Universidade Federal de Pernambuco
Centro de Informática
Pós Graduação em Ciência da Computação

Tese de Doutorado

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

Adolfo Almeida Duran

Esta tese foi apresentada à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Thesis presented to the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science.

Orientador(Supervisor):

Prof. Dr. Augusto César Alves Sampaio

Co-orientadora (Co-supervisor):

Profa. Dra. Ana Lúcia Caneca Cavalcanti

Duran, Adolfo Almeida

An algebraic approach to the design of compilers
for object-oriented languages / Adolfo Almeida Duran.

– Recife : O Autor, 2005.

xiii, 194 p. : il., tab., fig.

Tese (doutorado) – Universidade Federal de
Pernambuco. Cln. Ciência da Computação, 2005.

Inclui bibliografia e anexo.

1. Ciência da computação – Linguagem de
programação. 2. Linguagem orientada a objetos –
Compiladores corretos – Cálculo de refinamento. 3.
Compilação – Abordagem algébrica – Linguagem
ROOL. I. Título.

004.4'422

005.453

CDU (2.ed.)

CDD (22.ed.)

UFPE

BC2005-224

Resumo

Neste trabalho discutimos o projeto de compiladores corretos por construção para linguagens orientadas a objeto. Um compilador correto é aquele que garante que a semântica é preservada quando o programa fonte é traduzido para a linguagem destino.

O projeto de compiladores corretos para linguagens imperativas se encontra bem fundamentado; atualmente, o maior desafio é o desenvolvimento de uma abordagem para lidar com características de orientação a objetos. Nesta tese, descrevemos uma abordagem algébrica para construção de compiladores corretos para uma linguagem orientada a objetos chamada ROOL (acrônimo para Refinement Object-oriented Language), que é similar a Java e C++. Esta linguagem inclui classes, herança, ligação dinâmica, recursão, *cast* e teste de tipos, e visibilidade baseada em classes.

Na nossa abordagem, lidamos com o problema de corretude do compilador transformando a tarefa de compilação em uma tarefa de refinamento de programa. O processo de compilação passa ser identificado como sendo a redução de um programa fonte, escrito em um subconjunto executável da linguagem, para uma forma normal. A forma normal é gerada por uma série de transformações que preservam a corretude, e são provadas corretas a partir das leis básicas da linguagem; portanto o processo é correto por construção.

A maior vantagem da nossa abordagem reside na caracterização do processo de compilação dentro de um sistema uniforme onde as comparações e traduções entre semânticas são evitadas. A redução à forma normal é formalizada como uma álgebra onde a noção central é a de refinamento de programas. Portanto, o produto da compilação é um programa na própria linguagem fonte. Nossa forma normal é um programa na forma de um interpretador, escrito na mesma linguagem fonte, emulando o comportamento da máquina destino. A partir desse interpretador é que capturamos a sequência das instruções geradas. Definimos a Máquina Virtual de ROOL (*RVM*) como sendo nossa máquina destino; ela é baseada na Máquina Virtual de Java (*JVM*).

Tal uniformidade implica que todo o cálculo necessário para assegurar a corretude do processo de compilação é realizado em um único sistema de uma linguagem orientada a objetos cuja semântica é dada por leis algébricas. Nenhuma teoria relativa à linguagem fonte ou destino é desenvolvida ou usada no processo.

O processo de compilação é justificado por teoremas de redução à forma normal. Existem cinco fases: pre-compilação de classes, redirecionamento de chamada de métodos, simplificação, eliminação de controle e refinamento de dados. Para cada fase, um teorema assegura o resultado esperado. O teorema principal conecta os passos intermediários e estabelece o resultado para todo o processo. Uma vez que os teoremas de redução para cada fase são provados corretos a partir das leis básicas de ROOL, eles corroboram para a corretude de todo o processo.

Abstract

In this work we discuss the design of compilers, correct by construction, for object-oriented languages. By correct compilers we mean the translation of source programs into a target language, with a guarantee that the semantics is preserved.

The design of correct compilers for procedural languages is already well understood; our main challenge now is the development of an approach to deal with object-oriented features. Here, we describe an algebraic approach to construct a provably correct compiler for an object-oriented language called ROOL (for Refinement Object-oriented Language), which is similar to Java and C++. This language includes classes, inheritance, dynamic binding, recursion, type casts and test, and class-based visibility.

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, written in an executable subset of the language, to a normal form. The normal form is generated by a series of correctness-preserving transformations which are proved correct from the basic laws of the language; therefore it is built correct by construction.

The main advantage of our approach resides on the characterisation of the compilation process within a uniform framework where the comparisons and translations between semantics are avoided. The normal form reduction is formalised as an algebra where the central notion is refinement of programs. Therefore, the product of the compilation is a program in the source language itself. Our normal form is an interpreter-like program written in the same language, which emulates the behavior of the target machine. From this interpreter we capture the sequence of generated instructions. We define a ROOL Virtual Machine (RVM) as our target; it is based on the Java Virtual Machine (JVM).

Such uniformity implies that all the calculation necessary to assert the correctness of the compilation process is carried out within a single framework of an object-oriented language whose semantics is given by algebraic laws. No additional mathematical theory of the source or the target language is develop or used in the process.

The compilation process is justified by reduction theorems. There are five phases: Class Pre-compilation, Redirection of Method Calls, Simplification of Expressions and Some Source Transformations, Control Elimination, and Data Refinement. For each phase, a theorem asserts the expected outcome, and a main theorem links the intermediate steps and establishes the outcome for the entire process. Since the reduction theorem for each phase are proved correct from the basic laws of ROOL, they corroborate the correctness of the whole process.

To
Gabriel & Victor

Acknowledgments

The work described in this thesis has taken several years; therefore there are many people to be acknowledged.

I owe a major debt of gratitude to my supervisor, Augusto Sampaio, and my co-supervisor, Ana Cavalcanti, for their expert advice and guidance during the course of this thesis. They have been a major source of inspiration and encouragement, and I am very grateful to them.

I must thank my examiners, André Santos, Arnaldo Moura, Paulo Borba, Hermann Haeusler and Hermano Perrelli, for making a number of suggestions as how the presentation of this thesis could be improved.

Thanks to Jim Woodcock for his encouragement and for giving many suggestions which helped to fix some compilation rules in the earlier stages of my research. I also thank him for presenting my paper at ICFEM'2002.

I have benefited from the friendship of many people at Cin/UFPE; my sincere thanks go to Alexandre Mota, Márcio Cornélio, Nelson Rosa and Rita Suzana who shared the difficulties of being a PhD student. I apologise to all those I underdeliberately forgot to mention here.

As part of my PhD program, I spent one year in the UK. Special thanks must go to my colleagues at Kent University, Leonardo, Marcel, Xinbei and Rodolfo for making our room a pleasant place to work. For their friendship and support in difficult moments, I must thank Lauana, Viviane and Renato.

I thank my mother, my brothers and sisters, my parents-in-law, Silvinha and Hudson for their continued support which can never be fully acknowledged.

My deepest gratitude goes to my wife Nicia, for all the encouragement, patience and dedication. I am absolutely certain that I would not have done this work without her love and support.

I am very thankful to Anjolina, Ruy, Lucas and Pedro, for having me as a member of their family during my stay in Recife. Their practical and emotional support has been an invaluable asset. I will never be able to properly acknowledge everything they have done to me.

Finally, I wish to gratefully acknowledge the Universidade Federal da Bahia (UFBA) and CAPES for their financial support during the course of this research.

Contents

1	Introduction	1
1.1	Overview of Refinement Calculi	2
1.2	The Co-op Project	4
1.3	An Algebraic Approach to Compilation	5
1.4	An Overview of ROOL	7
1.5	Compilation Strategy	9
1.6	Overview of Subsequent Chapters	13
2	The ROOL Language and Its Laws	15
2.1	Syntax	15
2.2	Algebraic Laws	21
2.2.1	Laws of Commands	22
2.2.2	Laws of Classes	45
2.3	Summary	52
3	ROOL Virtual Machine	53
3.1	Data Types	53
3.2	<i>RVM</i> Components	54
3.3	The Normal Form	57
3.4	Instruction Set	59
4	Compilation Process	65
4.1	Class Pre-compilation	65
4.1.1	A strategy for normal form reduction	66
4.1.2	Changing visibility of attributes	67
4.1.3	Introducing trivial casts	67
4.1.4	Elimination of method redefinition in our example	68
4.1.5	Proof of Theorem 4.1.1	70
4.2	Redirecting method calls	71
4.2.1	The class <i>L</i>	72

4.2.2	Rules	74
4.2.3	Moving attributes down	78
4.2.4	Redirecting calls in our example	78
4.2.5	Proof of Theorem 4.2.1	80
4.3	Simplification of expressions and some source transformations	80
4.3.1	Eliminating parametrised commands	82
4.3.2	Parametrised Recursion	83
4.3.3	Simplifying Guards	86
4.3.4	Transforming Conditionals	87
4.3.5	Rewriting Assignments	87
4.3.6	Irrelevant sequences of <i>Push</i> and <i>Pop</i>	90
4.3.7	Our example	91
4.3.8	Proof of Theorem 4.3.1	95
4.4	Control Elimination	95
4.4.1	Introducing the Program Counter	96
4.4.2	Recursive commands in L	102
4.4.3	Nested Normal Form	106
4.4.4	Our example	108
4.4.5	Proof Theorem 4.4.1	108
4.5	Data Refinement	111
4.5.1	Proof Theorem 4.5.1	122
4.6	The Compilation Process	122
5	Final Considerations	127
5.1	Related Work	129
5.1.1	Approaches for object-oriented languages	129
5.2	Future Work	130
5.2.1	Mechanisation	131
5.2.2	Code optimisation	131
5.2.3	Language extensions	132
A	Proofs of Compilation Rules for ROOL	133
A.1	Lemmas	133
A.2	Redirection of method calls — Proofs	146
A.3	Simplification of expressions and some source transformations — Proofs	153
A.4	Control elimination — Proofs	161
A.5	Data Refinement — Proofs	176

List of Figures

1.1	The Morris Correctness Diagram	6
1.2	Our Compilation Approach	6
1.3	ROOL program for keeping track of a robot's path	8
1.4	Keeping track of a robot's path	9
1.5	An instance of the ROOL Interpreter	11
3.1	Representation of Data Types	54
3.2	The Class <i>Stack</i>	55
3.3	A frame in the ROOL Virtual Machine	55
3.4	The Constant Pool	56
3.5	The Classes <i>CdsHierarchy</i> and <i>ClassInfo</i>	57
3.6	The ROOL Interpreter ($cds_{RVM} \bullet I$)	58
4.1	Program obtained after introducing trivial casts	68
4.2	Class <i>Path</i> without references to super	69
4.3	Method <i>getLength</i> as a method without redefinitions	70
4.4	The pattern of a method with value parameters in the class <i>L</i>	73
4.5	The pattern of a method with result parameters in the class <i>L</i>	73
4.6	Method <i>lgetLength</i> declared in <i>L</i>	74
4.7	An algorithm to guide the application of Law 2.2.80	78
4.8	Main command obtained after redirecting method calls	79
4.9	Class <i>L</i> generated in the phase of Redirection of Method Calls	81
4.10	Class <i>L</i> after the elimination of parametrised commands	84
4.11	Intermediate main command obtained during the third compilation phase	92
4.12	Resulting main command obtained after the third compilation phase	92
4.13	Methods <i>lsetDirection</i> and <i>lsetLength</i> after the third compilation phase	93
4.14	Method <i>laddStep</i> after the after the third compilation phase	93
4.15	Method <i>lgetLength</i> after the third compilation phase	94
4.16	Intermediate program obtained before the elimination of <i>L</i>	102
4.17	Intermediate program before the elimination of method calls	107
4.18	Program with nested normal form main commands	107

4.19	General structure of our example after the Control Elimination phase	109
4.20	Sequence of guarded commands corresponding to the Main Command	109
4.21	Sequence of guarded commands corresponding to the Method <i>lgetLength</i>	110
4.22	An object in the abstract space	112
4.23	An object in the concrete space	113
4.24	An example of <i>encode</i> and <i>decode</i> methods	115
4.25	Sequence of bytecode instructions corresponding to the Method <i>lgetLength</i>	123
4.26	Main command after the Data Refinement	124

List of Tables

2.1	Grammar for expressions	17
2.2	Grammar for left expressions	18
2.3	Grammar for commands	18
2.4	Grammar for parametrised commands	19

Chapter 1

Introduction

The concern with correctness of computer programs has increased due to their use in applications where failure could result in unacceptable losses. Correct compilers address the translation of source programs into a target language, with a guarantee that the semantics is preserved.

In the literature one can find several approaches that have been used to generate correct compilers for imperative languages [95, 96, 72, 83, 93, 64]. Moreover, the design of correct compilers for procedural languages is already well understood. The main challenge now is the development of an approach to deal with object-oriented features. Even though object-oriented programming has become very popular in recent years, most applications are developed using informal methods.

