm in L that simulates the behaviour of m. Every method in L whose parameters are passed by value shares a common pattern, which is described in Figure 15, whereas those methods whose parameters are passed by result share the pattern depicted in Figure 16. Methods with a combination of value and result parameters can be handled in a similar way.

In the patterns, the formal parameters are the operand stacks S_{in} and S_{out}; o is the target of the method call; v represents the list of value arguments, whereas r denotes the list of result arguments. Since ROOL does not include value-result parameters (nor sharing), two stacks are needed: S_{in} to receive the arguments, and S_{out} to return the results. In Figure 15, the argument v is passed by value to the parametrised command pc, simulating the expected behaviour of a call to a method having o as the target object and v as the value argument. In the case of Figure 16, the argument r is passed by result to pc, imitating the behaviour of a call with a result argument r.

For methods that are not redefined, the parametrised command in lm is the same we find in the definition of m, except for the replacement of **self** by the variable o. This does not introduce any syntactic errors because, as a result of the previous phase of compilation, the occurrences of **self** are guaranteed to be cast.

```
meth $lm$ $\triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ $\bullet$
    var $o$ : $Object$; $r$ : $T$ $\bullet$
        $Pop(S_{in}, o)$; $pc[o/$self$](r)$; $Push(S_{in}, r)$; $Push(S_{in}, o)$; $S_{out} := S_{in}$
    end)
```

Fig. 16. The pattern of a method with result parameters in the class L

```
meth $lgetLength$ $\triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ $\bullet$
    var $o$ : $Object$; $l$ : int $\bullet$ $Pop(S_{in}, o)$;
      (res $l$ : int; $\bullet$
      if ($o$ is $Path$) $\rightarrow$
        var $aux$ : int $\bullet$
          if (($Path\, o$).$previous$ = null) $\rightarrow$ $aux := 0$
             [] (($Path\, o$).$previous$ <> null) $\rightarrow$ ($Path\, o$).$previous.getLength(aux)$
          fi;
          (res $l$ : int; $\bullet$ $l := (Step\, o).len$ )($l$); $l := l + aux$;
        end
      [] $\neg$($o$ is $Path$) $\rightarrow$ $l := (Step\, o).len$
      fi) ($l$);
      $Push_i(S_{in}, l)$; $Push(S_{in}, o)$; $S_{out} := S_{in}$
    end)
```

Fig. 17. Method $lgetLength$ declared in L

As an example, we consider $getLength$. In the first compilation phase, the attributes became public, all possible casts were introduced, and its redefinition was eliminated. Following the pattern in Figure 15, there is a method $lgetLength$ in L; its body is described in Figure 17. In a similar way, for each method in cds, there is a corresponding method in L; in summary, there is a one-to-one correspondence between all methods in cds and the methods in L.

In the definition of $lgetLength$ in Figure 17, there is a call to the method $getLength$. We could have used a recursive call to $lgetLength$ instead. Having method definitions in L with precisely the same parametrised commands as the original methods, however, is convenient when proving that the method in L preserves the behaviour of the corresponding method in cds. Later, we deal with method call elimination in general, and then with the elimination of method declarations; therefore, eliminating the calls to the methods of cds in L causes no extra overhead. In the next section, we show how we redirect method calls in order to make all method declarations in cds useless.

4.2.2. *Rules*

The next two rules define the transformations related to the Redirection of Method Calls to methods in L: there is a rule for each parameter passing mechanism. Multiple parametrisation is handled by combining their effects. In order to make the rules clearer and concise, we introduce abbreviations for sequences of commands over the operand stack.

Definition 8. (Abbreviations – non-primitive variables through the operand stack)

$$\begin{aligned}
Push(S, o) &\stackrel{def}{=} S.s := \langle o \rangle \frown S.s \\
Pop(S, o) &\stackrel{def}{=} o := head(S.s);\ \ S.s := tail(S.s) \\
Bop(S) &\stackrel{def}{=} S.s := \langle head(S.s)\,\mathbf{bop}\,head(tail(S.s))\rangle \frown tail(tail(S.s)) \\
Uop(S) &\stackrel{def}{=} S.s := \langle \mathbf{uop}\,head(S.s)\rangle \frown tail(S.s) \\
Putfd(S, C.t) &\stackrel{def}{=} S.s := \langle(((C)head(S.s);\ t : head(tail(S.s))))\rangle \frown tail(tail(S.s)) \\
Getfd(S, C.t) &\stackrel{def}{=} S.s := \langle (C)head(S.s).t\rangle \frown tail(S.s)
\end{aligned}$$

◇

In $Bop(S)$ and $Uop(S)$, **bop** and **uop** stand for arbitrary binary and unary operators. The abbreviation $Putfd(S, C.t)$

removes two elements from the operand stack S. The first is the object of class C whose attribute t is to be set; the second is the value to be assigned. A copy of the modified object is pushed onto the operand stack.

The abbreviation $Getfd(S, C.t)$ pops from the operand stack the object whose attribute value is to be obtained. This attribute is named t and its value is pushed onto the operand stack.

Definition 9. (Abbreviations — integer variables through the operand stack)

$$\begin{array}{lcl}
Push_i(S,x) & \overset{def}{=} & S.s := \langle(\textbf{new}\ DataInt;\ Info : x)\rangle \frown S.s \\
Pop_i(S,x) & \overset{def}{=} & x := (DataInt\ head(S.s)).Info;\ \ S.s :=\ tail(S.s) \\
Bop_i(S) & \overset{def}{=} & S.s := \langle(DataInt\ head(S.s)).Info\ \textbf{bop}\ (DataInt\ head(tail(S.s))).Info\rangle \frown tail(tail(S.s)) \\
Uop_i(S) & \overset{def}{=} & S.s := \langle\textbf{uop}\ (DataInt\ head(S.s)).Info\rangle \frown tail(S.s)
\end{array}$$

◇

Definition 10. (Abbreviations — boolean variables through the operand stack)

$$\begin{array}{lcl}
Push_b(S,x) & \overset{def}{=} & S.s := \langle(\textbf{new}\ DataInt;\ Info : x)\rangle \frown S.s \\
Pop_b(S,x) & \overset{def}{=} & x := (DataBool\ head(S.s)).Info;\ \ S.s :=\ tail(S.s) \\
Bop_b(S) & \overset{def}{=} & S.s := \langle(DataBool\ head(S.s)).Info\ \textbf{bop} \\
 & & (DataBool\ head(tail(S.s))).Info\rangle \frown tail(tail(S.s)) \\
Uop_b(S) & \overset{def}{=} & S.s := \langle\textbf{uop}\ (DataBool\ head(S.s)).Info\rangle \frown tail(S.s) \\
Is(S,C) & \overset{def}{=} & S.s := \langle(DataBool;\ info : (head(S.s)\ \textbf{is}\ C))\rangle \frown Tail(S.s)
\end{array}$$

◇

There are other versions of these abbreviations for each primitive type. In Definitions 9 and 10, we present those for the integer and boolean types. When manipulating primitive values through the operand stack we encapsulate them into instances of classes declared in cds_{RVM}. For instance, integer values are encapsulated in instances of $DataInt$. The abbreviation $Is(S, C)$ removes an element from the top of the operand stack S and tests if it has type C. The result of this test is encapsulated into an instance of $DataBool$ and pushed onto S.

For conciseness, in the rules presented from here on, we consider mainly the non-primitive types, since versions of the rules for primitive types are similar. Rule 1 deals with method calls $((C)le).m(x)$ with result parameter x.

Rule 1. (Redirecting a call with result parameter)
Consider that the declaration **class** C **extends** $D\ ads$ **meth** $m \triangleq pc\ ops$ **end** is included in cds, and $cds, A \rhd\ le : C$, and the following method declaration corresponding to m is in the class L.

```
cds_RVM cds, L ▷   meth lm ≜ (val S_in : Stack; res S_out : Stack •
                       var o : Object; r : T •
                          Pop(S_in, o); pc[o/self](r);
                          Push(S_in, r); Push(S_in, o); S_out := S_in
                       end)
```

Then

```
cds_RVM cds L, N ▷
      ((C)le).m(x)  ⊑  {le ≠ null ∧ le ≠ error}
                        var S : Stack; V : L •
                              S := new Stack; V := new L;
                              Push(S, le); V.lm(S, S);
                              Pop(S, le); Pop(S, x);
                        end
```

provided

$(\rightarrow)$ (1) m is declared in cds and has a result parameter; (2) S and V are fresh names; (3) all attributes referenced by lm are public and declared in cds; (4) N is either **main** or L. ◇

On the left-hand side of the refinement, as already explained we use $cds_{RVM}\ cds$ to stand for the union of the class declarations that describes the RVM and those in cds. On the right-hand side, the assertion requires le to be different from **null** and **error**, since the call on the right-hand side aborts in such cases. A variable block declares the operand stack S and a variable V of class L. The introduction of V is necessary because ROOL does not allow static methods. New objects are created to initialise S and V, and then le is pushed onto S. After calling lm, the value of the result argument is popped from S and assigned to x; the resulting object is popped from the stack and assigned to le.

The next rule deals with the simplification of a method call with a value parameter.

Rule 2. (Redirecting a call with value parameter)
Consider that the declaration **class** C **extends** $D\ ads$ **meth** $m \mathrel{\widehat{=}} pc\ ops$ **end** is included in cds, and $cds, A \rhd le : C$, and the following method declaration corresponding to m is in the class L.

```
cds_RVM cds, L ▷   meth lm ≙ (val S_in : Stack; res S_out : Stack •
                       var o : Object;  v : T •
                          Pop(S_in, o);  Pop(S_in, v);  pc[o/self](v);
                          Push(S_in, o);  S_out := S_in
                       end)
Then
 cds_RVM cds L, N ▷
        ((C)le).m(x)   ⊑   {le ≠ null ∧ le ≠ error}
                           var S : Stack;  V : L •
                                 S := new Stack;  V := new L;
                                 Push(S, x);  Push(S, le);  V.lm(S, S);
                                 V.lm(S, S);  Pop(S, le)
                           end
provided
```

($\rightarrow$) (1) m is declared in cds and has a value parameter; (2) S and V are fresh names; (3) all attributes referenced by lm are public and declared in cds; (4) N is either **main** or L.

◇

The above rule is similar to Rule 1, except for the role of the argument x. It is a value argument, hence it is pushed onto the operand stack before lm is called. As a desired effect of applying Rules 1 and 2 to redirect calls, the method declarations in cds become useless. Based on Law 8 (method elimination), they can then be eliminated.

In our example, calls to the method $getLength$ have a parameter passed by result. When the call $p.getLength(out)$ in the main command is redirected using Rule 1, the following variable block is obtained.

```
var S : Stack;  V : L • S := new Stack;  V := new L;
          Push(S, p);  V.lgetLength(S, S);  Pop(S, p);  Pop_i(S, out)
end
```

The method bodies in L might have calls to methods declared in cds, as the call to $getLength$ that appears in $lgetLenght$, in Figure 17. It occurs because each method lm in L contains a copy of the body of the method m in cds, hence m might have recursive calls or calls to other methods in cds.

As a result of applying Rules 1 and 2, for each method call in the main command and in the class L, a similar variable block is introduced; they are further transformed in the next phase of compilation to expand their scope. When we expand the scope of the variable blocks, unnecessary assignments to initialise the variables S and V become evident. Additional transformation is needed to eliminate them; this simplification is also the objective of the next phase of compilation. In this phase only method calls in L and in the main command are affected.

The method $getLength$ uses recursion to find the length of a path. The redirection of a recursive method call does not incur any overhead. Any recursion is embedded in calls to the associated method lm in L. As an example, we consider the call $(Path\ o.previous).getLength(aux)$ in Figure 17; when we rewrite it, we obtain the following.

```
var S : Stack;  V : L • S := new Stack;  V := new L;
      Push(S, o.previous);  V.lgetLength(S, S);  Pop(S, o.previous);  Pop_i(S, aux)
end
```

The resulting Class L obtained in this compilation phase appears in Figure 18.

```
class $L$
  meth $laddStep \triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ $\bullet$
      var $o$ : $Object$; $x, y$ : int $\bullet$ $Pop(S_{in}, o)$; $Pop_i(S_{in}, x)$; $Pop_i(S_{in}, y)$;
        (val $d, l$ : int $\bullet$ $(Path\ o).previous := (Path\ o)$;
          $(Path\ o).dir := d$; $(Path\ o).len := l$
        end)$(x, y)$; $Push(S_{in}, o)$; $S_{out} := S_{in}$
      end)
  meth $lgetLength \triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ $\bullet$
      var $o$ : $Object$; $x$ : int $\bullet$ $Pop(S_{in}, o)$; (res $l$ : int; $\bullet$
        if ($o$ is $Path$) $\rightarrow$
          var $aux$ : int $\bullet$
            if $((Path\ o).previous =$ null) $\rightarrow aux := 0$
            [] $((Path\ o).previous <>$ null) $\rightarrow$
              var $S$ : $Stack$; $V$ : $L$ $\bullet$ $S :=$ new $Stack$;
                $V :=$ new $L$; $Push(S, o.previous)$;
                $V.lgetLength(S, S)$; $Pop(S, o.previous)$; $Pop_i(S, aux)$
              end
            fi; (res $l$ : int; $\bullet$ $l := (Step\ o).len$ )$(l)$; $l := l + aux$
          end
        [] $\neg$($o$ is $Path$) $\rightarrow l := (Step\ o).len$
        fi) $(x)$; $Push_i(S_{in}, x)$; $Push(S_{in}, o)$; $S_{out} := S_{in}$
      end)
end
```

Fig. 18. Class L generated in the phase of Redirection of Method Calls

4.3. Simplification of Expressions

The main objective of this phase is to eliminate nested expressions in assignments and guards. We also eliminate parametrised commands, and transform conditionals with arbitrary guards into an *if-then-else* form. The expected outcome is stated by Theorem 3, whose proof can be found in [Dur05].