The purpose of this thesis is to describe an algebraic approach to construct a provably correct compiler for an object-oriented language called ROOL (for Refinement Object-oriented Language) [12, 19], which is similar to sequential Java [44] and C++ [91]. Our language was devised not only for the purpose of compilation, but also to study refinement of object-oriented programs in general. Besides classes, inheritance, dynamic binding, recursion, type casts and tests, and class-based visibility, it also includes specification constructors like those of Morgan's refinement calculus [68].

Our approach to compilation is inspired on that first described in [53], and further developed for imperative programs in [87]. It characterises the compilation process within a uniform framework, where comparisons and translations between semantics are avoided. This is the main advantage of our approach, which accounts for much of its simplicity.

We tackle the problem of compiler correctness by reducing the task of compilation to that of program refinement. Compilation is identified with the reduction (through refinement laws) of a source program, written in an executable subset of ROOL, to a normal form. Our normal form is an interpreter-like program which emulates the behavior of the target machine. From this interpreter we capture the sequence of generated instructions.

Refinement laws are used as compilation rules to reduce the source program to the normal form. Each compilation rule can be proved correct with respect to the algebraic laws of the language. The correctness of the compiler follows from the correctness of each refinement step. The immediate

correctness criterion is that the source program is refined by the interpreter.

In this thesis, we present compilation rules for ROOL and describe how compilation is carried out using them. The compilation process is split into five phases: Class Pre-compilation, Redirection of Method Calls, Simplification of Expressions, Control Elimination, and Data Refinement. Only the code generation phase of a compiler is addressed, parsing and semantic analysis are not considered.

This work is in the context of the Co-op (Calculus of Object-Oriented Programming) project, which aims to produce a refinement calculus for object-oriented development. It comprises the definition of a formal semantics for ROOL and the proposal and proof of basic, design, and compilation laws.

The remainder of this chapter is structured as follows. In the next section, we give an overview of refinement calculi. In Section 1.2 a brief introduction to the Co-op project is provided. After that, in Section 1.3, we provide a comprehensive view of our algebraic approach to compilation. In Section 1.4, we give an overview of our object-oriented language, ROOL. In Section 1.5, we then explain our compilation strategy for ROOL. Finally, we examine the contributions of this thesis and provide an outline of the chapters to come.

1.1 Overview of Refinement Calculi

Refinement calculi are logical frameworks for reasoning about correctness and refinement of programs. Lattice theory and higher-order logic [10, 30] together form a mathematical basis for these calculi. They allow us to prove the correctness of programs and to calculate program refinements in a rigorous, mathematically precise manner.

Earlier work on program verification [74, 39, 50] established a mathematically sound basis for studying program construction methods. The origins of the refinement calculi have emerged from Dijkstra’s seminal work [31] and an influential paper by Wirth [99]. They put forward the method of program derivation known as stepwise refinement. Modular program development by stepwise refinement has been an active research area in computer science, as we can see in the related literature [51, 62, 43, 17, 63, 97, 98, 9].

A primary application of the refinement calculi is the derivation of an imperative program that satisfies a given specification. In the derivation of programs from specifications, constructions of programming languages progressively replace certain patterns of specifications. A derivation typically begins with a non-executable specification, and advances through several intermediate forms containing executable and non executable constructs, ending up with a program having only executable constructs.

At each stage of the refinement process, the current program, or some sub-program of the current program, is transformed by applying a refinement law to it. Each refinement step is required to preserve correctness and yields a program that in some ways is better than the original one.

The refinement calculi is the formalisation of the stepwise refinement approach to program

development. It was originally described in [4]. Back first extended Dijkstra’s language of guarded commands with specifications. In this extension of Dijkstra’s calculus, the definition of the relation of correct refinement was studied, and a simple base language for program and specification was proposed. Specifications and programs shared equal status during program derivation. Thus, the problem of comparison between semantics is avoided from the beginning; programs appear as a subclass of specifications: a single notation is adopted both for programming and for specification.

Once the space of programs is embedded within a more general space of specification, the derivation task is reduced to one of transformations of specifications, within an uniform framework. Whenever a refinement step is performed, the need to check whether an appropriate relation holds between the semantics of the specification and the semantics of the executable construct introduced in its place arises. If these semantics are defined using different formalisms, then the task involves comparison between semantics, which can be very complex.

Further work has also integrated specifications and programs. For instance, Dijkstra’s language is also extended with specification statements in the refinement calculi developed by Hehner [49], Morgan [68], Morris [70], and Nelson[75].

The central concept in the refinement calculi is the refinement relation. Programs are identified with the elements of the predicate transformer lattice [5, 6]. The partial order of the lattice (in other words, the refinement order) relates two programs that may be at different abstraction levels. If $P_1 \sqsubseteq P_2$, then we say that program P_1 is refined by program P_2 . This means that P_2 is a correct implementation of P_1 for any correctness criteria satisfied by P_1 . This is the basis for the formalisation of the stepwise refinement.

Two mathematical properties are of fundamental importance in the formalisation of the stepwise refinement. The first one is known as compositionality; it corresponds to the monotonicity of the language constructs with respect to the ordering on the predicate transformer lattice. If $S[T]$ is a program statement, where S is a context that contains a substatement T then

$$T \sqsubseteq T' \Rightarrow S[T] \sqsubseteq S[T'] \text{ for any statement } T'$$

Thus, in any context, a substatement can always be replaced by its refinement. The other property is transitivity; we start from the program P_0 that satisfies some correctness criteria, then we derive a sequence of successive refinements

$$P_0 \sqsubseteq P_1 \sqsubseteq \dots \sqsubseteq P_n$$

Since each refinement step preserves the correctness of the previous step, we have that

$$P_0 \sqsubseteq P_n$$

Hence, despite the number of steps in the derivation process, the final program always satisfies the original specification.

A specification statement, like in [68], is a notable example of inclusion of specification in a program language. These statements increase expressiveness power, specifying parts of a program yet to be developed.

1.2 The Co-op Project

In recent years, object-oriented programming became so popular that it has emerged as the dominant computer programming style. Languages such as C++ [91] and Java [44] are widely used in academia and industry, mostly with informal methods. The reason for the large use of these languages resides in the fact that they offer the promise of reusable software components. Java and C++ offer dynamic binding and a flexible system of type-checking, which makes it possible for a component to be reused in many contexts.

The Co-op project consists of a collaborative research effort to devise improved methods for reasoning about programs written in object-oriented languages. It was aimed at extending the refinement calculus to object-oriented languages. The calculus includes laws for program derivation and optimisation, as well as laws for object-oriented class design and class restructuring.

Formalisation explains and justifies informal methods as well as being the basis for rigorous methods and tools. Formal methods of development provide a rigorous foundation for informal ones. Formalisation of modular reasoning is both important and challenging.

There has been considerable progress in the design of languages and in formalisation of specification methods. Formalisation has played an important role in language design; earlier formal studies of abstraction [28, 29, 51, 84] have benefited contemporary languages. Similarly, we can observe that results from theoretical studies [38, 2, 77, 73, 16] are playing a significant role in current deliberations on the future of Java. The implementation of assertions in Java is an example of such influence.

The challenge of reasoning about object-oriented programs is associated with the dynamic binding of methods: the version of a method that is invoked cannot be determined from the program text. During runtime the object can belong to a subclass of its declared class, which may have overridden methods.

Due to dynamic binding, the full potential of reuse is difficult to achieve because a component is allowed to have a complicated interaction with the context in which it is used. It is difficult to gain sufficient understanding of such contexts to reason convincingly about a component in isolation. It is also difficult to design interfaces so that, in a particular context, a component's behavior can be understood as a black box, without reference to its internals. Such modular reasoning is essential for the fulfillment of the promise of reusability.

It is essential to think about the components of a program in isolation from its use. For modular reasoning to be possible, extensions of a class must be behavioral subclasses, which means that all overriding methods satisfy the specification of the original method. The main contribution that is pursued involves the briefing and the formalisation of both this notion of behavioral subclassing, and the appropriate specification techniques for this approach [61, 58, 57, 56].

For the development of programs, it is necessary to establish a formal connection between specifications and programs. Mathematical formalisation is an essential foundation for methods

of specification and development of software. Some experimental steps have already been given in this direction [61, 67], but with strong limitations. The experience has shown that it is necessary to establish a basic model, rich enough to deal with a great variety of language constructions and specification notations.

The approach taken in the Co-op project to formalise the development of object-oriented programs is based on the refinement calculus [68, 9, 70]. To extend the refinement calculus for object-oriented development, it is necessary to deal with the additional features inherent to the language. Moreover, an appropriate theory of refinement for classes must be developed, in order to capture behavioral subclassing. The development of the Co-op project entails the following tasks.

- The definition of a tractable language (ROOL) that includes constructions in an appropriate form for theoretical study;
- the construction of a semantic model for the language, based on weakest preconditions, appropriate to validate principles of reasoning in terms of the operationally observable behaviour;
- the proposition and validation of basic and derivable laws;
- the application of the laws in case studies;
- proposal and proof of compilation rules, by means of correctness-preserving transformations for the executable subset of ROOL.

This thesis is concerned with this last task.

1.3 An Algebraic Approach to Compilation

The issue of compiler correctness surfaced together with the first compiler implementations. It is at the heart of all quality problems of software development: if commercial compilers are not free of bugs, no matter what measures are taken for improving the quality of software, these compilers can always produce erroneous executables.

The notion of modelling program compilation as a formal development process is by no means new. The first attempt used operational semantics to prove the correctness of a compiler for a simple expression language; this original work was undertaken by McCarthy and Painter [64].

The work of Burstall and Landin [18] gave rise to the algebraic approach to compilation. This work has influenced many researchers. For example, from this seminal work, Morris [69] characterises the notion of correctness by the commutativity of diagrams such as the one presented pictorially in Figure 1.1. In this diagram, the nodes are algebras and the arrows are homomorphisms. The compiler is defined via mappings between the source and target languages, and their semantics. The compiler correctness is established by proving that the diagram created by these mappings commutes.

Figure 1.1: The Morris Correctness Diagram

Despite the differences in styles to define source and target languages, all approaches based on Morris diagram (Figure 1.1) share a common characterisation of the compiler correctness. What changes is just the way the correctness diagram components are defined in each case; the overall structure is the same. Very similar characterisations of the notion of compiler correctness appears in the approaches based on denotational semantics [71, 81, 85, 94, 80]. Abstracting from what is assigned to the nodes and to the arrows of the correctness diagram in each case, the components are the same.

Figure 1.2: Our Compilation Approach

Experience has shown that comparisons (or worse, translations) between the semantics of the source and the target languages constitute an intricate task. The algebraic approach originally described in [53] (and further developed in [87]) for designing compilers for procedural languages,

can be viewed as an effort towards simplifying the problem of compiler correctness.

The desired simplicity comes from the characterisation of the compilation process within a uniform framework where the comparisons and translations between semantics are avoided. All the calculation necessary to assert the correctness of the compilation process is carried out within a single framework of a procedural language whose semantics is given by algebraic laws.

In this work we extend these results to an object-oriented language. Our language, called ROOL, is similar to Java and C++ and has been specially designed to support reasoning about object-oriented programs and specifications.

In Figure 1.2, the single arrow expresses the refinement relation ($\sqsubseteq$), which justifies the refinement steps from an arbitrary program in the source language to our particular normal form, written in the same language. This normal form represents an interpreter simulating the ROOL Virtual Machine executing a corresponding target program coded as a bytecode stream. Once we achieve the normal form, it is possible to capture the bytecodes that represent the target program.

1.4 An Overview of ROOL

ROOL is an object-oriented language based on sequential Java, but with a copy rather than a reference semantics. The copy semantics adopted for elements of primitive types is applied to objects as well. Considering two variables a and b of type *Employee*, for example, when the assignment $a := b$ is executed, a new object is created in the heap and it stores in a a fresh copy of the value stored in b . Therefore, changes in b do not affect a and vice-versa.

After executing the following code, the object in a is supposed to have its attribute age with value 15, whereas the object in b will have its attribute age with value 10.

```
a.SetAge(10);  
b := a;  
a.SetAge(15);
```

The copy semantics does simplify the language semantics and reasoning, but it makes impossible the modelling of references or sharing. Nevertheless, the laws of ROOL related to object-oriented features do not rely on copy semantics. This makes ROOL still useful for exploring a comprehensive set of laws. Besides, in the absence of sharing, the laws presented in this work are valid in a reference semantics as well.

ROOL has been specially designed to allow reasoning about object-oriented programs and specifications. Specifications are treated as abstract programs so that a single notation is used both for programming and for specification. Actually, the space of programs is embedded within the more general space of specifications. Programs appear as a subclass of specifications. Therefore the derivation task is reduced to one of transformation of specifications within a uniform framework.

A program in ROOL consists of a sequence *cds* of Java-like class declarations followed by a

main command c , which may contain objects of classes declared in cds . In Figure 1.3, we give an example of an executable program in ROOL that will be used to illustrate our compilation strategy. It simulates a mechanism to keep track of a robot's path. The robot starts in the position $(0, 0)$. Every time it moves, a step of length l is taken towards *north*, *south*, *east*, or *west*. The outcome of this program is the total length of the route described by the robot. An example of changes in the position of the robot is shown in Figure 1.4.

```

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

```

Figure 1.3: ROOL program for keeping track of a robot's path

Two classes are declared in Figure 1.3, *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 values of these attributes can be set using the methods *setDirection* and *setLength*, whereas the length of a step can be retrieved using the method *getLength*. The clause **meth** declares a method. The visibility mechanisms are similar to those of Java. The qualifiers **pri**, **prot** and **pub** are used to declare private, protected and public attributes. For simplicity, all methods are considered public.