Theorem 3. (Simplification of Expressions)
Let cds_{RVM} cds_{rmc} L $\bullet$ c_{rmc} be a program, then there is a program cds_{RVM} cds_{rmc} L_s $\bullet$ c_s such that

$$cds_{RVM}\ cds_{rmc}\ L \bullet c_{rmc} \sqsubseteq cds_{RVM}\ cds_{rmc}\ L_s \bullet c_s$$

where c_s has the form **var** $v : T \bullet q$ **end** and methods declared in L_s have the following form

$$\textbf{meth}\ lm \triangleq (\textbf{val}\ S_{in} : Stack;\ \textbf{res}\ S_{out} \bullet \textbf{var}\ w_m : T \bullet p_m\ \textbf{end})$$

Furthermore, c_s and L_s are such that: (i) p_m and q have no local declarations; (ii) each assignment operates through the operand stack; (iii) each boolean expression is a variable or its negation; (iv) there is no parametrised command; and (v) every conditional is in *if-then-else* form. ◇

The objective is to format the commands into patterns closely related to the ones used to define the instructions. This theorem imposes no restriction on cds_{rmc} or L because it is only concerned with expressions.

Basically, the task of eliminating nested expressions involves rewriting assignments and boolean expressions in the class L and in the main command c. Since new variables are introduced, we apply basic laws to expand the scope of variable blocks. Besides rewriting assignments and simplifying expressions, we eliminate parametrised commands, simplify guards and conditionals, and eliminate unnecessary sequences of $Push$ and Pop. The elimination of parametrised commands is the first step. When a parametrised command is eliminated, a variable block is introduced.

4.3.1. Eliminating parametrised commands and parametrised recursion

In order to eliminate parametrised commands, such as those appearing in each method declared in L, we rely on Laws 25 (Pcom elimination-res) and 24 (Pcom elimination-val).

```
**class** $L$
  **meth** $laddStep \triangleq$ (**val** $S_{in}$ : $Stack$; **res** $S_{out}$ : $Stack$ $\bullet$
      **var** $o : Object;\ x, y :$ **int** $\bullet$ $Pop(S_{in}, o);\ Pop_i(S_{in}, x);\ Pop_i(S_{in}, y);$
        $(Path\ o).previous := (Path\ o);$
        $(Path\ o).dir := x;\ (Path\ o).len := y$
      **end**)
  **meth** $lgetLength \triangleq$ (**val** $S_{in}$ : $Stack$; **res** $S_{out}$ : $Stack$ $\bullet$
      **var** $o : Object;\ x :$ **int** $\bullet$ $Pop(S_{in}, o);$
        **if** ($o$ **is** $Path$) $\rightarrow$
          **var** $aux$ : **int** $\bullet$
            **if** $((Path\ o).previous =$ **null**$)\ \rightarrow aux := 0$
            $[]\ ((Path\ o).previous <>$ **null**$)\ \rightarrow$
                **var** $S : Stack;\ V : L$ $\bullet$ $S :=$ **new** $Stack;\ V :=$ **new** $L;$
                  $Push(S, o.previous);\ V.lgetLength(S, S);$
                  $Pop(S, o.previous);\ Pop_i(S, aux)$
                **end**
            **fi**; $x := (Step\ o).len;\ x := x + aux$
          **end**
        $[]\ \neg(o$ **is** $Path) \rightarrow x := (Step\ o).len$
        **fi**; $Push_i(S_{in}, x);\ Push(S_{in}, o);\ S_{out} := S_{in}$
      **end**)
**end**
```

Fig. 19. Class L after the elimination of parametrised commands

Further manipulation is needed to simplify the resulting command. To do so, we use Laws 23 (; := substitution), 12 (**var**-; right dist), 14 (**var** - := final value), and 11 (**var** elim). Law 23 (; := substitution) states that if the value of a variable is known, we can replace the occurrences of this variable in an expression with that value. Law 12 (**var**-; right dist) establishes that if the right argument of a sequential composition declares the variable x, then the scope can be extended to the left component, provided that it does not interfere with the other variables with the same name. Law 14 (**var** - := final value) says that an assignment to a variable just before the end of its scope is irrelevant. Finally, Law 11 (**var** elim) allows us to eliminate a declared variable that is never used.

Parametrised recursive calls are eliminated using a strategy similar to that presented in [Sam97]. A recursive method lm is defined with the use of the command **rec** $X \bullet c$ **end**. We now explain how we perform this transformation.

The intuitive interpretation of the recursive command is that of *recursive unfoldings* [BvW98]. In the execution of c, whenever X is encountered, it is equivalent to the whole command **rec** $X \bullet c$ **end**. Consider a parametrised recursive method lm declared in L, having in its body a code fragment with a recursive call, as follows.

meth $lm \triangleq$ (**val** S_{in} : $Stack$; **res** S_{out} : $Stack$ $\bullet \ldots o.lm(S, S) \ldots)$

Then, the method body is replaced with a recursive command, and each parametrised recursive call, such as the above $o.lm(S, S)$, is replaced by the code fragment with a parameterless recursive call to X, as stated next.

```
**meth** $lm \triangleq$ **rec** $X \bullet$   (**val** $S_{in}$ : $Stack$; **res** $S_{out}$ : $Stack$ $\bullet$
                        $\ldots$
                        **dvar** $S_{in}, S_{out} : Stack;\ S_{in} := S;\ X;\ S := S_{out};$ **dend** $S_{in}, S_{out}$
                        $\ldots)$
```

A non-recursive call gives the initial values of S_{in} and S_{out}. After that, when a parameterless recursive call is executed, the initial value of S_{in}, which is the value parameter, is given by S. Similarly, when returning from a call, the value of S_{out}, which is the result parameter, is copied to S. To achieve the same behaviour for every recursive call, the use of dynamic declarations is essential. They simulate an abstract stack that handles parameters in the way they are implemented in practice. For each recursive call, the value of the arguments are stored in an activation record [ASU85] in a run time stack; this stack is also used to store local variables.

Our target machine has a run time stack F. The following rule allows us to introduce F, replacing the implicit use

```
meth $lgetLength \triangleq$ (val $S_{in}$ : $Stack$; res $S_{out}$ : $Stack$ •
  var $F : Stack$ •
    $F :=$ new $Stack$;
    rec $X$ •
      var $x, aux$ : int; $o : Object$; $x_1, x_2 : Boolean$; $S : Stack$; $V : L$ •
        $Pop(S_{in}, o)$; $S :=$ new $Stack$; $V :=$ new $L$;
        $Push(S, o)$; $Is(S, Path)$; $Pop_b(S, x_1)$;
        $Push(S, o)$ $Getfd(S, Path.previous)$; $Push(S,$ null$)$;
        $cmpeq(S)$; $Pop_b(S, x_2)$
        if $x_1 \rightarrow$
            if $x_2 \rightarrow Push(S, 0)$; $Pop(S, aux)$
                $Push(S, o)$; $Getfd(S, Path.previous)$;
            [] $\neg(x_2) \rightarrow Push(S, 0)$; $Pop(S, aux)$
                $Push(S, o)$; $Getfd(S, Path.previous)$;
                $Push(F, S_{in})$; $Push(F, S_{out})$;
                $S_{in} := S$; $X$; $S_{out} := S$;
                $Push(F, S_{out})$; $Push(F, S_{in})$;
                $Push(S, o)$; $Putfd(S, Path.previous)$;
                $Pop(S, o)$; $Pop_i(S, aux)$
            fi;
            $Push(S, o)$; $Getfd(S, Step.len)$; $Pop(S, x)$;
            $Push(S, x)$; $Push(S, aux)$; $Add(S)$; $Pop(S, x)$
        [] $\neg(x_1) \rightarrow Push(S, o)$; $Getfd(S, Step.len)$; $Pop(S, x)$
        fi;
      end;
      $Push_i(S_{in}, x)$; $Push(S_{in}, o)$; $S_{out} := S_{in}$
    end
  end)
end
```

Fig. 20. Method $lgetLength$ after the third compilation phase

of a stack with explicit calls to the stack methods $Push$ and Pop.

Rule 3. (Stack implementation — Parameters of recursion)

$$\begin{array}{l} \textbf{rec } X \bullet (\textbf{val } x : T;\ \textbf{res } y : T \bullet p) \\ \sqsubseteq \\ \textbf{val } x : T;\ \textbf{res } y : T \bullet (\textbf{var } F : Stack \bullet \textbf{rec } X \bullet p[(\textbf{dpar } x, y : T \bullet X)/X]\ \textbf{end}) \end{array}$$

where

$$(\textbf{dpar } x, y : T \bullet X)(s, t) \overset{def}{=} Push(F, x);\ Push(F, y);\ x := s;\ X;\ t := y;\ Pop(F, y);\ Pop(F, x) \qquad \diamond$$

Here, we use an explicit stack to represent the way parameters of a recursive command are implemented in practice.

In our example, the only method that requires the elimination of parametrised recursion is $lgetLength$. In Figure 20, we present the transformed version of this method. At this point, in the body of $lgetLength$, there are no calls of the form $o.lgetLength(S, S)$. The same effect is obtained with the recursive command. Whenever the recursive variable X is encountered, the recursive unfolding simulates the recursive call. The parametrised recursion in the method $lgetLength$ is eliminated with the introduction of the recursive command **rec** $X \bullet c$ **end**, as presented in Figure 20.

4.3.2. Simplifying Guards

The boolean expressions appearing as conditions in **while** and **if** commands may be arbitrarily nested. They need to be simplified to match the patterns used to reduce these commands to normal form in the Control Elimination phase.

The next compilation rule shows how the conditions of an **if** command can be simplified.

Rule 4. (Conditions of **if**)
If x_i does not occur in b_i nor in p_i, for $1 \leq i \leq n$, then

$$\begin{array}{lll} \textbf{if} \, []_{\langle 1 \leq\ i\ \leq\ n \rangle}\ b_i\ \rightarrow\ p_i\ \textbf{fi} & \sqsubseteq & \textbf{var}\quad x_1, \ldots, x_n : \mathit{boolean} \bullet \\ & & \qquad x_1 := b_1;\ \ldots;\ x_n := b_n; \\ & & \qquad \textbf{if} \, []_{\langle 1 \leq\ i\ \leq\ n \rangle}\ x_i\ \rightarrow\ p_i;\ \textbf{fi}; \\ & & \textbf{end} \end{array}$$

◇

The conditions in an **if** are reduced to single variables x_i whose values are given by assignments. The simplification of the expressions b_i are performed using the rules related to assignment, presented in Section 4.3.4. In our example, the boolean expression (o **is** Step) is assigned to a boolean variable x_2; then x_2 replaces the expression in the conditional. For an iteration, the approach to simplification is similar, but in this case we have just one guard.

4.3.3. Transforming Conditionals

In the next compilation phase, that is, Control Elimination, when reducing a conditional we need to calculate the location of each guarded command in the bytecode stream. A conditional in an *if-then-else* style always has two guarded commands, and hence it is simpler to deal with this form than with conditionals with an arbitrary number of guarded commands. Besides, an *if-then-else* conditional always requests a fixed number of branch instructions when it is rewritten in a bytecode style in our normal form. Therefore, the application of Law 19 (**if** - Conditional in the if-then-else style) and Law 20 (**if** - Conditional without guarded commands) reduces conditionals to an if-then-else form, and therefore allows us to have simpler compilation rules to deal with conditionals.

4.3.4. Rewriting Assignments

Considering that the RVM is a stack-based machine, the assignments must be rewritten in order to operate exclusively on the operand stack. The following compilation rule defines how an assignment is rewritten. In our strategy, assignments to RVM variables, such as S and V, are not affected by these rules.

Rule 5. (Assignment to a variable)
If S does not occur in e or f

$$(x := e)\ =\ \textbf{var}\ S : \mathit{Stack};\ \bullet\ S := \textbf{new}\ \mathit{Stack};\ \mathit{Push}(S, e);\ \mathit{Pop}(S, x)\ \textbf{end}$$

◇

In the above rule, for each assignment, a block declaring S is introduced, S is initialised, e is pushed onto the operand stack, and then the value is popped from the top of the stack and assigned to x. The expression e may be arbitrarily nested; we need to simplify it further to achieve a form in which e consists of a variable, an attribute selection, or a constant. The next rule handles $\mathit{Push}(S, e)$, when e is a binary expression.

Rule 6. (Binary operator)

$$\mathit{Push}(S, e\ \textbf{bop}\ f)\ \ \sqsubseteq\ \ \mathit{Push}(S, e);\ \mathit{Push}(S, f);\ \mathit{Bop}(S)$$

where $\mathit{Bop}(S)$ represents the application of an arbitrary binary operator to the two topmost elements of the operand stack; see Definition 8. ◇

The nested expression in the method call $\mathit{Push}(S, e\ \textbf{bop}\ f)$ is replaced with a sequence of method calls which first loads the subexpression e, then loads the subexpression f, and finally performs the **bop** operation. Similarly, the following rule deals with a call $\mathit{Push}(S, e)$, whose argument is an application of a unary operator.

Rule 7. (Unary operator)

$$\mathit{Push}(S, \textbf{uop}\ e)\ \ \sqsubseteq\ \ \mathit{Push}(S, e);\ \mathit{Uop}(S)$$

where $\mathit{Uop}(S)$ represents the application of an arbitrary unary operator to the element at the top of the operand stack; see Definition 8. ◇

The following two rules define how an attribute selection is rewritten.