Figure 1.4: Keeping track of a robot's path

The class *Path* extends *Step*, introducing the attribute *previous* to hold the preceding steps that outline the robot's path; *Path* is a recursive class. The method *addStep* introduces a step in the path; it first assigns the current path (**self**) to *previous*, and then invokes the two methods *setDirection* and *setLength* to record the current step. The length of a path is calculated by the method *getLength*, a recursive redefinition of the method with the same name declared in *Step*. Each recursive invocation of *getLength* visits a step in the path; it traverses the list of steps accumulating the length. The sequence of nested invocations ends when the first step is reached: the value of *previous* is **null**. To get the length of the current step, we use a method call **super.getLength** to guarantee that the method declared in *Step*, which is *Path*'s superclass, is invoked.

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 finally, the free variable *out* represents the output of the program, which is the length of the path described by the robot. The main command consists of 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.

1.5 Compilation Strategy

In this section we give an overview of the compilation strategy, characterising the five phases that form the compilation process.

The main advantage of our approach is the characterisation of the compilation process within a uniform framework, where translations or comparisons between semantics are avoided. This is achieved by embedding the target language within the source language. Thus, we tackle the problem of compiler correctness by reducing the task of compilation to that of program refinement.

Refinement laws are used as compilation rules to reduce the source program to a normal form. Our normal form is an interpreter-like program written in ROOL itself, which emulates the behavior of the target machine executing the target code. From this interpreter we capture the sequence of generated instructions. Figure 1.2 illustrates our compilation scheme.

We define a ROOL Virtual Machine (RVM) as our target. The RVM has the following components:

- PC — a program counter
- S — an operand stack
- F — a frame stack
- M — a store for variables
- CP — a constant pool
- Cls — a data structure with the source program class hierarchy

In Figure 1.5, we present an example of a program in our particular normal form. The normal form consists of a sequence of class declarations (cds_{RVM}) followed by a main command. Our normal form is a partial evaluation of ROOL interpreter, modelling a cyclic mechanism that executes one instruction at a time. Every cycle fetches the next instruction to be executed and simulates its effect on the internal data structures of the interpreter. We use two symbol tables, Ψ which maps each variable declared in the source program to the address of the memory storage M allocated to hold its value, and Φ which maps each constant, class attribute, and class name of the source program to elements in the CP (constant pool). The purpose of CP is to hold the RVM 's representation of constants and class elements (class names, attribute names and attribute types) that appear in the source program.

The variable PC is used for scheduling the selection and sequencing of instructions. The **while** statement is executed until PC reaches a value beyond the interval determined by s and f , which mark the start and the end of the target program. The body of the **while** tests the PC value and selects the instruction to be executed. All instructions modify PC .

The design of a compiler in our approach is actually an abstract design of a code generator. The sequence of bytecodes is represented by the set of guarded commands, which is an abstract representation of the target code. For instance, the instruction $load(n)$ pushes a value indicated by the index n onto the operand stack S . Actually, $load(n)$ is just an abbreviation of a sequence of ROOL commands that simulates its effect on the interpreter's data structure. It is defined as part of the specifications of the instruction set of the RVM , explained in Section 3.4.

The compilation process is justified by reduction theorems. It comprises five main phases: Class Pre-compilation, Redirection of Method Calls, Simplification of Expressions and Some Source

```

 $cds_{RVM} \bullet$  var  $PC : Int; S, F : Stack$  •
     $S := \mathbf{new} \text{ Stack}; F := \mathbf{new} \text{ STack};$ 
     $PC := s;$ 
    while  $PC \geq s \wedge PC < f \rightarrow$ 
        if  $PC = s \rightarrow \text{store}(\Psi_{mtd})$ 
             $\square PC = s + 2 \rightarrow \text{store}(\Psi_o)$ 
             $\square PC = s + 4 \rightarrow ldc(\Phi_0)$ 
             $\square PC = s + 6 \rightarrow \text{load}(\Psi_{mtd})$ 
             $\square PC = s + 8 \rightarrow cmpeq$ 
             $\square PC = s + 9 \rightarrow \text{store}(\Psi_{b_1})$ 
             $\square PC = s + 11 \rightarrow \text{load}(\Psi_o)$ 
             $\square PC = s + 13 \rightarrow \text{instanceof}(\Phi_{Path})$ 
        fi
    end

```

Figure 1.5: An instance of the ROOL Interpreter

Transformations, Control Elimination, and Data Refinement. For each phase, a theorem asserts the expected outcome, and a main theorem links the intermediate steps and states the correctness for the entire process. It guarantees that the normal form embeds a correct implementation of the source program. The correctness of the main theorem is justified by the reduction theorems which have been proved correct from the basic laws of ROOL. In the following, we anticipate the main theorem.

Theorem 1.5.1 (Compilation Process) *Let $cds \bullet c$ be an executable source program, Ψ a symbol table that maps each variable of $cds \bullet c$ to the address of the memory storage M allocated to hold its value, and Φ a symbol table mapping each constant, class attribute, and class name in the abstract data space (source program) to objects in CP (constant pool) that correspond to these elements in the concrete data space. Moreover, there is a set of guarded commands GSC in which every guard has the form $PC = k \rightarrow p$, where $s \leq k < f$ and p is a sequence of ROOL instructions that simulate the effect of a RVM bytecode instruction in the data structure of our normal form. Then we have*

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

The symbol $\sqsubseteq$ represents the refinement relation; this theorem guarantees that the normal form embeds a correct implementation of the source program. Its constructive proof characterises the compilation process, discussed in Chapter 4.

Since the source program operates on a data space different from that of the normal form, it does not make sense to compare them directly. A function $\hat{\Psi}$ performs the necessary change of

data representation. The symbol table Ψ maps the variables declared in the source program to addresses in the store M , in such a way that $M[\Psi_x]$ holds the value of x .

The first objective is to transform all method calls to a unique pattern, where the invoked method is always a method in a class L . The introduction of L is just an artifice to establish a unique pattern to method calls, reducing the complexity involved in the reduction of method bodies to our particular normal form.

The theorem in the Class Pre-compilation phase establishes that the compilation rules applied in this phase are sufficient to end up with a program in a form where

- all attributes in *cds* are made public;
- all targets are cast with its static type;
- all method redefinitions are eliminated.

The theorem in the Redirection of Method Calls phase states that the transformations applied are sufficient to end up with a program in which all method calls are (redirected) to methods in L , and the method declarations in *cds* are eliminated since they become useless.

In the Simplification of Expressions and Some Source Transformations phase, the theorem establishes that

- all parametrised commands are eliminated;
- all parametrised recursive calls are eliminated;
- all method redefinitions are eliminated;
- every conditional is written as an *if-then-else*;
- each guard is a boolean variable or its negation;
- all assignments is rewritten to operate exclusively through the operand stack;
- all attributes are moved to the classes in which they were originally declared. (They have been moved during the Class Pre-compilation.).

The next task is that of Control Elimination, which eliminates the class L and reduces the nested control structure to a single flat iteration, like that of the target program.

Finally, the remaining phase is that of Data Refinement, which replaces the abstract space of the source program with the concrete state of the target machine. This means that all references to variables, methods, attributes, and classes declared in the source program are replaced with the corresponding ones in the target machine. The theorem associated to this phase states that there is an equivalent program that operates exclusively on the concrete space.

To carry out the change of data representation, we use the distributivity properties of the function $\hat{\Psi}$; it is a polymorphic function that applies to programs and commands, and distributes over the commands in the main command, applying a function with the same name. The function $\hat{\Psi}$ does not affect the classes used to define our interpreter (cds_{RVM}), the components of our target machine, and commands that have no reference to variables or classes affected by $\hat{\Psi}$.

The proof of the main theorem is almost a direct consequence of theorems that characterise each phase. Compilation is achieved by applying the compilation rules following the strategies indicated in the proof of each of these theorems.

1.6 Overview of Subsequent Chapters

The rest of this thesis is organised as follows.

Chapter 2 presents the language ROOL and its laws. These algebraic laws characterise an axiomatic semantics for ROOL, stating properties of programming constructs. They are expressive enough to formally derive behaviour preserving program transformations, illustrated in this work by the derivation of our provably-correct compiler. The vast majority of the laws have been proved from a weakest pre-condition for ROOL [24].

Chapter 3 explains the ROOL Virtual Machine, our target machine. The target machine components are explained, and the instruction set is presented. The behavior of the target machine is given by an interpreter which models a cyclic mechanism which executes one instruction at a time. Each instruction is defined in terms of a sequence of ROOL commands.

Chapter 4 discusses a refinement strategy for the compilation of the ROOL executable constructs. The purpose of this chapter is to provide a detailed description of the compilation process, discussing how refinement laws are used as compilation rules to reduce the source program to a normal form that models an interpreter running the target code.

Chapter 5 concludes this thesis with a summary of what has been achieved, some related work, and an outline of future works that remain unexplored.

Appendix A provides the proof of the compilation rules presented in Chapter 4, showing how each compilation rule is proved correct with respect to the algebraic laws of the language.

Chapter 2

The ROOL Language and Its Laws

In this chapter we present the language ROOL, an acronym for Refinement Object-oriented Language, and its algebraic laws. Essentially, this language can be viewed as an object-oriented language in the style of Java [44], though having two major distinct features:

- instead of a reference semantics for objects, it has a copy semantics for both objects and elements of primitive types;
- in the style of Morgan’s refinement calculus [68], our language supports both programs and specifications.

ROOL adopts a copy semantics, while most practical object-oriented languages are based on reference semantics. Thus, references or sharing can not be modelled. Our object values are tuples with recursive nesting but no sharing. It simplifies language semantics and still allows us to reason about the subset of Java programs that do not have reference aliasing.

Refinement calculus is based on a unified language of specification, design and programming. The ROOL language was specially designed to support reasoning about object-oriented programs and specifications; hence it mixes both kind of constructs, so that a single notation can be used for programming, design, and specification. Specifications are treated as abstract programs, they consist of non-executable constructs.

This chapter is organised as follows. In the next section we provide the syntax of ROOL. Finally, in Section 2.2, we introduce the basic algebraic laws of ROOL. These laws help clarify aspects of the semantic of ROOL and serve as a basis for deriving more elaborate laws for practical application of program transformation, such as the compilation rules.

2.1 Syntax

A program in ROOL is written as $CDS \bullet c$, which consists of a sequence of class declaration (CDS), followed by a main command (c).

Similar to Java, ROOL is an object-oriented language that includes classes, inheritance, dynamic binding, recursion, assignment, type casts and tests, and class-based visibility, and many other imperative features. Furthermore, it also includes the specification statements of refinement calculus.