Rule 8. (Pushing an attribute)
If $cds_{RVM}\ cds, N \rhd\ le : C$

$$Push(S, le.t) \quad \sqsubseteq \ Push(S, le);\ \ Getfd(S, C.t)$$

◇

The expression le above may be arbitrarily nested; it can be simplified by applying this rule again to $Push(S, le)$. The abbreviation $Getfd(S, C.t)$ is described in Definition 8 (see Section 4.2.2).

Rule 9. (Popping to an attribute)
If $cds_{RVM}\ cds, N \rhd\ le : C$

$$Pop(S, (C\ le).t) \quad \sqsubseteq \ Push(S, le);\ Putfd(S, C.t);\ \ Pop(S, le)$$

◇

We use $Push(S, le)$ and $Pop(S, le)$ because le may be arbitrarily nested, and may need further simplification; in Definition 8 we describe $Putfd(S, C.t)$. Object creation is just another instance of assignment.

Rule 10. (Object Creation - Simplification)

$$(x := \textbf{new}\, C) \ = \quad \textbf{var}\ S : Stack;\ \bullet\ S := \textbf{new}\ Stack;\ \ Push(S, \textbf{new}\ C);\ \ Pop(S, x)\ \textbf{end}$$

◇

In this case, we use $Push(S, \textbf{new}\ C)$ to push the object onto the operand stack.

When the guards are simplified (Section 4.3.2), a type test is rewritten to an expression that is assigned to a boolean variable. Therefore, the rule for a type test is just a specific case of the rule for an assignment in which the boolean type of the target of the assignment must be considered.

Rule 11. (Type test - Simplification)

$$(x := (o\ \textbf{is}\, C)) \ = \quad \textbf{var}\ S : Stack;\ \bullet\ S := \textbf{new}\ Stack;\ \ Push(S, o);\ \ Is(S, C);\ Pop_b(S, x)\ \textbf{end}$$

◇

The object o in $Push(S, o)$ may actually be an arbitrarily complex object-valued expression and, therefore, it may need to be submitted to further simplifications. The abbreviation $Is(S, C)$ pops the element at the top of S and tests if it has type C. The result is encapsulated into an instance of $DataBool$ and pushed onto S. This instance of $DataBool$ has one attribute named $Info$, which has a boolean type, and holds the result of this type test. For this reason, we need to use the version of Pop for boolean variables ($Pop_b(S, x)$) to assign the result of the type test to x.

4.3.5. Irrelevant sequences of Push and Pop

In our approach, the code is generated in a bottom up way. When combining the small pieces of code into larger chunks, there is opportunity for some optimisations. In particular, during this phase of compilation, several variable blocks are introduced containing sequences of $Push$ and Pop involving the operand stack S. By expanding the scope of the variable blocks, we can apply laws to eliminate unnecessary assignments to initialise the variables S and V, as well as sequences of $Push$ and Pop. In this section, we explain how this can be achieved.

The lemmas in this section play a role similar to that of the algebraic laws; their proofs can be found in [Dur05]. They express general properties that arise from the combination of some operators, whereas the rules have the special purpose of reducing programs to normal form. The next lemma asserts that an assignment does not affect the stack.

Lemma 1. (Assignment does not affect the resulting stack)

$$(\{S = X\};\ Push(S, e);\ \ Pop(S, x)) = (\{S = X\};\ Push(S, e);\ \ Pop(S, x);\ \ \{S = X\})$$

◇

The initialiser declared in the class $Stack$ (presented in Figure 5) assigns an empty sequence to its attribute just after creation of an object. Therefore, we have the following lemma. It states that after a new object of class $Stack$ is assigned to S, we can always introduce an assumption whose condition is the emptiness of S.

Lemma 2. (Initial empty stack)

$$S := \textbf{new}\ Stack\ =\ (S := \textbf{new}\ Stack;\ \{empty(S)\})$$

where $empty(S) \overset{def}{=} S.s\ =\ \langle\rangle$ $\diamond$

This means that in the variable block introduced by Rule 5 we can introduce the assumption $\{empty(S)\}$ just before the end of its scope. This is useful to eliminate irrelevant assignments whenever we expand the scope of the variable blocks introduced during the compilation process.

The next rule states that $Push(S, o)$ composed in sequence with $Pop(S, o)$ has no effect whatsoever. During compilation, such an irrelevant pair of $Push$ and Pop can be eliminated using this rule.

Rule 12. ($Push;\ Pop$ effect)

$$Push(S,x);\ Pop(S,x) = \textbf{skip}$$

$\diamond$

When we have $Pop(S, x)$ followed by $Push(S, x)$, the effect is that of assigning the top of S to x.

Lemma 3. ($Pop;\ Push$ effect)

$$Pop(S,x);\ Push(S,x) = x := head(S.s)$$

$\diamond$

More optimisations can be used to improve the code, but our focus is on the overall compilation process.

4.4. Control Elimination

In this section, we present the Control Elimination compilation phase; it aims at reducing the nested control structure of the program to a single flat iteration. The outcome of this phase is a program whose control structure is that of our normal form. The next theorem summarises this result; its proof is in [Dur05].

Theorem 4. (Control Elimination)
Consider a program $cds_{RVM}\ cds_{rmc}\ L_s \bullet c_s$ where cds_{rmc} has only declarations of public attributes. Methods declared in L_s have the form

$$\textbf{meth}\ lm \triangleq (\textbf{val}\ S_{in} : Stack;\ \textbf{res}\ S_{out} \bullet \textbf{var}\ w : T \bullet\ p\ \textbf{end})$$

and c_s has the form $\textbf{var}\ v : T\ \bullet\ q\ \textbf{end}$. The commands p and q also satisfy the following conditions: (1) there are no local declarations, (2) all assignments are through the operand stack, (3) all boolean conditions are boolean variables or a negation of a boolean variable, (4) all conditionals are written in an *if-then-else* form, and (5) there are no parametrised commands. Then, there is a command c_{CE} such that

$$cds_{RVM}\ cds_{rmc}\ L_s\ \bullet\ c_s\ \sqsubseteq\ cds_{RVM}\ cds_{rmc}\ \bullet\ c_{CE}$$

The main command c_{CE} has the same control structure as I in the normal form $cds_{RVM} \bullet I$, but c_{CE} still operates on the abstract state space of the source program variables. $\diamond$

To accomplish the goal established by this theorem, we apply the compilation rules presented in the next two sections.

4.4.1. Introducing the Program Counter

Our normal form is equipped with the program counter PC used for scheduling the selection and sequencing of bytecode instructions. More precisely, PC is the pointer which indicates the location of the next bytecode instruction to be executed. In this compilation phase, we introduce the program counter PC.

We represent the RVM instructions as abbreviations of sequences of ROOL commands, as described in Section 3. Moreover, for convenience, we define an abbreviation for a command that is similar to the main command of our interpreter, but declares just one of the RVM components: PC; we call it a PC frame.

Definition 11. (Abbreviation for the PC frame)

$$\begin{array}{lll} PC : [s, GCS, f] & \overset{def}{=} & \textbf{var}\ PC : \textbf{int};\ \bullet \\ & & \quad PC := s; \\ & & \quad \textbf{while}\ PC \geq s \wedge PC < f \rightarrow \textbf{if}\ GCS\ \textbf{fi end}; \\ & & \quad \{PC = f\} \\ & & \textbf{end} \end{array}$$

The following patterns are introduced in this phase. They are similar to the RVM instructions previously defined, but they still operate on the (abstract) state space of the source program.

Definition 12. (Abbreviations for Control Elimination)

$$\begin{array}{lll} nop & \overset{def}{=} & PC := PC + 1 \\ Load_{CE}(e) & \overset{def}{=} & Push(S, e);\ PC := PC + 2 \\ Store_{CE}(x) & \overset{def}{=} & Pop(S, x);\ PC := PC + 2 \\ aconstnull & \overset{def}{=} & Push(S, \textbf{null});\ PC := PC + 1 \\ IsOf_{CE}(C) & \overset{def}{=} & Is(S, C);\ PC := PC + 2 \\ Putfield_{CE}(C.t) & \overset{def}{=} & Putfd(S, C.t);\ PC := PC + 2 \\ Getfield_{CE}(C.t) & \overset{def}{=} & Getfd(S, C.t);\ PC := PC + 2 \end{array}$$

The abbreviation $Load_{CE}(e)$ pushes an expression onto the operand stack. To pop a value and assign it to a variable, we use $Store_{CE}(x)$. We use the abbreviation $aconstnull$ to push the constant **null** onto the operand stack. A type test is performed by $IsOf_{CE}(C)$. Finally, $Putfield_{CE}(C.t)$ and $Getfield_{CE}(C.t)$ are used for attribute selection purposes. Similarly, below we introduce abbreviations for patterns of program that deal with integer and boolean expressions.

Definition 13. (Abbreviations for Control Elimination – integer and boolean values)

$$\begin{array}{lll} iLoad_{CE}(e) & \overset{def}{=} & Push_i(S, e);\ PC := PC + 2 \\ iStore_{CE}(x) & \overset{def}{=} & Pop_i(S, x);\ PC := PC + 2 \\ bLoad_{CE}(e) & \overset{def}{=} & Push_b(S, e);\ PC := PC + 2 \\ bStore_{CE}(x) & \overset{def}{=} & Pop_b(S, x);\ PC := PC + 2 \end{array}$$

The abbreviation $iLoad_{CE}$ pushes an integer value onto the operand stack, whereas $bLoad_{CE}$ pushes a boolean value. Similarly, $iStore_{CE}$ pops an integer and assigns it to a variable; in the same way, $bStore_{CE}$ deals with booleans.

Using the following rules, abbreviations such as $Push(S, x)$ can be reduced to the form given in Definition 11. Here we assume that S and x are different from PC and, for this reason, we omit these side conditions.

The next lemma shows that any command can be written in normal form. This is particularly useful in the reduction of method calls and assignments, as well as in the proof related to recursion elimination.

Lemma 4. (Commands in normal form) If w and PC are not free in p then

$$p \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow p;\ PC := f), f]$$

$\diamond$

The proof of this lemma is similar to that of Lemma 4.1 (Primitive commands) presented in [Sam97]. The following normal form representations (from Rules 13 to 24) are instantiations of the above lemma.

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

Rule 13. (Skip)

$$\textbf{skip} \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow nop), s + 1]$$

$\diamond$

As shown in Definition 12, the only effect of nop is the PC increment.

During the code generation process, **abort** may arise; it is handled by the rule below.

Rule 14. (Abort)

$$\textbf{skip} \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow \textbf{abort}), s]$$

$\diamond$

The reduction of **abort** does not increment the PC. When executing the target code, once PC reaches the address s, the program presents the most undefined behaviour: it may fail to terminate or it may terminate with any result.

The next few rules deal with the elements of abbreviations introduced in the previous phases. The first such rule presented below considers the pattern $Load_{CE}(x)$ to load a variable.

Rule 15. (Push variable)

$$Push(S,x) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow Load_{CE}(x)), s+2]$$

$\diamond$

As previously defined, $Load_{CE}(x)$ is an abbreviation of a sequence of commands that push the value of x onto the operand stack S, and increments the PC by two. Now we show the reduction of a pattern that stores an integer variable.

Rule 16. (Pop variable)

$$Pop(S,x) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow Store_{CE}(x)), s+2]$$

$\diamond$

Like the $Load$ instruction, the $Store$ instruction also requires two bytecodes. It pops a value from S and stores it in x.

The abbreviation $Load_{CE}(\textbf{new}\ C)$ pushes a fresh object onto the operand stack. Its reduction is handled by the Rule 17 below. As an example, we present its proof of soundness.

Rule 17. (Object Creation)

$$Push(S, \textbf{new}\ C) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow Load_{CE}(\textbf{new}\ C)), s+2]$$

Proof

```
$PC : [s, (PC = s \rightarrow Load_{CE}(\textbf{new}\ C)), s+2]$
$=$   var $PC$ : int •          [Definitions 11 (Abbr. for the $PC$ frame) and 12 (Abbr. for Control Elimination)]
        $PC := s$;
        while $PC \geq s \wedge PC < s+2 \rightarrow$
          if $PC = s \rightarrow S.s := \langle \textbf{new}\ C \rangle \frown S.s;\ PC := PC+2$ fi
        end; $\{PC = s+2\}$
      end

$=$   var $PC$ : int •          [Laws 22 (; := void assertion) and 16 (while unfold)]
        $PC := s;\ [PC = s]$;
        if $PC = s \rightarrow S.s := \langle \textbf{new}\ C \rangle \frown S.s;\ PC := PC+2$ fi
        while $PC \geq s \wedge PC < s+2 \rightarrow$
          if $PC = s \rightarrow S.s := \langle \textbf{new}\ C \rangle \frown S.s;\ PC := PC+2$ fi
        end; $\{PC = s+2\}$;
      end

$\sqsupseteq$   var $PC$ : int •          [Laws 18 (if selection) and 22 (; := void assertion)]
        $PC := s;\ [PC = s]$;
        $S.s := \langle \textbf{new}\ C \rangle \frown S.s;\ PC := PC+2;\ [PC = s+2]$;
        while $PC \geq s \wedge PC < s+2 \rightarrow$
          if $PC = s \rightarrow S.s := \langle \textbf{new}\ C \rangle \frown S.s;\ PC := PC+2$ fi
        end; $\{PC = s+2\}$
      end
```

$$= \quad \begin{array}{l}\textbf{var}\ PC : \textbf{int} \bullet \\ \quad PC := s;\ [PC = s]; \\ \quad S.s := \langle \textbf{new}\ C\rangle ^\frown S.s;\ PC := PC + 2; \\ \quad [PC = s+2];\ \{PC = s+2\} \\ \textbf{end}\end{array} \qquad \text{[Law 17 (\textbf{while} elimination)]}$$

$$= \quad \begin{array}{l}\textbf{var}\ PC : \textbf{int} \bullet \\ \quad PC := s;\ [PC = s]; \\ \quad S.s := \langle \textbf{new}\ C\rangle ^\frown S.s;\ PC := PC + 2 \\ \textbf{end}\end{array} \qquad \text{[Law 13 (; } [b];\ \{b\} \text{ simulation)]}$$

$$= \quad \begin{array}{l}\textbf{var}\ PC : \textbf{int} \bullet \\ \quad PC := s;\ [PC = s]; \\ \textbf{end}; \\ S.s := \langle \textbf{new}\ C\rangle ^\frown S.s\end{array} \qquad \text{[Laws 14 (\textbf{var}- := final value) and 15 (\textbf{var}-; left dist)]}$$

$$= \quad \begin{array}{l}\textbf{var}\ PC : \textbf{int} \bullet \textbf{end}; \\ S.s := \langle \textbf{new}\ C\rangle ^\frown S.s\end{array} \qquad \text{[Laws 22 (; := void assertion) and 14 (\textbf{var}- := final value)]}$$

$$\sqsupseteq S.s := \langle \textbf{new}\ C\rangle ^\frown S.s \qquad \text{[Law 11 (\textbf{var} elim)]}$$

$$= Push(S, \textbf{new}\ C) \qquad \text{[Definition 8 (Abbreviations — operand stack)]}$$

◇

For a type test, we have the following rule.