Class Declarations

Class declaration serves to specify the behaviour of instances of objects through use of method declarations, whereas the main command determines the initial execution point of a program. 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  $m \hat{=} (pds \bullet c)$ }*    //public methods
  {new  $\hat{=} c$  end; }*         //Initialisers
end

```

Classes can be recursive in that attributes and method parameters of a class N can have type N . Methods are not allowed to be mutually recursive, but classes are. Due to the copy semantics, the language does not support references, and therefore, mutually recursive methods would not be able to reference the same object because each one would have access to a different copy of the object. Thus, design patterns such as the Observer [41], where objects mutually refer to each other, are not implementable in ROOL.

The built-in **object** class is a superclass of all other classes. Subclassing and single inheritance are implemented through the clause **extends**. It determines the immediate superclass of the declared class N_1 . Whenever it is omitted, **object** is regarded as the superclass.

The visibility mechanism is similar to that of Java: the qualifiers **pri**, **prot**, and **pub** are used for private, protected, and public attributes. The clause **meth** declares a method. For simplicity, all methods are considered to be public. The list of parameters of a method is separated from its body by the symbol ‘ $\bullet$ ’.

The **new** clause declares initialisers: a syntatic sugar for methods that are called after creating objects of the corresponding class. Differently from Java, in our language they have no parameters.

Even though ROOL does not have interface or abstract classes, a method can play a role semantically similar to abstract methods in java when it is defined as **abort**. In essence, doing so we are specifying that nothing can be assumed about how subtypes implement (redefine) that method.

The convention adopted here is that data types T are the types of attributes, method parameters, local variables, and expressions. They are either primitive (like boolean or integer) or class

names. For variable identifiers, we use x , whereas f stands for a literal or built-in function; we also use b for boolean expressions, and X for a recursive block identifier.

Expressions

The expressions e comprise those typically found in object-oriented languages. Table 2.1 describes the rules to generate expressions.

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

Table 2.1: Grammar for expressions

The **self** and **super** references are similar to the *this*, and *super* of Java; the reference **self** refers to the implicit object the method should operate on, whereas the reference **super** denotes the superclass of the current method. Considering that **self** is not optional, when accessing the attribute a of the current class, we must write **self**. a .

The type test expression **is** corresponds to *instanceof* in Java and does not require exact type matching. A type cast is represented by $(N)e$, whose value is the object denoted e , if it belongs to the class N . Whenever a value of e is not of the dynamic type of N , $(N)e$ is an error. Attribute selection $e.x$ can be a run-time error if the value of e is **null**.

An update expression has the form $(e_1; x : e_2)$ and denotes a fresh object copied from e_1 , but with attribute x mapped to a copy of e_2 . Despite its name, the update expression has no side effects whatsoever, just like the other expressions. Variables are updated through execution of commands, such as the assignment $o.x := e$, which is semantically equivalent to $o := (o; x : e)$, and updates the attribute x of the object o .

$$\begin{aligned}
le \in Le & ::= le1 \mid \mathbf{self}.le1 \mid ((N)le).le1 \\
le1 \in Le1 & ::= x \mid le1.x
\end{aligned}$$

Table 2.2: Grammar for left expressions

The left expressions are those that can appear as targets of assignments, method calls, and as result arguments. Also, they can appear along with **self**, **super**, and cast expressions. In Table 2.2, we identify the *Le* subset of *Exp*.

Commands

The imperative constructs of ROOL are based on the language of Morgan’s refinement calculus, which extends Dijkstra’s language of guarded commands [32]. Therefore, the body of methods and constructors are commands similar to those described in [68]. Table 2.3 describes their syntax. Actually, we extend ROOL syntax for commands with constructors for dynamic declaration (**dvar** and **dend**) explained below.

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

Table 2.3: Grammar for commands

For convenience, we allow x , le and T to also represent list of identifiers, expressions, and types. An assignment has the form $le := e$, where e is an arbitrary expression. The specification statement $x : [pre, post]$ describes a program that, when executed in a state that satisfies the precondition pre , terminates in a state that satisfies the postcondition $post$, modifying only variables in x . We call x a frame, and it stands for a finite sequence of variables identifiers.

The usual sequential composition of commands c_1 and c_2 is indicated by $c_1; c_2$. The alternation command is composed by a collection of guarded command $b_i \rightarrow c_i$ separated by $\square$, as in Dijkstra's language [32].

The difference between the two kinds of variable blocks is the way in which the variables are initialised; in a **var** block, the initial value is arbitrary, whereas in a **avar** block, it is angelically chosen to make sure that c succeeds, if it is possible at all. The variables introduced in a **avar** block are angelic variables, also known as logical variables or logical constants. Angelic variables are not part of the executable subset of ROOL, because they are not implementable, but they are useful for reasoning.

As explained above, we expand the syntax of ROOL introducing dynamic declaration, following the same approach adopted in [87]. For reasoning purpose, it is useful to have independent constructors to introduce a variable and to end its scope, especially in the Data Refinement phase. The semantics of these constructors are given solely by their algebraic rules. The construct **dvar** introduces x with a dynamic scope semantics. Operationally, the idea is to associate an unbounded stack with each variable, so that rather than creating a new variable, **dvar** x has the effect of pushing the current value of x into an implicit 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 . On the other hand, 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 seen as parametrised commands $pc(e)$, which can be applied to a list of arguments to yield a command, in the style of Back [7, 20]. Once **self** is not optional in our language, a call to a method m on the current object must be written as **self**. m . In case of redefinitions, we can call the method m declared by the superclass by writing **super**. m .

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

Table 2.4: Grammar for parametrised commands

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-copy: call-by-value and call-by-result. Table 2.4 describes the syntax of parametrised commands.

Definition 2.1.1 (While definition)

$$\begin{array}{l} \text{while } b \rightarrow c \text{ end} \stackrel{def}{=} \text{rec } X \bullet \\ \quad \text{if } b \rightarrow c; X \square \neg b \rightarrow \text{skip fi} \\ \text{end} \end{array}$$

The block **rec** $X \bullet c$ **end** introduces a recursive command named X with body c ; occurrences of X in c are recursive calls. Even though ROOL does not include a **while** statement, it can be easily defined in the standard way using recursion. In Definition 2.1.1, we show how the while statement is defined in ROOL. It is merely syntactic sugar to improve conciseness and readability.

Specifications

From the theoretical point of view, ROOL can be regarded as a complete lattice whose ordering is a refinement relation on specifications. The bottom, represented by **abort**, is the worst possible specification.

$$\text{abort} \stackrel{def}{=} x : [\text{false}, \text{true}]$$

The precondition **false** implies that **abort** has the most undefined possible behaviour: it is never guaranteed to terminate, and when it does the outcome is completely arbitrary (postcondition **true**).

The other extreme is represented by **miracle**. It denotes the top of the lattice; it has the most possible defined behaviour and serves to any purpose. On other words, it is the best specification.

$$\text{miracle} \stackrel{def}{=} x : [\text{true}, \text{false}]$$

The precondition **true** states that it can execute in any state and establish as the outcome the impossible postcondition **false**.

In fact, **miracle** is infeasible and cannot be implemented. Clearly, these extremes specifications are not usually intentionally written, they are only theoretical concepts that turn out to be useful for reasoning. For instance, in program derivation or transformation it is useful to assume that a condition b holds at a given point in the program text. This can be characterised as an *assumption* of b , expressed as $\{b\}$, defined as follows.

$$\{b\} \stackrel{def}{=} [b, \text{true}]$$

Note that, if b does not hold, it behaves like **abort**. Otherwise, it behaves like a program that always terminates and leaves the state unchanged, denoted by **skip**.

$$\mathbf{skip} \stackrel{def}{=} [\mathbf{true}, \mathbf{true}]$$

Another important notion is assertion. We express an assertion as $[b]$, where b is a condition on the state.

$$[b] \stackrel{def}{=} [\mathbf{true}, b]$$

It can be regarded as a miraculous test: it behaves like **skip** if the b holds; otherwise, it behaves like **miracle**.

The weakest possible characterisation of a program that always terminates is given by

$$x : [\mathbf{true}, \mathbf{true}]$$

Although similar to **skip**, it is allowed to assign to variables x . Any program which terminates successfully must refine such specification pattern, instantiating x with the program global variables.

As a reasoning language, ROOL yields a framework where both the programming and the specification operators have the same status in the way that they can be viewed as predicate transformers. In this framework, specifications have the status of abstract programs; they are 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, such as the angelic variable block and the specification statement, are not implementable, the compilation is restricted to the executable subset of ROOL.

2.2 Algebraic Laws

The method of postulating algebraic laws of a given language can be criticised in the sense that it can give rise to complex and unexpected interactions between programming constructions. To avoid this danger, the algebraic semantics is linked with a mathematical model in which the laws can be proved. The majority of the laws presented here have already been proved correct with respect to the weakest precondition semantics of ROOL. These proofs can be found in [24].

Here we present algebraic laws of ROOL for both small and medium grain constructs: the laws of commands consider the small grain constructs, whereas laws of classes consider the medium grain constructs. The basic laws of ROOL first appeared in [12], [11], and [26]. These laws have been further extended in [25, 14, 13, 24]. In chapter 4, we show how the compilation rules are formally derived from these basic laws.

2.2.1 Laws of Commands

The laws of commands of ROOL are similar to the laws of imperative languages presented in, for instance, [37, 86, 68]. Such laws are compositional, and have been used as rewrite rules and program transformations. Nevertheless, ROOL has some commands whose syntax and semantics differ from the languages explored in the cited works. For example, there are commands that support object oriented features such as method calls, whose behaviour cannot be defined by the well known copy rule (for procedure call elimination), due to dynamic binding. Another example resides in the laws presented in the literature to deal with conditionals: they are appropriate for more restricted versions of conditional, rather than our version, whose body is a guarded command, as proposed by Dijkstra [32].

Some laws are independent of a particular context, others are inherently context dependent, especially those relying on class hierarchies.

Assignment

The assignment of the value of a left expression to itself does not change anything.

Law 2.2.1 ($:= \text{skip}$)

$$(le := le) \sqsubseteq \text{skip}$$

□

The above law is a refinement because when an expression that yields **error** is assigned to a variable, the whole assignment behaves like **abort**.

In fact, such a vacuous assignment can be added to any other assignment without changing its effect.

Law 2.2.2 ($:= \text{identity}$)

$$(le, le_1 := e, le_1) \sqsubseteq (le := e)$$

□

The list of left expressions and the corresponding expressions can be subjected to the same permutation without changing the effect of the assignment.

Law 2.2.3 ($:= \text{symmetry}$)

If i ranges over $1..n$ and π is any permutation of $1..n$, then

$$(le_i := e_i) \sqsubseteq (le_{\pi(i)} := e_{\pi(i)})$$

□

Later on we will present laws which show how assignment interacts with the various constructs of the language.

Variable Declaration

The notation **var** $x : T \bullet c$ **end** denotes a local variable block in which a list of distinct variables x is declared for use in the program c . Local variables blocks of this form may appear anywhere a command is expected.

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

Law 2.2.4 (*var association*)

If x and y have no variables in common

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

□

Nor does it matter in which order they are declared.

Law 2.2.5 (*var symmetry*)

$$\mathbf{var} \ x_1 : T_1 \bullet (\mathbf{var} \ x_2 : T_2 \bullet c \ \mathbf{end}) \ \mathbf{end} = \mathbf{var} \ x_2 : T_2 \bullet (\mathbf{var} \ x_1 : T_1 \bullet c \ \mathbf{end}) \ \mathbf{end}$$

□

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

Law 2.2.6 (*var elim*)

If x is not free in c , then

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

□

The name of the bound variable can be changed, provided the new name is not used for a free variable.

Law 2.2.7 (*var rename*)