Rule 18. (Type Test)

$$Is(S,C) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow IsOf_{CE}(C)), s+2]$$

◇

The next rule deals with the sequence of commands that sets a new value to an attribute of an object.

Rule 19. (Set an attribute)

$$Putfd(S, C.t) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow Putfield_{CE}(C.t)), s+2]$$

◇

The following rule addresses the sequence of commands $Getfield_{CE}(C.t)$ that pops an object from the operand stack, gets the value of its attribute t, and pushes it onto the operand stack.

Rule 20. (Set an attribute)

$$Getfd(S, C.t) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow Getfield_{CE}(C.t)), s+2]$$

◇

To reduce the call associated with a binary or unary operator, we use the following rules.

Rule 21. (Binary Operator)

$$Bop(S) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow bop), s+1]$$

◇

The bop (and uop) introduced in this phase is exactly the same given in the RVM instruction set (see Definition 3).

Rule 22. (Unary Operator)

$$Uop(S) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow uop), s+1]$$

◇

The reduction of a method call follows from Lemma 4. The rule is very simple: it preserves the call by rewriting it into normal form. The actual translation into bytecode instructions is discussed in Section 4.4.2.

Rule 23. (Method Call)

$$le.lm(x,x) \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow \; le.lm(x,x); \;\; PC := PC + 2), s + 2]$$

$\diamond$

Similarly, the rule to reduce assignments is as follows.

Rule 24. (Assignment)

$$x := \; e \quad \sqsubseteq \quad PC : [s, (PC = s \rightarrow \; x := \; e; \;\; PC := PC + 1), s + 1]$$

$\diamond$

The reduction of a sequential composition requires that both arguments are in normal form and the final address (j) of the guarded command set (GCS_1) of the left argument is the initial address of the set (GCS_2) of the right argument.

Rule 25. (Sequential composition)

$$PC : [s, GCS_1, j]; \; PC : [j, GCS_2, f] \quad \sqsubseteq \quad PC : [s, (GCS_1 \, [] \, GCS_2), f]$$

$\diamond$

The resulting normal form has the initial position of the left argument and the final position of the right argument.

The following rule shows how a conditional can be transformed to the reduced form when its branches have already been reduced, which can be assumed by structural induction.

Rule 26. (Conditional)
If the address intervals $[s_1, f_0]$ and $[s_2, f]$ are disjoint, then

$$\left(\begin{array}{l} \textbf{if}\,(x) \;\rightarrow PC : [s_1, GCS_1, f_0] \\ \;[]\, \neg(x) \rightarrow \;\; PC : [s_2, GCS_2, f] \\ \textbf{fi} \end{array} \right) \quad \sqsubseteq \quad PC : [s, R, f]$$

$$\text{where } R \; = \left(\begin{array}{l} \quad PC \; = \; s \;\; \rightarrow (\textbf{if}\,(x) \;\rightarrow\; PC := s_1 \,[]\, \neg(x) \;\rightarrow\; PC := s_2 \;\textbf{fi}) \\ \quad [] \; GCS_1 \\ \quad [] \; (PC \; = \; f_0) \;\; \rightarrow (PC := f) \\ \quad [] \; GCS_2 \end{array} \right)$$

$\diamond$

The first guarded command to be executed in the reduced command determines which of the original initial instructions should be activated. To reduce an iteration, we assume that its body is already reduced.

Rule 27. (Iteration)

$$\begin{array}{l} \textbf{while}\; x \;\; \rightarrow \\ \qquad PC : [(s+1), GCS, f_0] \qquad \sqsubseteq \quad PC : [s, R, (f_0 + 1)] \\ \textbf{end} \end{array}$$

$$\text{where } R \; = \left(\begin{array}{l} \quad PC \; = \; s \;\; \rightarrow (\textbf{if}\; x \;\rightarrow\;\; PC := PC + 1 \; [] \; \neg(x) \;\rightarrow\; PC := f_0 + 1 \;\textbf{fi}) \\ \quad [] \; GCS \\ \quad [] \; PC \; = \; f_0 \;\; \rightarrow (PC := s) \end{array} \right)$$

$\diamond$

In the loop, s and $f_0 + 1$ are the start and finish addresses for the while compiled code. In R the first command to be executed is that with guard $(PC = s)$. It tests if the condition of the iteration holds; in this case, the next instruction to be executed is the first one in the body of the iteration, and so the PC is incremented by 1. If, on the other hand, the condition does not hold, the PC is set so that we jump to the end of the loop.

The exhaustive application of the rules in this section yields a program of the form below.

var $F, S : Stack; \;\; v : T \bullet PC : [s_c, GCS_c, f_c]$ **end**

Applying Law 10 (**var** association), we can join the variable blocks obtaining a program that is similar to the normal

```
$Cds_{RVM}$, $cds$,
class $L$
    {meth $lm_i \triangleq$ (val $S_{in} : Stack$  res $S_{out} : Stack$ $\bullet$
        rec $X \bullet w_m : [s_m, GCS_m, f_m]$ end)}$^*$
end
    $\bullet$ $w_c : [s_c, GCS_c, f_c]$ end
```

Fig. 21. Intermediate program obtained before the elimination of L

form command, but still operates on the abstract data space. This program is described by Definition 14 below.

Definition 14. (Abbreviation for the flat iteration)

```
$w : [s, GCS, f]$  $\overset{def}{=}$  var $PC$ : int;  $F, S : Stack$;  $v : T$ $\bullet$
                      $F :=$ new $Stack$;  $S :=$ new$Stack$;  $PC := s$;
                      while $PC \geq s \wedge PC < f \rightarrow$
                          if $GCS$ fi
                      end;
                      $\{PC = f\}$
                  end
```

where w represents a list of variables comprising the RVM components PC, S, F, and the list of local variables v declared in the main command or in a method body. ◇

We observe that F and S represent the frame stack and the operand stack. The operand stack S is introduced by the rules of redirection of method calls, and assignment simplifications, whereas the frame stack F is introduced by the rules that deal with the elimination of recursive commands and parametrised recursive calls.

Eventually, after applying the above rules, a command of the form $w_m : [s_m, GCS_m, f_m]$ becomes the body of each method m declared in L, and a command $w_c : [s_c, GCS_c, f_c]$ becomes the main command. Figure 21 presents the form of the resulting program at this stage. There might exist several method declarations in L, each one operating on an assumed disjoint interval established by the pair of values (s_m, f_m). The main command also operates on an interval, defined by the pair (s_c, f_c), also assumed to be disjoint from the other intervals.

In order to reduce the program depicted in Figure 21 to normal form, it is necessary to eliminate the class L. This is the objective of the next step. The major obstacle resides in the calls to methods in L that may exist in GCS_m and in GCS_c. Recursive parametrised calls have eliminated as explained in Section 4.3.1. Nevertheless, recursive methods might exist, implemented using the recursive command **rec** $X \bullet p$ **end**. To eliminate the class L, we handle recursion and method calls separately, so that the overall task becomes modular.

4.4.2. Nested Normal Form

To eliminate calls one approach would be to replace them with the corresponding method body. The use of the copy rule to eliminate methods, however, is not acceptable because it may substantially increase the size of the target code. The method bodies and the main command have been compiled into separate segments of code. The replication of code is avoided by keeping just one copy of each segment, in such a way that control passes back and forth between these segments, simulating the same behaviour provided by the copy rule. We omit the details of this step here because it is similar to the imperative case [Sam97]; further details can also be found in [Dur05].

In Figure 22, we present the general control structure of our program example after the Control Elimination phase. The main command and each method body is placed in a code segment. Each segment is formed of a sequence of guarded commands whose guards indicate its location in the bytecode stream. For our example, the segment associated to $lgetLength$ (see Figure 20), for instance, is presented in Figure 23.

In practice, a compiler implementation based on, for instance, a term rewriting system, would need to ensure that the addresses of the target code are disjoint and contiguous. This can be achieved by annotating the source program with natural numbers representing the addresses of memory instructions. These addresses determine, for example, the values of k_1, k_2 abd k_3 in Figure 22. More details can be found in [Sam97].

```
$cds_{RVM}$ $cds$ $\bullet$   var $PC$ : int;  $S, F : Stack, w : T$   $\bullet$
      $S :=$ new $Stack$;  $F :=$ new $STack$;
      $PC := s$;
      while $PC \geq s \wedge PC < f \rightarrow$
        if $PC = s \rightarrow PC := k_3$
          [] $PC = k_1 \rightarrow$   $laddStep$
          ⋮
          [] $PC = k_2 \rightarrow$   $lgetlength$
          ⋮
          [] $PC = k_3 \rightarrow$   $Main\ Command$
        fi
      end
    end
```

Fig. 22. General structure of our example after the Control Elimination phase

```
⋮
[] $PC = k_2 + 40$   $\rightarrow Push_i(F, aux)$
[] $PC = k_2 + 42$   $\rightarrow Push_i(F, x)$
[] $PC = k_2 + 44$   $\rightarrow Push_i(F, x_2)$
[] $PC = k_2 + 46$   $\rightarrow Push_i(F, x_1)$
[] $PC = k_2 + 48$   $\rightarrow Push(F, o_4)$
[] $PC = k_2 + 50$   $\rightarrow invoke(k_2)$
[] $PC = k_2 + 52$   $\rightarrow Pop(F, o_4)$
[] $PC = k_2 + 54$   $\rightarrow Pop_i(F, x_1)$
[] $PC = k_2 + 56$   $\rightarrow Pop_i(F, x_2)$
[] $PC = k_2 + 58$   $\rightarrow Pop_i(F, x)$
[] $PC = k_2 + 60$   $\rightarrow Pop_i(F, aux)$
[] $PC = k_2 + 62$   $\rightarrow Putfield_{CE}(Path.previous)$
[] $PC = k_2 + 64$   $\rightarrow Store_{CE}(o_4)$
[] $PC = k_2 + 66$   $\rightarrow iStore_{CE}(aux)$
[] $PC = k_2 + 68$   $\rightarrow Load_{CE}(o_4)$
[] $PC = k_2 + 70$   $\rightarrow Getfield_{CE}(Path.previous)$
[] $PC = k_2 + 70$   $\rightarrow iStore_{CE}(x)$
[] $PC = k_2 + 74$   $\rightarrow iLoad_{CE}(x)$
[] $PC = k_2 + 76$   $\rightarrow iLoad_{CE}(x)$
[] $PC = k_2 + 78$   $\rightarrow add$
[] $PC = k_2 + 79$   $\rightarrow iStore_{CE}(x)$
[] $PC = k_2 + 81$   $\rightarrow goto(k_2 + 87)$
[] $PC = k_2 + 83$   $\rightarrow Load_{CE}(o_4)$
[] $PC = k_2 + 85$   $\rightarrow Getfield_{CE}(Step.len)$
[] $PC = k_2 + 87$   $\rightarrow iLoad_{CE}(x)$
[] $PC = k_2 + 89$   $\rightarrow Load_{CE}(o_4)$
[] $PC = k_2 + 91$   $\rightarrow Pop(F, PC)$ %{Return}
```

Fig. 23. Sequence of guarded commands corresponding to a fragment of $lgetLength$ after the Control Elimination phase

4.5. Data Refinement

The Data Refinement phase replaces the (abstract) data space of the source program with the concrete state of the target machine. This means that all references to variables, attributes, and classes declared in the source program are replaced with references to the components the data space of the target machine model.

The following theorem, proved in [Dur05], summarises the outcome of this phase of compilation.

Theorem 5. (Data Refinement)
Consider a program $cds_{RVM}\ cds_{rmc}\ L \bullet c_{CE}$, where cds contains only public attributes, and c_{CE} has the form of Definition 14. Then there is a Ψ and an I such that

$$\overset{\wedge}{\Psi}_w\ (cds_{RVM}\ cds_{rmc}\ L \bullet c_{CE}) \quad \sqsubseteq \quad cds_{RVM} \bullet I$$

and I is in normal form. $\diamond$

After the Data Refinement phase is performed, there are no references to classes declared in the source program. The replacement of the abstract space by the concrete space involves references to specific variables that are part of the data structure of the target machine. A mapping $\overset{\wedge}{\Psi}_w$ is required to link the data model of the RVM with the object model of the source program. To do so, symbol tables are necessary to relate the concrete and the abstract data spaces.