If x_2 is not free in c , then

$$\mathbf{var} \ x_1 : T \bullet c \ \mathbf{end} = \mathbf{var} \ x_2 : T \bullet c[x_2/x_1] \ \mathbf{end}$$

□

If each guarded command of a conditional declares the variable x the declaration may be moved outside the constructor, provided that x does not occur in the guards or in the condition.

Law 2.2.8 (**var** – [] *dist*)

If i ranges over $1..n$ and x is not free in b_i , then

$$\mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow (\mathbf{var} \ x : T \bullet c_i \ \mathbf{end}) \ \mathbf{fi} = \mathbf{var} \ x : T \bullet \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \ \mathbf{fi} \ \mathbf{end}$$

□

It is possible increase the scope of a variable without effect, provided there is no capture of free variables. For example, if the left argument of a sequential composition declares the variable x then the scope can be extended to the right component.

Law 2.2.9 (**var**–; *left dist*)

If x is not free in c_2 , then

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

□

Similarly, 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 2.2.10 (**var**–; *right dist*)

If x is not free in c_1 , then

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

□

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

Law 2.2.11 (**var**– := *final value*)

If x is not free in c , then

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

□

When the two arguments of a sequential composition declare the same variable, they can be combined into a single one, but this may lead to a refinement when the second block uses x without assigning to it.

Law 2.2.12 (*var-; dist*)

$$\mathbf{var\ } x : T \bullet c_1 \mathbf{ end; var\ } x : T \bullet c_2 \mathbf{ end} \sqsubseteq \mathbf{var\ } x : T \bullet c_1; c_2 \mathbf{ end}$$

□

The local variable block assigns an arbitrary value to x . The nondeterminism can be reduced by the initialisation of x .

Law 2.2.13 (*var- := initial value*)

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

□

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

Law 2.2.14 (*var- := method call unit*)

$$\mathbf{var\ } y, z : T_1, T_2 \bullet p; y.m(z); \mathbf{ end} \sqsubseteq \mathbf{var\ } y, z : T_1, T_2 \bullet p \mathbf{ end}$$

□

The following laws allow us to introduce three styles of variable blocks, according to the role played by the introduced variable.

A variable block can be introduced in a value-result style.

Law 2.2.15 (*var block-vres*)

If l is not free in c , then

$$c = \mathbf{var\ } l : T \bullet l := x; c[l/x] \ x := l \mathbf{ end}$$

□

provided l does not occur in c .

The next law introduces a variable block in a value style.

Law 2.2.16 (*var block-val*)

If l is not free in c , then

$$c = \mathbf{var\ } l : T \bullet l := x; c[l/x] \mathbf{ end}$$

□

provided l does not occur in c ; x is not the target of assignments and method calls.

Finally, the following law addresses the introduction of a variable block in a result style.

Law 2.2.17 (*var block-res*)

If l is not free in c , then

$$c = \mathbf{var} \ l : T \bullet c[l/x]; \ x := l; \ \mathbf{end}$$

□

provided l does not occur in c ; x is not the target of assignments, nor does it occur as a value argument in a method call.

Parametrised Commands

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

Law 2.2.18 (*Pcom elimination-val*)

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

□

provided the variables of l do not occur free in c , x and vl .

Law 2.2.19 (*Pcom elimination-res*)

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

□

provided the variables of l do not occur free in c , x and vl .

Dynamic Scope

The commands **dvar** and **dend** obeys laws similar to those of **var** and **end**. Nevertheless, one immediate difference resides in that renaming is not valid in general for dynamic declarations, except when the static and dynamic declarations have the same effect.

Both **dvar** and **dend** are associative. It does not matter whether variables are dynamically declared or ended in one list or singly.

Law 2.2.20 (**dvar**; **dend** *assoc*)

If x and y have no variables in common, then

- (1) $(\mathbf{dvar} \ x : T_x; \ \mathbf{dvar} \ y : T_y) = (\mathbf{dvar} \ x : T_x; \ y : T_y)$
- (2) $(\mathbf{dend} \ x; \ \mathbf{dend} \ y) = (\mathbf{dend} \ x; \ y)$

□

The dynamic scope of a variable can be increased without effect, provided it does not interfere with other free variables.

Law 2.2.21 (**dvar**; **dend** *change scope*)

If x is not free in c , then

- (1) $(c; \ \mathbf{dvar} \ x : T_x) = (\mathbf{dvar} \ x : T_x; \ c)$
- (2) $(\mathbf{dend} \ x; \ c) = (c; \ \mathbf{dend} \ x)$

□

Both **dvar** and **dend** distribute rightward through the conditional, provided that this does not interfere with the guards.

Law 2.2.22 (**dvar**; **dend** – $\square$ *dist*)

If i ranges over $1..n$ and x is not free in b_i , then

- (1) $\mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow (\mathbf{dvar} \ x : T; \ c_i) \ \mathbf{fi} = \mathbf{dvar} \ x : T; \ \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \ \mathbf{fi}$
- (2) $\mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow (\mathbf{dend} \ x; \ c_i) \ \mathbf{fi} = \mathbf{dend} \ x; \ \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow ; \ c_i \ \mathbf{fi}$

□

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

Law 2.2.23 (**dend**- $:=$ *final value*)

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

□

Invoking a method can be irrelevant if the affected variables are just before the end of their scope.

Law 2.2.24 (**dend**- $:=$ *method call unit*)

$$p; \ y.m(z); \ \mathbf{dend} \ y, z \sqsubseteq p; \ \mathbf{dend} \ y, z$$

□

The next law assigns the dynamic declaration semantics to **dvar** and **dend**. It states that **dend** x followed by **dvar** $x : T$ keeps all variables but x unchanged, whereas **dvar** $x : T$ followed by **dend** x has no effect whatsoever. Thus, the pair (**dvar** $x : T$, **dend** x) is a simulation.

Law 2.2.25 (**dend**; **dvar** *simulation*)

$$(\mathbf{dend} \ x; \ \mathbf{dvar} \ x : T) \sqsubseteq \mathbf{skip} = (\mathbf{dvar} \ x : T; \ \mathbf{dend} \ x)$$

□

The following law states that the sequential composition of **dend** x ; with **dvar** $x : T$ has no effect whatsoever if it is followed by an assignment to x that does not rely on the previous value of x .

Law 2.2.26 (**dend**; **dvar** – **skip**)

$$(\mathbf{dend} \ x; \ \mathbf{dvar} \ x : T; \ x := e) = x := e$$

□

The next law asserts that, by using push and pop, one can make explicit the implicit stack that **dvar** and **dend** operate on.

Law 2.2.27 ((*Push*, *Pop*) – (**dvar**, **dend**) *conversion*)

If F is not free in p or q , then

$$\begin{aligned} & (\mathbf{var} \ x : T \bullet p[(\mathbf{dvar} \ x : T; \ q \ \mathbf{dend} \ x)/X] \ \mathbf{end}) \\ = & (\mathbf{var} \ x : T; \ F : \mathit{Stack} \bullet p[(\mathit{Push}(F, x); \ q \ \mathit{Pop}(F, x))/X] \ \mathbf{end}) \end{aligned}$$

□

The notation $p[r/X]$ indicates the replacement of every free occurrence of X in p by r . To avoid the capture of free identifiers of r the local declarations in p are renamed.

Finally, the next law establishes the connection between two forms of variable declaration.

Law 2.2.28 (**dvar**; **dend** *conversion*)

If p is block structured with respect to x , and x has a contiguous scope in p , then

$$(\mathbf{dvar} \ x : T; \ p \ \mathbf{dend} \ x;) = (\mathbf{var} \ x : T; \bullet p \ \mathbf{end})$$

□

In order to satisfy the proviso of the contiguous scope, we require that nested declarations always use distinct names for variables. We also assume that these names are distinct from those used for global variables.

The following law states that we can introduce local declarations of variables. It is based on Law 2.2.28.

Law 2.2.29 (*var introduction*)

If p is block structured with respect to x , and x has a contiguous scope in p , then

$$(x := e; p; x := f;) = (\text{var } x : T; \bullet x := e; p \text{ end}; x := f)$$

□

Proof:

$$\begin{aligned} & \text{LHS} \\ &= \{ \text{Law 2.2.26} \} (\text{dend}; \text{dvar} - \text{skip}) \\ & \quad \text{dend } x; \text{dvar } x : T; x := e; p \text{ dend } x; \text{dvar } x : T; x := f \\ &= \{ \text{Law 2.2.28} \} (\text{dvar}; \text{dend conversion}) \\ & \quad \text{dend } x; \text{var } x : T; x := e; p \text{ end}; \text{dvar } x : T; x := f \\ &= \{ \text{Law 2.2.21} \} (\text{dvar}; \text{dend change scope}) \\ & \quad \text{var } x : T; x := e; p \text{ end}; \text{dend } x; \text{dvar } x : T; x := f \\ &= \{ \text{Law 2.2.26} \} (\text{dend}; \text{dvar} - \text{skip}) \\ & \quad \text{RHS} \end{aligned}$$

◇

These laws are used to prove transformations on source programs during our compilation process. Nevertheless, **dvar** and **dend** are not part of our source language. For this reason, when applying these laws, we assume that the programs always have contiguous scope with respect to any local variable.

Conditional

The first law states that the order of the guarded commands of conditional is irrelevant.

Law 2.2.30 (*if symmetry*)

If i ranges over $1..n$ and π is any permutation of $1..n$, then

$$\text{if } \square_{(1 \leq i \leq n)} b_i \rightarrow c_i \text{ fi} = \text{if } \square_{(1 \leq i \leq n)} b_{\pi(i)} \rightarrow c_{\pi(i)} \text{ fi}$$

□

A conditional having a single guarded command which has a true guard behaves like the command itself.

Law 2.2.31 (*if true guard*)

$$\mathbf{if\ true\ \rightarrow\ c\ fi} = c$$

□

A command with a false guard can be eliminated from the conditional, as it would never be executed.

Law 2.2.32 (*if false guard*)

$$\mathbf{if\ false\ \rightarrow\ c\ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i\ fi} = \mathbf{if\ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i\ fi}$$

□

A conditional without guarded commands behaves like **abort**.

Law 2.2.33 (*if abort unit*)

$$\mathbf{if\ ()\ fi} = \mathbf{abort}$$

□

When all guarded commands are the same, the conditional can be eliminated.

Law 2.2.34 (*if void guards*)

If i ranges over 1..n then

$$\mathbf{if\ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c\ fi} = c$$

□

When the same command has two different guards, these guards can be joined.

Law 2.2.35 (*if - $\vee$ distrib*)

If i ranges over 3..n then

$$\begin{aligned} & \mathbf{if\ } b_1 \rightarrow c \ \square \ b_2 \rightarrow c \ \square_{\langle 1 \leq i \leq n \rangle} (b_i \rightarrow c_i) \ \mathbf{fi} \\ & = \\ & \mathbf{if\ } (b_1 \vee b_2) \rightarrow c \ \square_{\langle 1 \leq i \leq n \rangle} (b_i \rightarrow c_i) \ \mathbf{fi} \end{aligned}$$

□

A particular guarded command can be selected if we know that its guard is true and all the other guards are false.

Law 2.2.36 (*if elim*)

If i ranges over $1..n$, j ranges over $1..m$, and $\neg (b \wedge b_i)$, for all i , then

$$\begin{aligned} & \text{if } b \rightarrow (\text{if } b \rightarrow c \sqcap_{\langle 1 \leq i \leq n \rangle} (b_i \rightarrow c_i) \text{ fi}) \sqcap_{\langle 1 \leq j \leq m \rangle} (b_j \rightarrow c_j) \text{ fi} \\ = & \\ & \text{if } b \rightarrow c \sqcap_{\langle 1 \leq j \leq m \rangle} (b_j \rightarrow c_j) \text{ fi} \end{aligned}$$

□

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

Law 2.2.37 (*if selection*)

If i and j range over $1..n$, $i \neq j$, and $(b_i \wedge b_j) = \text{false}$, then

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

□

The expression $[b_j]$ above is an assertion.

Nested conditionals of the particular form below can be eliminated.

Law 2.2.38 (*if - $\vee$ distrib2*)

$$\begin{aligned} & \text{if } b_1 \rightarrow c_1 \sqcap \neg b_1 \rightarrow (\text{if } b_2 \rightarrow c_1 \sqcap \neg b_2 \rightarrow c_2 \text{ fi}) \text{ fi} \\ = & \\ & \text{if } (b_1 \vee b_2) \rightarrow c_1 \sqcap \neg (b_1 \vee b_2) \rightarrow c_2 \text{ fi} \end{aligned}$$

□

We have a similar effect for conjunction.

Law 2.2.39 (*if - $\wedge$ distrib*)

$$\begin{aligned} & \text{if } b_1 \rightarrow (\text{if } b_2 \rightarrow c_1 \sqcap \neg b_2 \rightarrow c_2 \text{ fi}) \sqcap \neg b_1 \rightarrow c_2 \text{ fi} \\ = & \\ & \text{if } (b_1 \wedge b_2) \rightarrow c_1 \sqcap \neg (b_1 \wedge b_2) \rightarrow c_2 \text{ fi} \end{aligned}$$

□

Some refinement laws for conditional are stated below; these are useful in proving the correctness of the compiler design.

Extending the guarded command set by introducing new guarded commands leads to refinement.

Law 2.2.40 (*if - Guarded command introduction*)

If i ranges over $1..n$ and $\neg (b \wedge b_i)$, for all i , then

$$\begin{aligned} & \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \ \mathbf{fi} \\ \sqsubseteq & \\ & \mathbf{if} \ (\square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i) \ \square b \rightarrow c \ \mathbf{fi} \end{aligned}$$

□

Now we show that the guarded command set can be reduced. Eliminating a guarded command, whose guard is equivalent (or stronger) to at least one of the other guards, might also reduce nondeterminism.

Law 2.2.41 (*if - Guarded command elimination*)

If i ranges over $1..n$ and $b \Rightarrow \bigvee i \bullet b_i$, then

$$\begin{aligned} & \mathbf{if} \ b \rightarrow c \ \square_{\langle 1 \leq i \leq n \rangle} (b_i \rightarrow c_i) \ \mathbf{fi} \\ \sqsubseteq & \\ & \mathbf{if} \ (\square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i) \ \mathbf{fi} \end{aligned}$$

□

One can always chose to execute a command guarded by true, but this possibly reduces non-determinism since there might be other true guards.

Law 2.2.42 (*if - true guard ref*)

If i ranges over $1..n$

$$\mathbf{if} \ \mathbf{true} \rightarrow c \ \square_{\langle 1 \leq i \leq n \rangle} (b_i \rightarrow c_i) \ \mathbf{fi} \ \sqsubseteq \ c$$

□

A guarded command can always be preceded by an assumption of its guards.

Law 2.2.43 (*if - void assertion*)

$$\mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \ \mathbf{fi} \ = \ \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow [b_i]; \ c_i \ \mathbf{fi}$$

□

The following law allows the replacement of a guarded command inside a while.

Law 2.2.44 (*if - replace guarded command*)

Let $R = \text{if } a \rightarrow p \sqcap b \rightarrow q \text{ fi}$.

If $r; \text{ while } (a \vee b) \rightarrow R \text{ end} \sqsubseteq p; \text{ while } (a \vee b) \rightarrow R \text{ end}$, then