To carry out the change of data representation, we use the distributivity properties of $\overset{\wedge}{\Psi}$ presented later in this section. It is a polymorphic simulation function (built from the symbol tables) that applies to programs and commands, and distributes over the commands in the main command. The function $\overset{\wedge}{\Psi}$ *reclassifies* [DDDCG02, Ser99] the objects stored in the variables referenced by the main command. Reclassification enables an object to have its class membership changed, preserving its identity, by creating a concrete representation of its attribute values.

In Figure 24 we present the concrete representation of an object in a variable x. When the change of data representation is performed, the object in x is reclassified and stored in the memory storage M, in the location indicated by Ψ_x. M is a sequence of $Data$ objects; as explained in Section 3, the class $Data$ has two subclasses: $PriData$ and $ObjData$. Objects are stored in $ObjData$ instances, whereas primitive values are instances of $PriData$. The class $PriData$ has several subclasses, one for each primitive type. In this example, *DataInt* is the subclass of $PriData$ whose instances encapsulate the values of the integer attributes of x.

In the concrete space, the source class declarations *cds* do not exist, so a concrete instance must have an extra attribute to keep the type information of the object it represents in the abstract space. In the class $ObjData$, the attribute *cl* codifies the type of the abstract instance. The attribute *att* is a sequence of $Data$, where each element is the concrete representation of an attribute of the class type indicated by cl. We use Φ as the symbol table mapping each attribute of a class in the abstract space to an address in the sequence of attributes (att) in the concrete space. The term $\Phi_{C.x}$ gives the location in the sequence att in which the value of the abstract non-primitive attribute x is stored. So $M[\psi_o].att[\Phi_{C.i}]$ is the concrete storage of the attribute i of an object of class C, stored in the variable o.

Differently from [DDDCG02] and [Ser99], where an object may have many types according to trees of class hierarchies, the object reclassification that we present has the following characteristics: all objects in the concrete space are instances of the class $Data$ or of its subclasses, and abstract instances do not coexist with concrete instances, except during the moment in which data refinement is performed.

To allow objects to change their class membership when the data refinement is performed, two methods ($encode$ and $decode$) are introduced in each class declared in the source program. An object with $encode$ and $decode$ methods knows how to copy itself from the abstract state to the concrete state and vice-versa, maintaining the corresponding representation of the original attributes and initialising the extra attributes. These methods are introduced and eliminated in this phase; we introduce them merely to simplify the description of our Data Refinement phase.

To illustrate how these methods are structured, we consider the class declarations in Figure 25. Each class has its specific $encode$ method to build an abstract instance from a concrete instance. Every $encode$ method has an object of class $ObjData$ as a value parameter D. Based on its att and cl attributes, the $encode$ method retrieves the values of each attribute of an object in the abstract space of the source program.

In Figure 25 we give as examples the methods $encode$ and $decode$ for two classes A and B. Class B declares two attributes, whereas class A declares just one. The attribute z has a class type, whereas the attributes l and k have integer types. The methods $encode$ and $decode$ are introduced, in the beginning of this phase, using Law 8 (method elimination), from right to left, so that we introduce, rather than eliminate these methods.

Since some attributes can be inherited from a superclass, if the immediate superclass is not **object**, the $encode$ method begins with a call (**super**$.encode(D)$) to invoke the $encode$ method declared in its immediate superclass. In

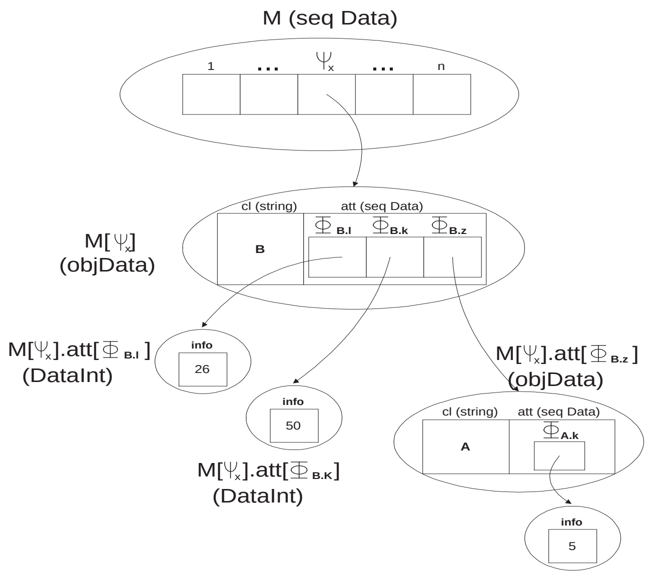

Fig. 24. An object in the concrete space

```
class $A$ extends object
   pub $k$ : int
   meth $encode \triangleq$ (val $D : objData$ • self.$k := (DataInt\ D.att[\Phi_{A.k}]).info$)
   meth $decode \triangleq$ (res $D : objData$ • $D :=$ new $objData$; $D.cl := A'$;
                           $D.att[\Phi_{A.k}] :=$ (new $DataInt$; $info :$ self.$k$))
end
class $B$ extends $A$
   pub $z : A$
   pub $l$ : int
   meth $encode \triangleq$ (val $D : objData$ •
         super.$encode(D)$;
         if $D.att[\Phi_{B.z}] =$ null → self.$z :=$ null
         [] ¬($D.att[\Phi_{B.z}] =$ null) →
              if $(D.att[\Phi_{B.z}].cl = 'A')$ → self.$z :=$ new $A$ [] $(D.att[\Phi_{B.z}].cl = 'B')$ → self.$z :=$ new $B$ fi;
              self.$z.encode(D.att[\Phi_{B.z}])$
         fi;
         self.$l := (DataInt\ D.att[\Phi_{B.l}]).info$)
   meth $decode \triangleq$ (res $D : objData$ •
         $D :=$ new $objData$; $D.cl := 'B'$;
         if self.$z =$ null → $D.att[\Phi_{B.z}] :=$ null [] ¬(self.$z =$ null) → self.$z.decode(D.att[\Phi_{B.z}])$ fi;
         $D.att[\Phi_{B.l}] :=$ (new $DataInt$; $info :$ self.$l$);
         var $D_s : objData$ • super.$decode(D_s)$; $D.att := D.att \frown D_s.att$ end)
end
```

Fig. 25. An example of *encode* and *decode* methods

the example of Figure 25, the value of the attribute k declared in the class A is retrieved by calling the $encode$ method declared in class A, using **super**. The term $D.att[\Phi_{B.l}]$ holds the value of the attribute l, declared in class B. In the case of this primitive attribute, its value is encapsulated in an instance of $DataInt$.

Since z is a non-primitive attribute of type A, there are two possible (run time) types for z: A and B. Furthermore, we distinguish the case when it is **null** or not. If $D.att[\Phi_{B.z}]$ is **null**, then, the attribute z is **null**; otherwise, testing the attribute cl, it is possible do initialise z with the proper type, A or B. Once $D.att[\Phi_{B.z}]$ is different from **null**, the call **self**.$z.encode(D.att[\Phi_{B.z}])$ initialises the attributes of the new object assigned to the attribute z.

Similarly, the $decode$ method creates the corresponding concrete instance of an object. The $decode$ methods described in Figure 25 have a result parameter D, which is an instance of $ObjData$. The attribute cl is assigned according to the type of the target object of the $decode$ call. The value of each non-primitive abstract attribute is encapsulated in the corresponding concrete instance, and is stored in a specific position in the sequence $D.att$. For example, the attribute l is encapsulated in an instance of $DataInt$, and stored in $D.att[\Phi_{B.l}]$. For the attribute z, which has a class type, it is necessary to check its value; if z is **null**, then **null** is assigned to $D.att[\Phi_{C.z}]$, otherwise, the $decode$ method associated with the type of this attribute is invoked to obtain the value of the concrete instance. The attributes inherited from a superclass are copied to the concrete instance by calling the $decode$ method declared in the immediate superclass. When this call completes, the resulting sequence ($D_s.att$) is concatenated with $D.att$.

The next lemma establishes the outcome of the sequential composition of calls to $encode$ and $decode$ methods. It is used in the proofs of the rules applied in this phase; recall that M represents the memory storage of our virtual machine, whereas Ψ_x is the location where the concrete instance of x is stored in M.

Lemma 5. (Encode; Decode)

$$\begin{array}{l} (1)\ o.encode(M[\Psi_o]);\ o.decode(M[\Psi_o]) \ = \ o.encode(M[\Psi_o]) \\ (2)\ o.decode(M[\Psi_o]);\ o.encode(M[\Psi_o]) \ = \ o.decode(M[\Psi_o]) \end{array}$$

provided o is not **null**. ◇

On the left-hand side of (1), an abstract instance for o is constructed, through $encode$, in the concrete space $M[\Psi_o]$. Then $decode$ is used to recover the concrete state from the abstract space. Clearly, this results in $M[\Psi_o]$ itself and, therefore, the $decode$ has no effect. The intuition for (2) is similar.

Since we have explained the $encode$ and $decode$ methods, now we can discuss the $encoding$ and $decoding$ blocks. The $encoding$ block introduces the abstract state and ends the scope of the concrete state, while the $decoding$ block introduces the concrete state and ends the scope of the abstract state. The approach is based on that in [Bac80, BvW90].

In our approach to Data Refinement, the commands **dvar** and **dend** play an essential role in defining the $encoding$ and $decoding$ blocks. For conciseness, consider an abstract space with only two variables: o and i. The variable o has a class type C, whereas i has the integer primitive type. From that we define the following encoding block.

Definition 15. (Encoding block $\overset{\wedge}{\Psi}_{o,i}$)

```
[$M[\Psi_o].cl = \Phi_C$]
dvar $o : C;\ i :$ int;
    if $M[\Psi_o] = $ null $\rightarrow\ o :=$ null $[] \neg(M[\Psi_o] = $ null$) \ \rightarrow o :=$ new $C;\ o.encode(M[\Psi_o])$ fi;
    $i := (DataInt\ M[\Psi_i]).Info$;
dend $M$
```

◇

It maps the concrete space to the abstract space. By calling the corresponding $encode$ method of each non-primitive source variable, and encapsulating the primitive source variables according to their types, the data state of the target program stored in the memory storage M is copied to the data state of the source program. Once the abstract state is initialised, the concrete state is no longer needed and therefore its scope is terminated. The assertion $[M[\Psi_o].cl = \Phi_C]$ avoids the need for conditionals to test the type of the variable o.

The $decoding$ block below retrieves the concrete space of the target machine model. Once it is properly initialised, the abstract data space of the source program is terminated.

Definition 16. (Decoding Block $\overset{\wedge}{\Psi}^{-1}_{o,i}$)

```
**dvar** $M : seq\ Data$;
    **if** $o = \mathbf{null} \ \rightarrow\ M[\Psi_o] := \mathbf{null}\ [] \neg(o = \mathbf{null})\ \rightarrow\ o.decode(M[\Psi_o])$ **fi**;
    $M[\Psi_i] := (\mathbf{new}\ DataInt;\ Info : i)$;
**dend** $o, i$
```

◇

The *encoding* block retrieves the abstract space from the concrete state, assigning to each source variable the value stored in the corresponding location in the memory storage M. In case of a class type variable, the corresponding *encode* method is called, according to its type, to retrieve its attributes. On the other hand, the *decoding* block does exactly the opposite, obtaining the concrete representation of each source variable. Similarly, to convert a class type variable, the corresponding *decode* method is called to build its concrete instance.

There is an asymmetry between the definitions of the *encode* and *decode* methods; *encode* deals only with the attributes, whereas *decode* also deals with the object creation. This is a consequence of the way we have modelled these methods, associating them with the class declared in the source program.

The next lemma formalises the relationship between $\overset{\wedge}{\Psi}$ and $\overset{\wedge}{\Psi}^{-1}$; they form a simulation. We take the list of variables of the source program to be w. The proof is in [Dur05].

Lemma 6. (($\overset{\wedge}{\Psi}$, $\overset{\wedge}{\Psi}^{-1}$) simulation)

$$\overset{\wedge}{\Psi}_w;\ \overset{\wedge}{\Psi}^{-1}_w \ \sqsubseteq\ \mathbf{skip}\ \sqsubseteq\ \overset{\wedge}{\Psi}^{-1}_w;\ \overset{\wedge}{\Psi}_w$$

◇

We use the first component of a simulation as a function.

Definition 17. (Simulation Function)
Let $(\overset{\wedge}{\Psi}_w, \overset{\wedge}{\Psi}^{-1}_w)$ be a simulation. We use $\overset{\wedge}{\Psi}$ itself as a function defined by

$$\overset{\wedge}{\Psi}_w\ (c)\ = \overset{\wedge}{\Psi}_w;\ \ c;\ \overset{\wedge}{\Psi}^{-1}_w$$

We use the overloaded version of $\overset{\wedge}{\Psi}_{i,o}$ to deal with expressions. When applied to an expression, $\overset{\wedge}{\Psi}_{i,o}$ replaces free occurrences of non-primitive variables o in e with the corresponding concrete $M[\Psi_o]$, whereas the primitive variable i is replaced with $M[\Psi_i].Info$, and the object identified by o with $M[\Psi_o]$.