$$\begin{array}{l} \text{while } (a \vee b) \rightarrow \\ \quad \text{if } a \rightarrow r \sqcap b \rightarrow q \text{ fi} \quad \sqsubseteq \quad \text{while } (a \vee b) \rightarrow R \text{ end} \\ \text{end} \end{array}$$

□

The next law reduces a conditional to a set of alternations.

Law 2.2.45 (*if - Conditional in the if-then-else style*)

$$\begin{array}{l} \text{if } \sqcap_{\langle k \leq i \leq n \rangle} x_i \rightarrow p_i \text{ fi} \quad \sqsubseteq \quad \text{if } x_k \rightarrow p_k \\ \quad \sqcap \neg x_k \rightarrow \text{If } \sqcap_{\langle k+1 \leq i \leq n \rangle} b_i \rightarrow p_i \text{ fi} \\ \quad \text{fi} \end{array}$$

□

This transformation consists of a refinement because the nondeterminism is eliminated when we impose an order in the guards evaluation.

Sequential Composition

The execution of **skip** always terminates and leaves everything unchanged, to precede or follow a command c by **skip** does not change the effect of c .

Law 2.2.46 (*;- skip unit*)

$$(\text{skip}; c) = c = (c; \text{skip})$$

□

To precede a command by **abort** results in **abort**, because **abort** cannot be relied upon to terminate. So, **abort** is regarded as a left zero of sequential composition.

Law 2.2.47 (*;- abort left zero*)

$$\text{abort}; c = \text{abort}$$

□

To specify the execution of a command c after a **miracle** results in **miracle**; therefore, **miracle** is a left zero of sequential composition.

Law 2.2.48 (*; miracle left zero*)

$$\mathbf{miracle}; c = \mathbf{miracle}$$

□

Sequential composition is associative.

Law 2.2.49 (*; assoc*)

$$(c_1; c_2); c_3 = c_1; (c_2; c_3)$$

□

Sequential composition distributes leftward through the conditional.

Law 2.2.50 (*; if left dist*)

If i ranges over $1..n$

$$\mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \ \mathbf{fi}; c = \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow (c_i; c) \ \mathbf{fi}$$

□

Sequential composition can distribute leftward through the conditional if the guards are not affected.

Law 2.2.51 (*; if right dist*)

If i ranges over $1..n$

$$c; \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \ \mathbf{fi} = \mathbf{if} \ \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow (c; c_i) \ \mathbf{fi}$$

□

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

Law 2.2.52 (*; if selection*)

Provided $(b \Rightarrow \neg c_i)$ and i ranges over $1..n$, then

$$\{b\}; \mathbf{if} \ b \rightarrow p \ \square_{\langle 1 \leq i \leq n \rangle} c_i \rightarrow q_i \ \mathbf{fi} = \{b\}; p$$

□

The sequential composition of two assignments to the same left expression can be easily combined to a single assignment.

Law 2.2.53 ($;$:= *combination*)

$$(le := e_1; le := e_2) = (le := e_2[e_1/le])$$

□

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

Law 2.2.54 ($;$:= *substitution*)

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

□

Assignment commutes with a command in the following sense.

Law 2.2.55 ($;$:= *commutation*)

$$le := e; p = p[e/le]; le := e$$

□

Assignments distributes rightward through a conditional, replacing occurrences of the assigned left expression in the guards by the corresponding expression.

Law 2.2.56 ($;$:= *right dist*)

If i ranges over $1..n$

$$le := e; \text{ if } \square_{\langle 1 \leq i \leq n \rangle} b_i \rightarrow c_i \text{ fi} = \text{ if } \square_{\langle 1 \leq i \leq n \rangle} b_i[e/le] \rightarrow (le := e; c_i) \text{ fi}$$

□

The following Law allows the elimination of an assignment whenever an assertion, or an assumption, ensures that it does not introduce any change in the state of the affected variable.

Law 2.2.57 ($;$:= *void assignment*)

$$\begin{aligned} (1) \quad & [x = y]; x := y = [x = y] \\ (2) \quad & \{x = y\}; x := y = \{x = y\} \end{aligned}$$

□

An assignment can be refined by an assertion. The reason is that an error can occur on the assignment and the final value of the variable might be arbitrary, whereas the assertion does not change the value of any variable.

Law 2.2.58 (*$:=$ refined by assertion*)

$$x := y \sqsubseteq [x = y]$$

□

The sequential composition of two assumptions can be combined to a single assumption.

Law 2.2.59 (*assumption combination*)

$$\{a\}; \{b\} = \{a \wedge b\}$$

□

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

Law 2.2.60 (*$;$:= void assumption*)

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

□

A similar law is valid for assertions.

Law 2.2.61 (*$;$:= void assertion*)

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

□

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 2.2.62 (*$;$ $[b]; \{b\}$ simulation*)

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

□

We can commute the order of execution of an assumption/assertion b and an arbitrary program p , provided b does not refer to variables in p .

Law 2.2.63 (*; $\equiv$ commute assumption*)

$$p; \{b\} = \{b\}; p$$

□

Law 2.2.64 (*; $\subseteq$ commute assertion*)

$$p; [b] \subseteq [b]; p$$

□

Recursion

Fixed point equations are a class of equations that is particularly important in computing. In the literature, they are often called recursive equations. Fixed point calculus involves the solution of recursive equations defined by a monotonic endofunction on a partially ordered set. An endofunction is a function whose target and source belongs to the same set.

Consider X as the name of the recursive program we want to construct. Let $F(X)$ define the intended behaviour of the program, for a given context F . F maps programs from one language to programs written in the same language. If F is defined exclusively in terms of the notations introduced already, it follows by structural induction that F is monotonic:

$$c_1 \subseteq c_2 \Rightarrow F(c_1) \subseteq F(c_2)$$

A fixed point of a function F is often given as the limit of the iterations of F . This is the case for the calculus asserted by Tarski-Knaster theorem [92], which says that $\mathbf{rec} X \bullet F(X)$ is a solution of the equation $X = F(X)$; in addition to that, it is the least solution. A solution to a recursive equation is called “fixed-point” because applying F to the solution leaves the value unchanged.

Law 2.2.65 (*rec fixed point*)

$$\mathbf{rec} X \bullet F(X) = F(\mathbf{rec} \bullet F(X))$$

□

Law 2.2.66 (*rec least fixed point*)

$$F(X) \subseteq Y \Rightarrow \mathbf{rec} X \bullet F(X) \subseteq Y$$

□

The meaning of a parametrised recursive method body is given by the fixed point and the least fixed point laws. For instance, the fixed point law becomes

Law 2.2.67 (*rec fixed point for recursive methods*)

$$\mathbf{rec} M \bullet (\text{par } x:T \bullet p) = \text{par } x:T \bullet p[\mathbf{rec} M \bullet (\text{par } x : T \bullet p)/M]$$

□

Occurrences of M in p are interpreted as recursive calls, all having the form $le.M(t)$, for some target le and appropriate actual parameter t

Considering Y a parametrised program, where $Y = (\text{par } x : T \bullet q)$, then the least fixed point law emerges as

Law 2.2.68 (*rec least fixed point for recursive methods*)

$$(\text{par } x : T \bullet p) \sqsubseteq Y \Rightarrow \mathbf{rec} M \bullet (\text{par } x : T \bullet p) \sqsubseteq Y$$

□

Galois Connection

In mathematics, especially in order theory, a Galois connection is a particular correspondence between two partially ordered sets. The general notion of Galois Connection was first introduced by Ore in 1944 [79]. Since then, it has been used in many contexts. Computer science abound with examples of Galois connections [55, 45, 65, 52, 27].

Let F and G be functions on programs such that, for all programs X and Y

$$F(X) = Y \equiv X = G(Y)$$

Therefore, G is a function that does the reverse of F , so that G is the inverse of F and vice-versa. More formally, if F is a function with domain D , then G is its inverse function if and only if for every X in D we have:

$$G(F(X)) = F(G(X)) = X$$

For a function F to have a valid inverse, it must be a bijection. However, since it is well known that the set of bijective functions are relatively small, the notion of inverse functions becomes rather restricted. As a standard approach to generalise the notion of inverse functions, the definition of approximate inverses is introduced as follows.

Definition 2.2.1 (Approximate inverses)

Let F and F^{-1} be functions on programs, such that for all X and Y

$$F(X) \sqsubseteq Y \equiv X \sqsubseteq F^{-1}(Y)$$

□

The pair (F, F^{-1}) is an example of Galois connection. We call F the weakest inverse of F^{-1} , whereas F^{-1} is the strongest inverse of F . The notion of weakest inverses has been used for top-down design of programs [52, 37]. The use of strongest inverses are less common in the literature. In [42], Gardiner and Pandya have applied the notion of strongest inverses to reason about recursion. In this paper, some properties of strongest inverses are proved. Before we list those properties, we first review the following definitions.

For an arbitrary set of programs $\mathcal{P}$, $\sqcup \mathcal{P}$ denotes the least upper bound of $\mathcal{P}$.

Definition 2.2.2 (Least upper bound)

$$(\sqcup \mathcal{P} \sqsubseteq p) \equiv (\forall X \mid X \in \mathcal{P} \bullet X \sqsubseteq p)$$

□

The least upper bound of the set $\mathcal{P}$ can be replaced by p , if and only if, for all X in $\mathcal{P}$, p refines X .

In a similar way the greatest lower bound, denoted by $\sqcap \mathcal{P}$, is defined.

Definition 2.2.3 (Greatest lower bound)

$$(p \sqsubseteq \sqcap \mathcal{P}) \equiv (\forall X \mid X \in \mathcal{P} \bullet p \sqsubseteq X)$$

□

The leftward distribution of sequential composition through these operators is as follows.

Law 2.2.69 ($;$ - $\sqcup$ and $;$ - $\sqcap$ left distribution)

- (1) $\sqcup \mathcal{P}; p = \sqcup \{X \mid X \in \mathcal{P} \bullet (X; p)\}$
- (2) $\sqcap \mathcal{P}; p = \sqcap \{X \mid X \in \mathcal{P}\}$

□

F is universally conjunctive if for all (possibly empty) sets $\mathcal{P}$.

$$F(\sqcap \mathcal{P}) = \sqcap \{X \mid X \in \mathcal{P} \bullet F(X)\}$$

Similarly, F is universally disjunctive if for all (possibly empty) sets $\mathcal{P}$.

$$F(\bigsqcup \mathcal{P}) = \bigsqcup \{X \mid X \in \mathcal{P} \bullet F(X)\}$$

Theorem 2.2.1 *Strongest inverses*

- (1) If F^{-1} exists then both F and F^{-1} are monotonic.
- (2) F^{-1} is unique if it exists.
- (3) If F^{-1} exists then, for all programs X ,

$$F(F^{-1}(X)) \sqsubseteq X \sqsubseteq F^{-1}(F(X))$$
- (4) F^{-1} exists if and only if F is universally disjunctive; in this case it is defined by

$$F^{-1}(Y) \stackrel{\text{def}}{=} \bigsqcup \{X \mid F(X) \sqsubseteq Y \bullet X\}$$
- (5) F^{-1} is universally conjunctive if it exists.

□

The next lemma states that sequential composition has a strongest inverse in its first argument. As observed by Gardiner and Pandya [42], this can be used to yield a concise proof of an important property about composition of sequential commands. This proof is given later in the next section.

Lemma 2.2.1 *(Strongest inverse of ;)*

Let $F(X) \stackrel{\text{def}}{=} (X; p)$

Then F has a strongest inverse which we denote by $F^{-1}(X) \stackrel{\text{def}}{=} (X \sqcup p)$

Furthermore, for all X , $(X \sqcup p); p \sqsubseteq p$

□

Iteration

Considering that **while** is a derived operator in the ROOL language, rather than postulate some of its properties, we prove them. The objective here is to illustrate how modularity can be achieved from the basic algebraic laws when more elaborated transformations are needed. These proofs can help make the proofs of the compilation rules more intelligible. These compilation rules will be presented as part of our compilation strategy (Chapter 4).

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

Law 2.2.70 (*while elimination*)

If $(b_1 \Rightarrow \neg b_2)$, then

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

□

Proof:

$$\begin{aligned}
& LHS \\
&= \{Definition\ 2.1.1\} \text{ (While definition) and } \{Law\ 2.2.65\} \text{ (rec fixed point)} \\
&\quad [b_1]; \\
&\quad \text{if } b_2 \rightarrow c; \text{ while } b_2 \bullet c \text{ end} \\
&\quad \quad \square \neg(b_2) \rightarrow \text{skip} \\
&\quad \text{fi} \\
&= \{Law\ 2.2.36\} \text{ ((if elim)) and } \{Law\ 2.2.46\} \text{ (;- skip unit)} \\
& RHS
\end{aligned}$$

◇

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

Law 2.2.71 (*while unfold*)

If $(b_1 \Rightarrow b_2)$, then

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

□

Proof:

$$\begin{aligned}
& LHS \\
&= \{Definition\ 2.1.1 \text{ (While definition) } \} \text{ and } \{Law\ 2.2.65\} \text{ (rec fixed point)} \\
&\quad [b_1]; \\
&\quad \text{if } b_2 \rightarrow c; \text{ while } b_2 \bullet c \text{ end} \\
&\quad \quad \square \neg(b_2) \rightarrow \text{skip} \\
&\quad \text{fi} \\
&= \{Law\ 2.2.36\} \text{ ((if elim))} \\
& RHS
\end{aligned}$$

◇

In our proofs, there is a very common step which consists of unfolding a **while** and simplifying the unfolded body when this is a conditional.

Law 2.2.72 (while – if unfold)

Let $R = \mathbf{if} \ a \rightarrow p \sqcap b \rightarrow q \ \mathbf{fi}$. Then

$$[a]; \ \mathbf{while} \ (a \vee b) \rightarrow R \ \mathbf{end} = [a]; \ p; \ \mathbf{while} \ (a \vee b) \rightarrow R \ \mathbf{end}$$

□

Proof:

$$\begin{aligned} & LHS \\ = & \{Law\ 2.2.71\} \ (\mathbf{while} \ \text{unfold}) \\ & [a]; \ R; \ \mathbf{while} \ (a \vee b) \rightarrow R \ \mathbf{end} \\ = & \{Law\ 2.2.36\} \ ((\mathbf{if} \ \text{elim}) \) \\ & RHS \end{aligned}$$

◇

We can eliminate guarded commands within an iteration if the condition of the iteration allows only part of the guards to hold.

Law 2.2.73 (while – if guard elim)

$$\begin{aligned} & \mathbf{while} \ b_1 \rightarrow \\ & \quad \mathbf{if} \ b_1 \rightarrow p \sqcap b_2 \rightarrow q \ \mathbf{fi} \quad \sqsubseteq \ \mathbf{while} \ b_1 \rightarrow p \ \mathbf{end} \\ & \mathbf{end} \end{aligned}$$

□

Proof: Let $R = \mathbf{if} \ b_1 \rightarrow p \sqcap b_2 \rightarrow q \ \mathbf{fi}$

$$\begin{aligned} & LHS \\ = & \{Definition\ 2.1.1 \ (\text{While definition}) \} \ \text{and} \ \{Law\ 2.2.43\} \ (\mathbf{if} \ - \ \text{void assertion}) \\ & \mathbf{rec} \ X \bullet (\mathbf{if} \ b_1 \rightarrow ([b_1]; \ R; \ X) \sqcap \neg(b_1) \rightarrow \mathbf{skip} \ \mathbf{fi}) \ \mathbf{end} \\ = & \{Law\ 2.2.36\} \ (\mathbf{if} \ \text{elim}) \\ & \mathbf{rec} \ X \bullet (\mathbf{if} \ b_1 \rightarrow ([b_1]; \ p; \ X) \sqcap \neg(b_1) \rightarrow \mathbf{skip} \ \mathbf{fi}) \ \mathbf{end} \\ = & \{Definition\ 2.1.1 \ (\text{While definition}) \} \ \text{and} \ \{Law\ 2.2.43\} \ (\mathbf{if} \ - \ \text{void assertion}) \\ & RHS \end{aligned}$$