Definition 18. (Simulation as substitution)

$$\overset{\wedge}{\Psi}_{i,o}\ (e)\ =\ e[M[\Psi_i].Info/i]\,[M[\Psi_o]/o]$$

◇

The function $\overset{\wedge}{\Psi}_w$ does not affect the classes and components of our interpreter nor commands that have no references to variables or classes affected by $\overset{\wedge}{\Psi}_w$. Similarly, any command that has no reference to any class or variable declared in the source program is not affected. The following rule addresses the distribution of $\overset{\wedge}{\Psi}$ over an assignment.

Rule 28. (Assignment- $\overset{\wedge}{\Psi}_{x,w}$ - Data Refinement)

$$\overset{\wedge}{\Psi}_{x,w}\ (x := e)\ \ \sqsubseteq\ \ M[\Psi_x] := \overset{\wedge}{\Psi}\ (e)$$

◇

Above x is replaced with the corresponding machine location $M[\Psi_x]$. When applied to constructors that deal with control, like conditional, $\overset{\wedge}{\Psi}$ distributes over their components. The simple rules are omitted. The next rules deal with abbreviations introduced in the previous compilation phase; the following rule gives the effect of $\overset{\wedge}{\Psi}$ over $Load_{CE}(x)$.

Rule 29. (Load variable - Data Refinement)

$$\overset{\wedge}{\Psi}_w\ (Load_{CE}(x)) \quad \sqsubseteq \quad Load(\Psi_x)$$

$\diamond$

While the $Load_{CE}(x)$ pushes an abstract instance onto S, $Load(\Psi_x)$ pushes the concrete counterpart. A similar rule holds for $Store_{CE}(x)$. When $\overset{\wedge}{\Psi}$ is applied to a constant a, it is placed in the entry of the constant pool Φ_a.

Rule 30. (Load Constant - Data Refinement)

$$\overset{\wedge}{\Psi}_w\ (Load_{CE}(a)) \quad \sqsubseteq \quad Ldc(\Phi_a)$$

$\diamond$

After the simplification phase, each object creation is refined to an abbreviation like $Load(\textbf{new}\ C)$. It has as parameter an expression that retains a reference to C, a class declared in the source program. In the control elimination phase $Load(\textbf{new}\ C)$ is transformed to $Load_{CE}(\textbf{new}\ C)$. The function $\overset{\wedge}{\Psi}$ eliminates this reference, replacing $Load_{CE}(\textbf{new}\ C)$ with the abbreviation $new(\Phi_C)$ whose parameter is an index to an entry in the constant pool *CP*, where the concrete instance of a fresh object of type C is stored.

Rule 31. (Object Creation - Data Refinement)

$$\overset{\wedge}{\Psi}_w\ (Load_{CE}(\textbf{new}\ C)) \quad \sqsubseteq \quad new(\Phi_C)$$

$\diamond$

The instruction $Instanceof(\Phi_C)$ pops an object from the operand stack. If this object is an instance of C or of a subclass of C, the value *true* is pushed onto the stack; otherwise, the value *false* is pushed onto the stack. If the object at the top of the stack is **null**, the result is always *false*. This instruction relies on the context of classes declared in the source program. In the following we present its definition as a sequence of commands.

```
var $D : objData;\ r : boolean\ \bullet$
    $D := head(S.s);\ S.s := Tail(S.s);\ Cls.Instanceof(D, \Phi_C, r);$
    $S.s := \langle(\textbf{new}\ DataBool;\ Info : r)\rangle \frown S.s$
end
```

where the method *Instanceof* traverses the class hierarchy to establish whether the object at the top of the operand stack belongs to a class type C or to any of its subclasses. The term Φ_C indicates the object (*ClsData*) that represents C in *Cls*. The method *Instanceof* is defined in [Dur05]. The next rule applies the function $\overset{\wedge}{\Psi}$ to eliminate the reference to a class C present in a type test, and originally declared in *cds*.

Rule 32. (Instanceof - Data Refinement)

$$\overset{\wedge}{\Psi}_w\ (IsOf(C)) \quad \sqsubseteq \quad Instanceof(\Phi_C)$$

$\diamond$

The following rule deals with getting a value of an object attribute.

Rule 33. (Getfield - Data Refinement)

$$\overset{\wedge}{\Psi}_w\ (Getfield(C.t)) \quad \sqsubseteq \quad Getfield(\Phi_{C.t})$$

$\diamond$

The index $\Phi_{C.t}$ indicates the location of the attribute t in a sequence of attributes in a concrete instance of an object whose static type is C in the abstract space. This concrete instance is popped from the operand stack; $Getfield(\Phi_{C.t})$ gets the value corresponding to t and pushes it onto the top of the operand stack S. Similarly, the abbreviation

$$
\begin{array}{ll}
\vdots & \\
[]\ PC = k_2 + 40 & \rightarrow save(\Psi_{aux}) \\
[]\ PC = k_2 + 42 & \rightarrow save(\Psi_{x}) \\
[]\ PC = k_2 + 44 & \rightarrow save(\Psi_{x_2}) \\
[]\ PC = k_2 + 46 & \rightarrow save(\Psi_{x_1}) \\
[]\ PC = k_2 + 48 & \rightarrow save(\Psi_{o4}) \\
[]\ PC = k_2 + 50 & \rightarrow invoke(k_2) \\
[]\ PC = k_2 + 52 & \rightarrow restore(\Psi_{o_4}) \\
[]\ PC = k_2 + 54 & \rightarrow restore(\Psi_{x_1}) \\
[]\ PC = k_2 + 56 & \rightarrow restore(\Psi_{x_2}) \\
[]\ PC = k_2 + 58 & \rightarrow restore(\Psi_{x}) \\
[]\ PC = k_2 + 60 & \rightarrow restore(\Psi_{aux}) \\
[]PC = k_2 + 62 & \rightarrow putfield(\Phi_{Path.previous}) \\
[]\ PC = k_2 + 64 & \rightarrow store(\Psi_{o4}) \\
[]\ PC = k_2 + 66 & \rightarrow store(\Psi_{aux}) \\
[]\ PC = k_2 + 68 & \rightarrow load(\Psi_{o4}) \\
[]\ PC = k_2 + 70 & \rightarrow getfield(\Phi_{Path.previous}) \\
[]\ PC = k_2 + 70 & \rightarrow store(\Psi_{x}) \\
[]\ PC = k_2 + 74 & \rightarrow load(\Psi_{x}) \\
[]\ PC = k_2 + 76 & \rightarrow load(Psi_{x}) \\
[]\ PC = k_2 + 78 & \rightarrow add \\
[]\ PC = k_2 + 79 & \rightarrow store(\Psi_{x}) \\
[]\ PC = k_2 + 81 & \rightarrow goto(88) \\
[]\ PC = k_2 + 83 & \rightarrow load(\Psi_{o4}) \\
[]\ PC = k_2 + 85 & \rightarrow getfield(\Phi_{Step.len}) \\
[]\ PC = k_2 + 87 & \rightarrow load(\Psi_{x}) \\
[]\ PC = k_2 + 89 & \rightarrow load(\Psi_{o4}) \\
[]\ PC = k_2 + 91 & \rightarrow return
\end{array}
$$

Fig. 26. Sequence of instructions corresponding to a fragment of $lgetLength$ after the Data Refinement phase

$Putfield(\Phi_{C.t})$ sets the value of the attribute identified by t in the object stored at the top of the operand stack.

Rule 34. (Putfield - Data Refinement)

$$\overset{\wedge}{\Psi}_w\ (Putfield(C.t)) \quad \sqsubseteq \quad Putfield(\Phi_{C.t})$$

$\diamond$

As illustration, Figure 26 presents the resulting segment of guarded commands that represents the sequence of bytecode instructions of the method $lgetLength$. The resulting program has the same structure of that presented in Figure 22. The classes and variables declared in the source program are eliminated. The program operates exclusively on the concrete space. From this program we can capture the sequence of generated instructions for the target machine (RVM).

4.6. The overall compilation process

The overall compilation process asserts that, for any program $cds \bullet c$ we can transform $\overset{\wedge}{\Psi}\ (cds \bullet c)$ into a normalised program $cds_{RVM} \bullet I$. This is captured by a theorem that links the intermediate steps and establishes the correctness of the entire compilation process, as presented below.

Theorem 6. (Compilation Process) For every executable program $cds \bullet c$ composed by class declarations and a main command c, there are symbol tables Ψ, which maps each variable of $cds \bullet c$ to the address of the memory storage M allocated to hold its value, and Φ, mapping each constant, attribute, and class name to objects in the virtual machine such that $\overset{\wedge}{\Psi}\ (cds \bullet c) \quad \sqsubseteq \quad cds_{RVM} \bullet I$ where, as already mentioned, cds_{RVM} is a set of class declarations that describes the RVM components, and I is an abbreviation representing a main command in normal form; it is expressed in terms of a set of guarded commands GCS of the form $(PC = k) \rightarrow p$, where PC is the program counter, k is a natural number between s and f, representing the address of an instruction in the interval of the intended start address (s) and the finish address (f) of the code to be executed, and p is a sequence of ROOL instructions that simulate the effect of an RVM bytecode instruction. $\diamond$

From Theorem 1 (Class Pre-compilation), we can transform $cds \bullet c$, eliminating method redefinitions, introducing casts and changing attribute visibility to public. With these transformations, a program $cds_{pc} \bullet c_{pc}$ is obtained.

Theorem 2 (Redirection of method calls) establishes that $cds_{pc} \bullet c_{pc}$ can be refined to $cds_{RVM}\ cds_{rmc}\ L \bullet c_{rmc}$. At this point, the class L together with cds_{RVM} are introduced, all method calls are redirected to corresponding ones in L and, therefore, all methods declared in cds_{pc} become useless and are eliminated.

After that, we can refer to Theorem 3 (Simplification of Expressions), which guarantees that an intermediate program $cds_{RVM}\ cds_{rmc}\ L_s \bullet c_s$ can be obtained from the program $cds_{RVM}\ cds_{rmc}\ L \bullet c_{rmc}$. In the code of the methods in L_s and c_s each assignment operates through the operand stack, each boolean expression is a variable or a negation of a variable, there are no parametrised commands, and every conditional is written in an *if-then-else* form.

At this point, based on Theorem 4 (Control elimination), we can transform the nested control structure of the program $cds_{RVM}\ cds_{rmc}\ L_s \bullet c_s$ into a single flat iteration, obtaining a program $cds_{RVM}\ cds_{rmc} \bullet c_{CE}$, which has the same structure of the normal form, but still operates on the abstract data space.

Finally, based on Theorem 5 (Data Refinement), using the distributive properties of the simulation function $\overset{\wedge}{\Psi}$ the necessary changes of data representation are performed. Therefore, cds_{rmc} becomes useless and is eliminated. The outcome program operates exclusively over the concrete space. The final program $cds_{RVM} \bullet I$ is in normal form.

Proof of Theorem 6

$$
\begin{array}{ll}
\overset{\wedge}{\Psi}(cds \bullet c) & \\
\sqsubseteq \overset{\wedge}{\Psi}(cds_{pc} \bullet c_{pc}) & \text{[Theorem 1 (Class Pre-compilation)]} \\
\sqsubseteq \overset{\wedge}{\Psi}(cds_{RVM}\ cds_{rmc}\ L \bullet c_{rmc}) & \text{[Theorem 2 (Redirection of method calls)]} \\
\sqsubseteq \overset{\wedge}{\Psi}(cds_{RVM}\ cds_{rmc}\ L_s \bullet c_s) & \text{[Theorem 3 (Simplification of expressions)]} \\
\sqsubseteq \overset{\wedge}{\Psi}(cds_{RVM}\ cds_{rmc} \bullet c_{CE}) & \text{[Theorem 4 (Control Elimination)]} \\
\sqsubseteq\ cds_{RVM} \bullet I & \text{[Theorem 5 (Data Refinement)]}
\end{array}
$$

$\diamond$

This completely algebraic formalisation of the compilation process for an object-oriented language is the main result of this work. All definitions and proofs can be found in [Dur05].

5. Conclusions

This work is a contribution to the field of compiler design and correctness, extending the approach in [Sam97] from the purely imperative to the object-oriented paradigm. Here (and in [DCS02, DCS03, Dur05]) we presented a set of compilation rules; they are justified in terms of basic laws of an object-oriented language, and laws of data refinement. These laws are expressed in a language called ROOL, which is similar to a subset of sequential Java [CN00]. It includes recursive classes, inheritance, access control, dynamic binding, type tests and casts, recursion, assignment, and many other imperative features. Also, it includes specification constructs, such as the specification statements of Morgan's refinement calculus [Mor94]. Both the source and the target (a simplified model of the JVM, called here RVM) languages adopted here are far more complex than that described in [Sam97].

There are many formalisations of Java and the JVM. In [BS98], using Abstract State Machines (ASM), a compilation scheme of Java programs to JVM code is presented as a case study on mechanical verification of compiler correctness proofs. A large sequential sublanguage of Java is described in [NO98]. Its abstract syntax, type system, well-formedness conditions, and an operational semantics are formalised as ASMs. Based on this formalisation, type soundness is proved. Using the theorem prover Isabelle/HOL all definitions and proofs are checked.

In [SSB01], a structured sequence of mathematical models for the static and dynamic behaviour of Java is defined. The models clarify and explore the official description [GJSB00, LY97]. Java and the JVM are decomposed into language layers and security modules, dividing the verification tasks into a series of tractable subtasks. This work does not leave out any relevant language feature. This decomposition is made in such a way that, in the resulting sequence of machines, each ASM is an extension of its predecessor. They are used to establish when Java programs can be proved type safe, and successfully verified, and correctly executed when compiled to JVM code.