◇

The following law allows the replacement of a guarded command inside an iteration.

Law 2.2.74 (*while – if replace guarded command*)

Let $R = \mathbf{if} \ a \rightarrow p \sqcap b \rightarrow q \ \mathbf{fi}$.

If $r; \mathbf{while} \ (a \vee b) \rightarrow R \ \mathbf{end} \sqsubseteq p; \mathbf{while} \ (a \vee b) \rightarrow R \ \mathbf{end}$, then

$$\begin{aligned} & \mathbf{while} \ (a \vee b) \rightarrow \\ & \quad \mathbf{if} \ a \rightarrow r \sqcap b \rightarrow q \ \mathbf{fi} \sqsubseteq \mathbf{while} \ (a \vee b) \rightarrow R \ \mathbf{end} \\ & \mathbf{end} \end{aligned}$$

□

Proof:

$$\begin{aligned} & RHS \\ = & \{ \text{Definition 2.1.1 (While definition)} \} \text{ and } \{ \text{Law 2.2.65} \} \ (\mathbf{rec} \text{ fixed point}) \\ & \mathbf{if} \ (a \vee b) \rightarrow (R; RHS) \sqcap \neg(a \vee b) \rightarrow \mathbf{skip} \ \mathbf{fi} \\ = & \{ \text{Law 2.2.69} \} \ (; -\sqcup \text{ and } ; -\sqcap \text{ left distribution}) \\ & \mathbf{if} \ (a \vee b) \rightarrow (\mathbf{if} \ a \rightarrow (p; RHS) \sqcap b \rightarrow (q; RHS) \ \mathbf{fi}) \sqcap \neg(a \vee b) \rightarrow \mathbf{skip} \ \mathbf{fi} \\ \sqsubseteq & \{ \text{Assumption} \} \\ & \mathbf{if} \ (a \vee b) \rightarrow (\mathbf{if} \ a \rightarrow (r; RHS) \sqcap b \rightarrow (q; RHS) \ \mathbf{fi}) \sqcap \neg(a \vee b) \rightarrow \mathbf{skip} \ \mathbf{fi} \\ = & \{ \text{Law 2.2.69} \} \ (; -\sqcup \text{ and } ; -\sqcap \text{ left distribution}) \\ & \mathbf{if} \ (a \vee b) \rightarrow (\mathbf{if} \ a \rightarrow r \sqcap b \rightarrow q \ \mathbf{fi}); RHS \sqcap \neg(a \vee b) \rightarrow \mathbf{skip} \ \mathbf{fi} \\ \Rightarrow & \{ \text{Law 2.2.66} \} \ (\mathbf{rec} \text{ least fixed point}) \\ & LHS \end{aligned}$$

◇

The next law establishes the connection between tail-recursion and iteration.

Law 2.2.75 (*while – rec tail recursion*)

$$\mathbf{while} \ b \rightarrow p \ \mathbf{end}; q \quad = \quad \mathbf{rec} \ X \bullet (\mathbf{if} \ b \rightarrow (p; X) \sqcap \neg b \rightarrow q \ \mathbf{fi})$$

□

Proof: ($LHS \sqsubseteq RHS$)

$$\begin{aligned} = & \{ \text{Definition 2.1.1 (While definition)} \} \text{ and } \{ \text{Law 2.2.65} \} \ (\mathbf{rec} \text{ fixed point}) \\ & LHS = \mathbf{if} \ b \rightarrow (p; \mathbf{while} \ b \rightarrow p \ \mathbf{end}) \sqcap \neg b \rightarrow \mathbf{skip} \ \mathbf{fi}; q \\ = & \{ \text{Definition 2.2.50} \} \ (; \mathbf{if} \text{ left dist}) \text{ and } \{ \text{Law 2.2.46} \} \ (; -\mathbf{skip} \text{ unit}) \\ & LHS = \mathbf{if} \ b \rightarrow (p; LHS) \sqcap \neg b \rightarrow q \ \mathbf{fi} \\ \Rightarrow & \{ \text{Law 2.2.66} \} \ (\mathbf{rec} \text{ least fixed point}) \\ & LHS \sqsubseteq RHS \end{aligned}$$

$$(RHS \sqsupseteq LHS)$$

$$\begin{aligned}
&= \{Law\ 2.2.65\} \text{ (rec fixed point)} \\
&\quad RHS = \mathbf{if}\ b \rightarrow (p; RHS) \sqcap \neg b \rightarrow q \ \mathbf{fi} \\
&\Rightarrow \{From\ Lemma\ 2.2.1\ (Strongest\ inverse\ of\ ;) \text{ we have } (RHS \sqcup q); \ q \sqsubseteq RHS\} \\
&\quad RHS \sqsupseteq \mathbf{if}\ b \rightarrow (p; (RHS \sqcup q); q) \sqcap \neg b \rightarrow q \ \mathbf{fi} \\
&= \{Law\ 2.2.50\} (; \mathbf{if}\ left\ dist) \text{ and } \{Law\ 2.2.46\} (; \mathbf{skip}\ unit) \\
&\quad RHS \sqsupseteq \mathbf{if}\ b \rightarrow (p; (RHS \sqcup q)) \sqcap \neg b \rightarrow \mathbf{skip}\ \mathbf{fi}; \ q \\
&= \{Definition\ 2.2.1\ (Approximate\ inverses)\} \\
&\quad (RHS \sqcup q) \sqsupseteq \mathbf{if}\ b \rightarrow (p; (RHS \sqcup q)) \sqcap \neg b \rightarrow \mathbf{skip}\ \mathbf{fi} \\
&\Rightarrow \{Law\ 2.2.66\} \text{ (rec least fixed point)} \text{ and } \{Definition\ 2.1.1\ (While\ definition)\} \\
&\quad (RHS \sqcup q) \sqsupseteq \mathbf{while}\ b \rightarrow p \ \mathbf{end} \\
&= \{Definition\ 2.2.1\ (Approximate\ inverses)\} \\
&\quad RHS \sqsupseteq LHS
\end{aligned}$$

◇

The following law is crucial in proving the correctness of the normal form reduction of sequential composition.

Law 2.2.76 (*while sequence*)

$$\begin{aligned}
&\mathbf{while}\ b_1 \rightarrow p \ \mathbf{end}; \\
&\mathbf{while}\ (b_1 \vee b_2) \rightarrow p \ \mathbf{end} \quad = \quad \mathbf{while}\ (b_1 \vee b_2) \rightarrow p \ \mathbf{end}
\end{aligned}$$

□

Proof: $(RHS \sqsupseteq LHS)$

$$\begin{aligned}
&RHS \\
&= \{Law\ 2.2.34\ (\mathbf{if}\ void\ guards)\} \\
&\quad RHS = \mathbf{if}\ b \rightarrow RHS \sqcap \neg b \rightarrow RHS \ \mathbf{fi} \\
&= \{Definition\ 2.1.1\ (While\ definition)\} \text{ and } \{Law\ 2.2.65\} \text{ (rec fixed point)} \\
&\quad RHS = \mathbf{if}\ b \rightarrow \mathbf{if}\ (b \vee c) \rightarrow (p; RHS) \sqcap \neg(b \vee c) \rightarrow \mathbf{skip}\ \mathbf{fi} \\
&\quad \quad \sqcap \neg b \rightarrow RHS \\
&\quad \quad \mathbf{fi} \\
&= \{Definition\ 2.2.43\ ((\mathbf{if}\ -\ void\ assertion)\}) \text{ and } \{Law\ 2.2.70\} \text{ (while elimination)} \\
&\quad RHS = \mathbf{if}\ b \rightarrow \mathbf{if}\ (b \vee c) \rightarrow (p; RHS) \sqcap \neg(b \vee c) \rightarrow RHS \ \mathbf{fi} \\
&\quad \quad \sqcap \neg b \rightarrow RHS \\
&\quad \quad \mathbf{fi}
\end{aligned}$$

$$\begin{aligned}
&= \{Law\ 2.2.39\} \text{ (if - } \wedge \text{ distrib)} \\
&\quad RHS = \mathbf{if}\ b \rightarrow (p; RHS) \square \neg b \rightarrow RHS \mathbf{fi} \\
&\Rightarrow \{Law\ 2.2.66\} \text{ (rec least fixed point)} \\
&\quad RHS \sqsubseteq \mathbf{rec}\ X \bullet (\mathbf{if}\ b \rightarrow (p; X) \square \neg b \rightarrow RHS \mathbf{fi}) \mathbf{end} \\
&= \{Law\ 2.2.75\} \text{ (while - rec tail recursion)} \\
&\quad RHS \sqsubseteq \mathbf{while}\ b \rightarrow p \mathbf{end}; \mathbf{while}\ (b \vee c) \rightarrow p \mathbf{end} \\
&= RHS \sqsubseteq LHS
\end{aligned}$$

$$(LHS \sqsubseteq RHS)$$

$$\begin{aligned}
&= \{Definition\ 2.1.1\ \text{(While definition)}\} \text{ and } \{Law\ 2.2.65\} \text{ (rec fixed point)} \\
&\quad LHS = \mathbf{if}\ b \rightarrow (p; \mathbf{while}\ b \rightarrow p \mathbf{end}) \square \neg b \rightarrow \mathbf{skip}\ \mathbf{fi}; \mathbf{while}\ (b \vee c) \rightarrow p \mathbf{end} \\
&= \{Law\ 2.2.50\ ((; \mathbf{if}\ \text{left dist}))\} \text{ and } \{Law\ 2.2.46\} \text{ (;- skip unit)} \\
&\quad LHS = \mathbf{if}\ b \rightarrow (p; LHS) \square \neg b \rightarrow RHS \mathbf{fi} \\
&= \{Law\ 2.2.65\ ((\mathbf{rec}\ \text{fixed point}))\} \\
&\quad LHS = \mathbf{if}\ b \rightarrow (p; LHS) \\
&\quad \quad \square \neg b \rightarrow (\mathbf{if}\ (b \vee c) \rightarrow (p; RHS) \square \neg(b \vee c) \rightarrow \mathbf{skip}\ \mathbf{fi}) \\
&\quad \quad \mathbf{fi} \\
&\Rightarrow \{Law\ 2.2.38\ ((\mathbf{if}\ - \vee \text{ distrib2}))\} \text{ and } LHS \sqsubseteq RHS \\
&\quad LHS \sqsubseteq (\mathbf{if}\ (b \vee c) \rightarrow (p; LHS) \square \neg(b \vee c) \rightarrow \mathbf{skip}\ \mathbf{fi}) \\
&\Rightarrow \{Definition\ 2.1.1\ \text{(While definition)}\} \text{ and } \{Law\ 2.2.66\} \text{ (rec least fixed point)} \\
&\quad LHS \sqsubseteq RHS
\end{aligned}$$

◇

2.2.2 Laws of Classes

In this section, we present some laws focused on the object-oriented features of our language. These laws were first presented in [12, 11, 13]. Here we introduce only those laws necessary for the description and justification of our compilation process. A more comprehensive set laws of classes can be found in [13, 24].

Laws which deal with object-oriented features of ROOL are mainly concerned with properties of class declarations and method calls which inherently rely on the context in which they are applied, especially when considering class hierarchies. Thus, this sort of laws must address context issues. We use the notation $cds_1 =_{cds, c} cds_2$ to denote that the sequences of class declaration cds_1 and cds_2 are equivalent in context of class declarations in cds , and the main command. Thus, both ‘ $cds\ cds_1$ ’ and ‘ $cds\ cds_2$ ’ should be valid declarations, where the juxtaposition of classes represents their union. The above equation is just an abbreviation; it is equivalent to $cds_1\ cds \bullet c = cds_1\ cds \bullet c$.

In order to make explicit which conditions must be satisfied for the application of a law, we adopt the following symbols: ‘ $(\rightarrow)$ ’ 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 in both directions.

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

Law 2.2.77 (*Class elimination*)

$$cds \text{ } cd_1 \bullet c =_{cds,c} cds \bullet c$$

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. $\square$

Attribute declarations

The next law addresses the change of visibility of an attribute, relating protected and public attributes. 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 referred to in the class in which it is declared and its subclasses.

Law 2.2.78 (*change visibility: from protected to public*)

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

provided

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

In order to clarify the notation used in these rules we assume that the attribute declaration **prot** $a : T$ along with all other attribute declarations in class C are denoted by **prot** $a : T, ads$. The declarations of methods and constructors are represented by ops . The notation $B.a$ stands for the use of the name a via expressions whose static type is exactly B . Finally, we the symbol $\leq$ denotes the subclass relation.

Law 2.2.79 (*change visibility: from private to public*)

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

provided

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

A public attribute can be made private when the above rule is applied from right to left. To do so, The proviso requires that the attribute cannot be used anywhere outside the class. From the above two laws, we can derive a law to change the visibility of an attribute from protected into private, and vice-versa.

The next law allow us to move a public attribute a from a class C to a superclass B , and vice-versa. Differently from Java, ROOL does not allow attribute redefinition or hiding. Therefore, when moving the attribute up to B , it is required that this does not cause a name conflict: no subclass of B , other than C , can declare an attribute with the same name.

Law 2.2.80 (*move attribute to superclass*)

<pre> class B extends A ads ops end class C extends B pub a : T; ads' ops' end </pre>	$=_{cds,v}$	<pre> class B extends A pub a : T; ads ops end class C extends B ads' ops' end </pre>
--	-------------	--

provided

($\rightarrow$) the attribute a is not declared by the subclasses of B in cds ;

($\leftarrow$) $D.a$, for any $D \leq B$ and $D \not\leq C$, does not appear in cds , c ops , or ops' . □

Method declarations

In this section we present laws that deal with the declaration of methods. The following law establishes that we can introduce or remove a trivial method redefinition, which consists only in a call to the method in the superclass.

Law 2.2.81 (*introduce method redefinition*)

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 $\mathbf{meth} \hat{=} \mathbf{super}.m$ as an abbreviation for $\mathbf{meth} \hat{=} (pds \bullet \mathbf{super}.m(\alpha pds))$, where αpds represents the list of parameter names declared in pds .

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 2.2.82 (*move redefined method to superclass*)

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** nor expressions in the form $((C)\mathbf{self}).a$ for any private attribute a in ads' ;
- ($\leftarrow$) m is not declared in ops' . □

The provisos about **super** are absolutely necessary when moving it from a subclass to a superclass, or vice-versa; otherwise, its semantics may be affected. The other provisos concern the validity of programs involved. The body of m can only be moved up when it does not refer to elements of the class where it is declared through uncast **self**. In other words, **self** must be used for calling methods and selecting attributes of the current object.

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 permitted 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 2.2.83 (*move original method to superclass*)

<pre> class B extends A ads ops end class C extends B ads' meth m $\hat{=}$ pc ops' end </pre>	$=_{cds,c}$	<pre> class B extends A ads meth m $\hat{=}$ pc ops end class C extends B ads' ops' end </pre>
---	-------------	---

provided

- ($\leftrightarrow$) (1) **super** and private attributes do not appear in pc ; (2) m is not declared in any superclass of B in cds ;
- ($\rightarrow$) m is not declared in ops , and can only be declared in a class D , for any $D \leq B$ and $D \not\leq C$, if it has the same parameters as pc ; (2) pc does not contain uncast occurrences of **self** nor expressions in the form $((C)\mathbf{self}).a$ 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 provisos concerning this law are similar to those of Laws 2.2.80 and 2.2.82. The first two 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 .

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

Law 2.2.84 (*method elimination*)

$$\boxed{
 \begin{array}{l}
 \text{class } C \text{ extends } D \\
 \quad ads \\
 \quad \text{meth } m \stackrel{\wedge}{=} pds \bullet c \\
 \quad ops \\
 \text{end}
 \end{array}
 } =_{cds, c} \boxed{
 \begin{array}{l}
 \text{class } C \text{ extends } D \\
 \quad \text{pub } a : T; \quad ads \\
 \quad ops \\
 \text{end}
 \end{array}
 }$$

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 . □

Method calls

The laws presented in the previous sections give properties of equivalence relation for program and class declarations. The next laws establish the same properties for commands. To do so, an additional notation is needed. We write $cds \ CDS$ to indicate the union of class declarations in cds and CDS , whereas $cds, N \triangleright c = d$ means that the equation $c = d$ holds inside a class named N , in a context defined by the set of class declarations cds . If we want to assert that the equality holds inside the main command, we replace the class name by **main**.

The following law allows us to substitute a method call **super**. m in a class C with a copy of the body of m as declared in the immediate superclass of C . The only proviso for this law precludes the body of m from containing **super** or private attributes.

Law 2.2.85 (*eliminate super*)

Consider that CDS is a set of two class declarations as follows.