To address the issue of scaling of the verifications, an ASM-based approach is presented in [BB08]; it shows how to combine feature-based modular design and proof techniques in order to handle software product lines. Based on the structured way ASMs are used in [SSB01], the components, and the accompanying statements and proofs of the theorems verifying the desired compilation correctness, are developed incrementally.

In [KN06] a Java-like sequential programming language with formal semantics is introduced. This provides a unified model of the source language, the virtual machine, and the compiler. It is a standard operational semantics, and all correctness proofs for program compilation are machine-checked in Isabelle/HOL.

Another example based on verification comes from the functional language community [Tia06]. In this work, the correctness of a CPS compiler is formalised using higher-order-abstract syntax. In order to guarantee that the source program and the CPS-transformed program have the same observable behavior, the CPS translation together with its correctness proof is implemented and mechanically verified in a logical framework. To prove correctness, it is shown the correspondence between the high-level language and the low-level abstract machine.

The certifying compiler back-end presented in [BG08a, BG08b] provides the translation of an intermediate language into MIPS [PH05] code. In this compiler, the code generation phase is extended by a certificate generator producing Coq correctness proofs for each compiler run. These are proved correct in Coq yielding the guarantee that the compiler has worked correctly. As a certifying compiler, these proofs do not make a statement about the compiler implementation correctness, but only about the relation of two programs. Coq is also used in the development of CompCert [Ler09], a compiler for a subset of C comparable to those currently used by the safety-critical embedded-systems industry. The target language is that of a processor favoured in the avionics industry. The approach is based on a series of intermediate languages with a formal operational semantics. Each phase of compilation is proved to preserve the semantics of the program using Coq; the compiler itself is written in the functional language of Coq.

Closely related to the algebraic style adopted here is the work reported in [LF02], which presents a formal compilation model for real-time programs. As we do here, compilation is characterised as a sequence of refinement transformations. The source language for this compilation formalism consists of a small, imperative programming language with special real-time statements, whereas the target language is an intermediate representation language, more abstract than the machine code. The refinement calculus adopted in this work is a hybrid of the standard refinement calculus and the real-time calculus described by Hayes and Utting [Hay98, HU98].

Wildman [Wil02] presented a case study on compilation verification for imperative programs to assembly code. A compilation strategy is presented as a data refinement. The compiled code is achieved by calculation.

Another work focusing on using refinement as a compilation tool for imperative languages is presented in [Wat03]. In this work program refinement is extended to the assembler level; compilation is regarded as a phase in the entire refinement process, from specification to machine code. As the source language, a version of Dijkstra's Guarded Command Language is used. The target language is an assembly language based on a fragment of Microsoft's *Common Interface Language* (CIL), which is part of the .NET framework [LT01, Pla01].

A remarkable example that handles parallelism, communication and external choice is in [He93]. The elimination of these complex structures is carried out at the source level rather than via the direct generation of a normal form. Once the program obtained by eliminating concurrency is entirely described in terms of the source language, a low-level implementation is generated by reduction to a normal form. This work complements ours since it addresses concurrency, but not object-orientation. In the same direction, a more recent work [PW07a, PW07b, PW08] focuses on the algebraic compilation of Handel-C [Bow98], a hardware language that includes concurrency and time operators.

We are not aware of any other work, based on pure algebraic calculation, which handles compilation of object-oriented languages.

Our work can be extended in several directions. We are investigating the mechanisation of the compilation strategy. As our rules have the form of rewrite rules, we can use a term rewrite system both to mechanise their proofs and to carry out compilation automatically. In this way, a prototype compiler is a by-product of its proof of correctness.

We deal with dynamic binding in a purely algebraic way to support a compilation strategy based on the systematic application of laws. In practice, however, dynamic binding is not implemented using dynamic tests as we do; instead, each object refers to a table of the associated method implementations. To deal with this compilation technique, we need a framework that can handle method implementations (code) as part of data structures (objects). This is certainly both an interesting and a challenging topic to explore in an algebraic setting.

In terms of code optimisation, we provide only the rules related to the elimination of irrelevant sequences of push and pop. Therefore, there is further work to be done, especially when dealing with the creation of temporary variables for the elimination of nested expressions and the management of memory during method invocation. Another topic to investigate is the replacement of sequences of bytecode for efficiency.

A further topic of investigation resides in the extension of our source language with more complex structures. Considering pointers, for example, does not affect the compilation rules themselves, since we already handle memory addresses. On the other hand, this does impact the proof of soundness of the compilation rules for commands (particularly, assignment statements) and the rules that justify the data refinement phase.

We deal with sharing separately from inheritance, dynamic binding, and the other object-oriented constructs. In [HCW08] we address pointers using the relational models of Hoare and He's unifying theories of programming [HJ98], which can also cater for concurrency. Based on the laws that we use here, we also present in [SCS06] a copy semantics for object orientation. We are now working on the integration of these theories, to provide a basis for justifying object-oriented programming laws in a reference semantics. Our long term goal is to provide a compilation strategy that covers a comprehensive modern concurrent object-oriented language.

Acknowledgements The work of Ana Cavalcanti is partially funded by the Royal Society of London and QinetiQ, and that of Augusto Sampaio by the Brazilian Research Agency CNPq (grant 309048/2006-0).

References

[ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[Bac80] R. Back. *Correct Preserving Program Refinements: Proof Theory and Application*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[BB08] E. Börger and D. S. Batory. Coupling design and verification in software product lines. In *FoIKS*, pages 1–4, 2008.

[BG08a] J. O. Blech and B. Grégoire. Certifying code generation with Coq. In *Workshop Compiler Optimization meets Compiler Verification (COCV 2008), ENTCS*. Elsevier, 2008.

[BG08b] J. O. Blech and B. Grégoire. Certifying code generation with Coq: A tool description. In *Workshop Compiler Optimization meets Compiler Verification (COCV 2008), ENTCS*. Elsevier, 2008.

[Bow98] M. Bowen. *Handel-C Language Reference Manual, 2.1*. Embedded Solutions Limited, 1998.

[BS98] E. Börger and W. Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 17–35, London, UK, 1998. Springer-Verlag.

[BSC03] P. Borba, A. Sampaio, and M. Cornélio. A Refinement Algebra for Object-Oriented Programming. In *ECOOP 2003: European Conference on Object-oriented Programming 2003*, volume 2743, pages 457–482. LNCS, Springer-Verlag, 2003.

[BSCC04] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1-3):53–100, 2004.

[BvW90] R. Back and J. von Wright. Refinement calculus, part I: sequential nondeterministic programs. In *REX workshop: Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 42–66. Springer-Verlag, 1990.

[BvW98] R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[CCS02] M. Cornélio, A. Cavalcanti, and A. Sampaio. Refactoring by Transformation. In *REFINE 2002 Workshop*, volume 70, pages 641–660. Electronic Notes in Theoretical Computer Science, Springer-Verlag, 2002.

[CN00] 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.

[Cor04] M. Cornélio. *Applying Object-Oriented Refactoring and Patterns as Formal Refinements*. PhD thesis, Universidade Federal de Pernambuco, 2004.

[DCS02] A. Duran, A. Cavalcanti, and A. Sampaio. Refinement Algebra for Formal Bytecode Generation. In *ICFEM 2002 - 4th International Conference on Formal Engineering Methods*, volume 2495, pages 347–358, Shanghai, China, October 2002. LNCS, Springer-Verlag.

[DCS03] A. Duran, A. Cavalcanti, and A. Sampaio. A Strategy for Compiling Classes, Inheritance, and Dynamic Binding. In *FME 2003 - International Symposium of Formal Methods Europe*, volume 2805, pages 301–320, Pisa, Italy, September 2003. LNCS, Springer-Verlag.

[DDDCG02] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle II. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Engewood Cliffs, 1976.

[Dur05] A. Duran. *An Algebraic Approach to the Design of Compilers for Object-Oriented Languages*. PhD thesis, Universidade Federal de Pernambuco, 2005. Available at http://www.les.ufba.br/duranthesis/.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[Hay98] I. J. Hayes. Separating timing and calculation in real-time refinement. In J. Gundy, M. Schwenke, and T.Vickers, editors, *Pacific'98: International Refinement Workshop and Formal Methods*, pages 1–16. Discrete Mathematics and Theoretical Computer Science, Springer-Verlag, 1998.

[HCW08] W. Harwood, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Theory of Pointers for the UTP. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 141 – 155. Springer-Verlag, 2008.

[He93] J. He. Hybrid parallel programming and implementation of synchronised communication. In *MFCS '93: Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, volume 711, pages 537–546, London, UK, 1993. LNCS, Springer-Verlag.

[HHS93] C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HU98] I. J. Hayes and M. Utting. Deadlines are termination. In D. Gries and W.-P de Roever, editors, *PROCOMET'98: IFIP – International Conference on Programming Concepts and Methods*, pages 186–204. Chapman and Hall, 1998.

[KN06] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006. http://doi.acm.org/10.1145/1146809.1146811.

[Ler09] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107 – 115, 2009.

[LF02] K. Lermer and C. Fidge. A formal model of real-time program compilation. *Theoretical Computer Science*, 282(1):151–190, 2002.

[LT01] H. Lam and T. Thai. *.NET Framework Essentials*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[MO97] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*. Springer-Verlag, 1997.

[Mor94] C. Morgan. *Programming from Specifications*. Prentice-Hall, Inc., second edition, 1994.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Symposium on Applied Mathematics*, volume 19, pages 33–41. American Mathematical Society, 1967.

[NO98] T. Nipkow and D. Oheimb. Java-light is type-safe — Definitely. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–170, New York, NY, USA, 1998. ACM Press.

[Opd92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[PH05] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[Pla01] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2001.

[Pol81] W. Polak. *Compiler Specification and Verification*. Springer-Verlag, 1981.

[PW07a] J. I. Perna and J. Woodcock. A denotational semantics for Handel-C hardware compilation. In *ICFEM*, pages 266–285, 2007.

[PW07b] J. I. Perna and J. Woodcock. Proving wire-wise correctness for Handel-C compilation in HOL. Technical Report YCS-2008-429, Computer Science Department, The University of York, December 2007.

[PW08] J. I. Perna and J. Woodcock. Wire-Wise Correctness for Handel-C Synthesis in HOL. In Gordon J. Pace and Satnam Singh, editors, *Seventh International Workshop on Designing Correct Circuits (DCC)*, pages 86–100, March 2008.

[Sam97] A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.

[SCS06] T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 18 – 37. Springer-Verlag, 2006.

[Ser99] M. Serrano. Wide Classes. *Lecture Notes in Computer Science*, 1999.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.

[SSL08] L. Silva, A. Sampaio, and Z. Liu. Laws of Object-Orientation with Reference Semantics. *Software Engineering and Formal Methods, International Conference on*, 0:217–226, 2008.

[Tia06] Y. H. Tian. Mechanically verifying correctness of CPS compilation. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 41–51, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[TWW81] 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.

[Wat03] G. Watson. Compilation by refinement for a practical assembly language. In *Formal Methods and Software Engineering. ICFEM 2003: 5^{th} International Conference on Formal Engineering Methods*, volume 2885, pages 286–305, Singapore, November 2003. LNCS, Springer-Verlag.

[Wil02] L. Wildman. A formal basis for a program compilation proof tool. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, volume 2391, pages 491–510, London, UK, 2002. LNCS, Springer-Verlag.

A. Laws of ROOL

Here, we present the algebraic laws referenced in the description of our compilation process. The next law establishes that a method declaration and its redefinition can be merged into a single declaration in the superclass. The appropriate behaviour is determined by type tests in the resulting method.

Law 1. (move redefined method to superclass)

```
class $B$ extends $A$
    $ads$
    meth $m \triangleq (pds \bullet b)$
    $ops$
end
class $C$ extends $B$
    $ads'$
    meth $m \triangleq (pds \bullet b')$
    $ops'$
end
```

$=_{cds,c}$

```
class $B$ extends $A$
    $ads$
    meth $m \triangleq (pds \bullet$
        if $\neg$(self is $C) \rightarrow b$
            [] self is $C \rightarrow b'$
        fi)
    $ops$
end
class $C$ extends $B$
    $ads'$
    $ops'$
end
```

provided

($\leftrightarrow$) (1) **super** and private attributes do not appear in b'; (2) **super**.m does not appear in ops';
($\rightarrow$) b' does not contain uncast occurrences of **self**;
($\leftarrow$) m is not declared in ops'. ◇

We use Law 1 to handle dynamic binding as a separate issue. In doing so, we surprisingly end up needing simple laws like copy rule to characterise method call elimination. Since the order in which the methods of a class are declared is irrelevant, a law like this allows us to transform any method in a class. Another notation is introduced. Hereafter, we write $cds, N \rhd e : C$ to indicate that in the class N declared in cds, the expression e has static type C.

Law 2. (method call elimination)
Consider that the following class declarations

```
class $C$ extends $D$
    $ads$
    meth $m \triangleq pc$
    $ops$
end
```

is included in cds, and $cds, A \rhd le : C$. Then
$cds\, A \rhd le.m(e) = \{le \neq \textbf{null} \wedge le \neq \textbf{error}\};\ pc[le/\textbf{self}]$

provided

($\leftrightarrow$) (1) m is not redefined in cds and pc does not contain references to **super**; (2) all attributes which appear in the body pc of m are public ◇

The notation $pc[le/\textbf{self}]$ represents the parametrised command pc where every occurrence of **self** is replaced with le. We need an assumption on the right hand side of this law because in case le is **null** or **error**, a method call $le.m(e)$ aborts. If the condition in the assumption holds it behaves like **skip**, otherwise as **abort**.

The following law allows us to introduce or eliminate a class declaration.

Law 3. (class elimination)

$cds\, cd_1 \bullet c = cds \bullet c$

provided

($\rightarrow$) The class declared in cd_1 is not referred to in cds or c;
($\leftarrow$) (1) The name of the class declared in cd_1 is distinct from those of all classes declared in cds; (2) the superclass appearing in cd_1 is either **object** or declared on cds; (3) and the attribute and methods names declared by cd_1 are not declared by its superclasses in cds, except in the case of method redefinitions. ◇

The order in which classes are declared does not matter, so we can use this law to introduce or remove a declaration in any position of the program. The next law addresses the change of visibility of an attribute. From left to right, it states that a protected attribute can be made public, from right to left, it establishes that a public can be made protected, provided it is only directly used in the class in which it is declared and its subclasses.

Law 4. (change visibility: from protected to public)

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \text{extends}\ D\\ \quad\ \textbf{prot}\ a : T;\ ads\\ \quad\ ops\\ \textbf{end}\end{array}} =_{cds,c} \boxed{\begin{array}{l}\textbf{class}\ C\ \text{extends}\ D\\ \quad\ \textbf{pub}\ a : T;\ ads\\ \quad\ ops\\ \textbf{end}\end{array}}$$

provided

$(\leftarrow)$ $B.a$, for any $B \leq C$, appears only in ops and in the subclasses of C in cds. ◇

We write **prot** $a : T;\ ads$ to denote the attribute declaration **prot** $a : T$ and all the declarations in ads, whereas ops stands for the declarations of methods and constructors. The notation $B.a$ refers to uses of the name a via expressions whose static type is exactly B, as opposed to any of its subclasses. For example, if we write that $B.a$ does not appear in ops, we mean that ops does not contain any expression such as $e.a$, for any e of type B, strictly. Since the order of the declarations in a class is irrelevant, a law like this allows us to change any declaration in a class.

Law 5. (change visibility: from private to public)

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \text{extends}\ D\\ \quad\ \textbf{pri}\ a : T;\ ads\\ \quad\ ops\\ \textbf{end}\end{array}} =_{cds,c} \boxed{\begin{array}{l}\textbf{class}\ C\ \text{extends}\ D\\ \quad\ \textbf{pub}\ a : T;\ ads\\ \quad\ ops\\ \textbf{end}\end{array}}$$

provided

$(\leftarrow)$ $B.a$, for any $B \leq C$, does not appear in cds, c. ◇

When applied from left to right, the above law makes a private attribute public. The proviso is necessary when applying this law in the other direction to make a private attribute public. In this case, the attribute cannot be used anywhere outside the class where it is declared. A property of declarations is that the order in which the attributes are declared is not relevant. We use a sequence, rather than, for instance, a mapping because the program terms obey the syntax.

The following law establishes that we can introduce or remove a trivial method redefinition, which consists only of a call to the method in the superclass of the same name. We observe again that there is no overloading in ROOL.

Law 6. (introduce method redefinition)

$$\boxed{\begin{array}{l}\textbf{class}\ B\ \text{extends}\ A\\ \quad ads\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} pc\\ \quad ops\\ \textbf{end}\\ \textbf{class}\ C\ \text{extends}\ B\\ \quad ads'\\ \quad ops'\\ \textbf{end}\end{array}} = \boxed{\begin{array}{l}\textbf{class}\ B\ \text{extends}\ A\\ \quad ads\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} pc\\ \quad ops\\ \textbf{end}\\ \textbf{class}\ C\ \text{extends}\ B\\ \quad ads'\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} \textbf{super}.m\\ \quad ops'\\ \textbf{end}\end{array}}$$

provided

$(\rightarrow)$ m is not declared in ops'. ◇

Actually, in ROOL a method declaration is an explicit parametrised command, such that pc has the form $(pds \bullet c)$. For simplicity we adopt the notation **meth** $\mathrel{\widehat{=}}$ **super**$.m$ as an abbreviation for **meth** $\mathrel{\widehat{=}}$ $(pds \bullet \textbf{super}.m(\alpha pds))$, where αpds represents the list of parameter names declared in pds. This law is independent of its context of class declarations and main command. So, it is an equality on class declarations.

The following law allows us to move up in the hierarchy a method declaration that is not a redefinition. Our language supports method redefinition but not overloading. Hence, it is not possible to have different methods in the same class, or in a class and a subclass, with the same name, but different parameters. This law also indicates that we can move a method down, as long as this method is used as if it were defined in the subclass.

Law 7. (move original method to superclass)

$$\begin{array}{|l|}\hline \textbf{class}\ B\ \text{extends}\ A \\ \quad ads \\ \quad ops \\ \textbf{end} \\ \textbf{class}\ C\ \text{extends}\ B \\ \quad ads' \\ \quad \textbf{meth}\ m \mathrel{\hat{=}} pc \\ \quad ops' \\ \textbf{end} \\ \hline\end{array} =_{cds,c} \begin{array}{|l|}\hline \textbf{class}\ B\ \text{extends}\ A \\ \quad ads \\ \quad \textbf{meth}\ m \mathrel{\hat{=}} pc \\ \quad ops \\ \textbf{end} \\ \textbf{class}\ C\ \text{extends}\ B \\ \quad ads' \\ \quad ops' \\ \textbf{end} \\ \hline\end{array}$$

provided

($\leftrightarrow$) (1) **super** and private attributes of the class where m is declared do not appear in pc; (2) m is not declared in any superclass of B in cds;

($\rightarrow$) (1) m is not declared in ops, and can only be declared in a class D, for any $D \leq B$ and $D \nleq C$, if it has the same parameters as pc; (2) pc does not contain uncast occurrences of **self** nor expressions in the form $((C)\textbf{self}).a$ for any private attribute a in ads'.

($\leftarrow$) (1) m is not declared in ops'; (2) $D.m$, for any $D \leq B$, does not appear in cds, c, ops, or ops'. ◇

The first two provisos are necessary to preserve semantics, whereas the others guarantee that we relate syntactically valid programs. The second proviso, associated to the application of the law in both directions, precludes superclasses of B from defining m. This is an important restriction because, when moving m, we could affect the semantics of calls such as $b.m(e)$, for a b storing an object of B. There are two provisos associated to applications of the Law from left to right. The first one avoids the occurrence of method overloading, which is not supported by ROOL. The second one guarantees that we can only move a original method to superclass if it does not refer to elements of the class where it is declared. Finally, the proviso associated to applications of the law from right to left prevent us from moving down in the class hierarchy a method declaration that is redefined in the subclass.; moreover, it precludes that method calls become invalid when moving a method declaration down in the class hierarchy.

To introduce or eliminate a method in a class, we use the next law. It states that a method that is not called can be eliminated. Conversely, a new method can always be introduced in a class.

Law 8. (method elimination)

$$\begin{array}{|l|}\hline \textbf{class}\ C\ \text{extends}\ D \\ \qquad ads \\ \qquad \textbf{meth}\ m \mathrel{\hat{=}} pds \bullet b \\ \qquad ops \\ \textbf{end} \\ \hline\end{array} =_{cds,c} \begin{array}{|l|}\hline \textbf{class}\ C\ \text{extends}\ D \\ \qquad ads \\ \qquad ops \\ \textbf{end} \\ \hline\end{array}$$

provided

($\rightarrow$) $B.m$ does not appear in cds, c nor in ops, for any B such that $B \leq C$;

($\leftarrow$) m is not declared in ops nor in any superclass or subclass of C in cds. ◇

The first proviso precludes the elimination of a method that is in use; if a method call $C.m$ to m appears in ops and cds, the method m cannot be eliminated. Conversely, if m is a method name that is not declared in a superclass or subclass of C in cds, the second proviso allows us to introduce m in C.

The following law formalises the fact that any expression can be cast with its declared type.

Law 9. (cast introduction in expressions)

If $cds, A \rhd\ e : C$, then $cds, A \rhd\ e = (C)e$

◇

This also holds for left-expressions, but, like Java, we do not allow casts in targets of assignments and result parameters.

It does not matter whether variables are declared in one list or singly.

Law 10. (**var** association)
If x and y have no variables in common, then

$$\textbf{var}\ x : T_x \bullet (\textbf{var}\ y : T_y \bullet c\ \textbf{end})\ \textbf{end} \ = \ \textbf{var}\ x, y : T_x, T_y \bullet c\ \textbf{end}$$

◇

If a declared variable is never used, it can be eliminated.

Law 11. (**var** elim)
If x is not free in c, then

$$\textbf{var}\ x : T \bullet\ c\ \textbf{end}\ =\ c$$

◇

If the right argument of a sequential composition declares the variable x the scope can be extended to the left component, provided that it does not interfere with the other variables with the same name.

Law 12. (**var**-; right dist)
If x is not free in c_1, then

$$c_1;\ \textbf{var}\ x : T \bullet\ c_2\ \textbf{end}\ =\ \textbf{var}\ x : T \bullet\ c_1;\ c_2\ \textbf{end}$$

◇

The next law establishes that the pair ($[b]$; $\{b\}$) is a simulation. Simulations are useful for calculations. $[b]$ is called the strongest inverse of $\{b\}$ whereas $\{b\}$ is the weakest inverse. Whenever a program followed by its inverse appears as a subterm of a program being transformed, it is possible to replace them with **skip**.

Law 13. (; $[b]$; $\{b\}$ simulation)

$$(\{b\};\ [b])\ =\ \{b\}\ \sqsubseteq\ \textbf{skip}\ \sqsubseteq\ [b]\ =\ [b];\ \{b\}$$

◇

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

Law 14. (**var**- := final value)

$$\textbf{var}\ x : T \bullet\ c;\ x := e\ \textbf{end}\ \sqsubseteq\ \textbf{var}\ x : T \bullet\ c\ \textbf{end}$$

◇

It is possible to increase the scope of a variable without effect, provided there is no capture of free variables.

Law 15. (**var**-; left dist)
If x is not free in c_2, then

$$\textbf{var}\ x : T \bullet\ c_1\ \textbf{end};\ c_2\ =\ \textbf{var}\ x : T \bullet\ c_1;\ c_2\ \textbf{end}$$

◇

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

Law 16. (**while** unfold)
If $(b_1 \Rightarrow b_2)$, then

$$[b_1];\ \textbf{while}\ b_2 \bullet c\ \textbf{end}\ =\ [b_1];\ c;\ \textbf{while}\ b_2 \bullet c\ \textbf{end}$$

◇

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

Law 17. (**while** elimination)
If $(b_1\ \Rightarrow\ \neg b_2)$, then

$$[b_1];\ \textbf{while}\ b_2 \bullet c\ \textbf{end}\ =\ [b_1]$$

◇

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

Law 18. (**if** selection)
If j ranges over $1..n$ then

$$[b_j];\ \textbf{if}\ []_{\langle 1\leq\ i\ \leq\ n\rangle}\ b_i\ \rightarrow\ c_i\,\textbf{fi}\ \sqsubseteq\ [b_j];\ c_j$$

$\diamond$

The following allows the transformation of a conditional with at least one guarded command to an if-then-else form.

Law 19. (**if** - Conditional in the if-then-else style)
If $k \leq n$, then

$$\begin{array}{lll}\textbf{if}\ []_{\langle k\leq\ i\ \leq\ n\rangle}\ b_i\ \rightarrow\ p_i\ \textbf{fi} & = & \textbf{if}\ \ b_k\ \rightarrow\ p_k \\ & & []\ \neg b_k\ \rightarrow\ \textbf{if}\,[]_{\langle k+1\leq\ i\ \leq\ n\rangle}\ b_i\ \rightarrow\ p_i\ \textbf{fi} \\ & & \textbf{fi}\end{array}$$

$\diamond$

A conditional without guarded commands corresponds to an infinite loop: a concrete implementation of **abort**. The following law states that a conditional without guarded commands is equivalent to an infinite loop. This syntactic transformation allows us to manage this particular case of conditional using recursion compilation rules.

Law 20. (**if** - Conditional without guarded commands)
If $k > n$, then

$$\textbf{if}\ []_{\langle k\leq\ i\ \leq\ n\rangle}\ b_i\ \rightarrow\ p_i\ \textbf{fi}\ \ =\ \ \textbf{rec}\ X \bullet X\ \textbf{end}$$

$\diamond$

To follow an assignment by an assumption whose condition reflects the assignment itself has no effect.

Law 21. (; := void assumption)
If x is not free in e, then

$$x := e;\ \{x = e\}\ =\ x := e$$

$\diamond$

A similar law is valid for assertions.

Law 22. (; := void assertion)
If x is not free in e, then

$$x := e;\ [x = e]\ =\ x := e$$

$\diamond$

If the value of a variable is known, we can replace the occurrences of this variable in an expression with that value.

Law 23. (; := substitution)

$$(x = e) \rightarrow (y := f)\ \ =\ \ (x = e) \rightarrow (y := f[e/x])$$

$\diamond$

A parametrised command can be eliminated according to the parameter transmission mechanism adopted. The next two laws deal with value (**val**) and result (**res**) parameters separately.

Law 24. (Pcom elimination-val)
If the variables of l do not occur free in c, x and vl, then

$$(\textbf{val}\ vl :\ T\ \bullet\ c)(x)\ =\ \textbf{var}\ l : T \bullet\ l := x;\ \ c[l/vl]$$

◇

Law 25. (Pcom elimination-res)
If the variables of l do not occur free in c, x and vl, then

$$(\textbf{res}\ vl :\ T\ \bullet\ c)(x)\ =\ \textbf{var}\ l : T \bullet\ c[l/vl];\ \ x := l;\ \ \textbf{end}$$

◇

The validity of the two transformations above follows direct from the definition of parametrised command. Laws for mixed parameter lists can be easily obtained as a generalisation of these laws, and have been omitted for simplicity.