```

class B extends A
  ads
  meth m  $\stackrel{\wedge}{=} pc$ 
  ops
end

class C extends B
  ads'
  ops'
end

```

Then we have that

$$cds \text{ CDS}, C \triangleright \mathbf{super}.m = pc$$

provided

($\rightarrow$) **super** and the private attributes in *ads* do not appear in *pc*. $\square$

The above law is similar to the standard copy rule for procedures [68, 87]; calls like **super.m** are not affected by dynamic binding. Besides, this law does not interfere in the arguments to which **super.m** is applied because *pc* is applied to the same arguments.

Due to dynamic binding, the use of copy rule to characterise method calls is rather complex in the sense that replacing the call by the body of the method depends on the type of the target object. Then, first we would have to inspect the type of the object to discover what is the correct class from which we can obtain the body of the method.

Fortunately, we can use Law 2.2.82 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. Another notation is introduced. Hereafter, we write $cds, N \triangleright e : C$ to indicate that in the class *N* declared in *cds*, the expression *e* has static type *C*. As before, to indicate that the typing holds inside the main command we use **main**.

Law 2.2.86 (*method call elimination*)

Consider that CDS the following class declarations

class *C* **extends** *D*

ads

meth *m* $\hat{=}$ *pc*

ops

end

is included in *cds* and $cds, A \triangleright le : C$. Then

$$cds \ A \triangleright le.m(e) = \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\}; \ pc[le/\mathbf{self}]$$

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 $\square$

The notation $pc[le/\mathbf{self}]$ 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**.

Assumptions are also used to capture the behaviour of casts. The next law handles the elimination of type cast in targets of method calls.

Law 2.2.87 (*eliminate cast of method call*)

If $cds, A \triangleright e : B, C \leq B$ and m is declared in B or in any of its superclass in cds , then $cds, A \triangleright ((C)e).m(e') = \{e \text{ is } C\}; e.m(e')$ $\square$

Expressions

The following law allow us cast any expression with its declared type.

Law 2.2.88 (*cast introduction in expressions*)

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

Similar to Law 2.2.87, the next law states that casts can be eliminated in assignments through the introduction of an assumption.

Law 2.2.89 (*eliminate cast of expressions*)

If $cds, A \triangleright le : B, e : B', C \leq B$ and $B' \leq B$, then $cds, A \triangleright le := (C)e = \{e \text{ is } C\}; le := e.$ $\square$

2.3 Summary

This chapter presented an informal description of the language ROOL and introduced some basic algebraic laws of ROOL. These laws help clarify aspects of the semantic of ROOL and serve as a basis for deriving more elaborate laws for practical application of program transformation, such as the compilation laws that will be used to compile the executable subset of ROOL into the ROOL Virtual Machine, ensuring correctness of the translation by construction.

In our approach, the compilation task is reduced to one of program refinement achieved by a series of correctness-preserving transformations which can be used to generate bytecodes for our target machine, the ROOL Virtual Machine. Although here we describe an algebraic approach to construct a provably correct compiler for ROOL, this language was devised as a mean for studying refinement of object-oriented programs in general, not only compilation. Its weakest precondition and its algebraic semantics have been studied in [19, 14]. In [25], it is used to formalise refactoring as a refinement activity.

Although pointers are ubiquitous in practice, ROOL is sufficiently similar to Java to be used in meaningful case studies and to capture some of the central difficulties that arise on the formalisation of development methods.

Chapter 3

ROOL Virtual Machine

In this chapter we present our target machine. We define a ROOL Virtual Machine (*RVM*) as our target, which is based on the Java Virtual Machine (*JVM*) [60].

In our approach to compilation, the behavior of the target machine is given by an interpreter-like program written in ROOL itself. The interpreter determines a normal form and the compilation laws are the basis for a strategy to reduce an arbitrary program to this normal form.

Compilation is identified with the reduction of an arbitrary source program to the normal form. The immediate correctness criterion is to require that the source program is refined by the corresponding program in normal form, which represents the interpreter executing the corresponding target code in the target machine.

Each executable command is translated to a sequence of bytecode instructions encoded as ROOL commands. From the normal form program, we can capture the sequence of bytecodes for the *RVM*.

3.1 Data Types

All values stored in the our *RMV* are represented as instances of the class *Data* (Figure 3.1). This class has two immediate subclasses: *PriData* and *ObjData*. The class *PriData* is related to primitive types supported by the language ROOL. This class specialises into subclasses whose instances are used to hold primitive values. For instance, an integer value is always 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 only support for two primitive types: Integer and Boolean. In order to consider others primitive types, first we have to expand the set of immediate subclasses of *PriData*. Then, the instruction set can be increased with the associated instructions that manipulate the corresponding primitive types. The *JVM* does not support boolean types, integer values are used to represent boolean values.

Figure 3.1: Representation of Data Types

3.2 RVM Components

The interpreter represents the target machine components using the following variables: *PC* (program counter), *S* (the operand stack), *M* (store for variables), *F* (the frame stack), *Cls* (data structure with the hierarchy of classes declared in the source program), and *CP* (constant pool). The execution of an instruction may change these variables, updating the machine state.

In the machine considered here, the *PC* register is used for scheduling the selection and sequencing of instructions. The initial value of *PC* is the address of the location of the first instruction to be executed. Whenever an instruction is executed, it modifies the value of *PC* to indicate the next instruction to be executed.

Because the virtual machine has no registers for storing temporary values, every operand must be pushed onto the operand stack *S* before it can be used in the execution of an instruction. In our interpreter, *S* is an instance of *Stack*. It has just one public attribute *s*, which consists of a sequence of **object**. In Figure 3.2, we present the class *Stack*. When an instance of *Stack* is created, its initialiser provides the attribute *s* to be initialised as an empty sequence, instead of having an arbitrary value.

```

class Stack extends object
  pub s : seq object;
  new  $\hat{=}$  (self.s :=  $\langle \rangle$ ) end
end

```

Figure 3.2: The Class *Stack*

The observable data space of our interpreter is a store for variables M : the concrete counterpart of the variables of the source program. It is a map from the address of the variables to their values. The symbol table Ψ maps the variables declared in the source program to addresses in the store M , in such a way that $M[\Psi_x]$ holds the value of x . In our interpreter, M is implemented as a sequence of *Data*.

	Next Frame
PC- Program Counter	Current Frame
Variables	
. . .	Previous Frame

Figure 3.3: A frame in the ROOL Virtual Machine

The frame stack F has type class *Stack*. It is devised to hold the state of execution of a method. When a method is invoked, a new frame is placed at the top of the frame stack; when the method completes, the frame is discarded (Figure 3.3).

Figure 3.4: The Constant Pool

The component *Cls* is a global variable holding the essential information about class declarations in the source program. This variable has class type *CdsHierarchy*. Basically, it is represented by a sequence of objects of the class *ClassInfoObj*. Each of them has the following attributes: *cl*, which codifies the class name, and *subcls*, the sequence of immediate subclasses. This class is especially designed to model the **object** class. The class *ClassInfo* its subclass and declares the attribute *super*, which codifies the superclass. The *ClassInfo* is used to model the other classes different from **object**.

In Figure 3.5, we present the declarations of *CdsHierarchy* and *ClassInfo*. The method *Instanceof* declared in *CdsHierarchy* is used to check if a given object has a certain type. It is defined recursively; it tests one for class and if the type does not match, then the method is called for each one of the subclasses, therefore, it traverses the class hierarchy stored in *Cls*.

```

class CdsHierarchy extends object
  pri clsH : Seq ClassInfo;
  meth Instanceof  $\hat{=}$  (val D : objData; j : Integer; res r : boolean •
    var s : Seq Integer •
    if D.cl = self.clsH[j].cl  $\rightarrow$  r := true
     $\square \neg(D.cl = \mathbf{self}.clsH[j].cl) \rightarrow$ 
      s := self.clsH[j].subcls; r := false;
      while(length(s) > 0) and(r = false)  $\rightarrow$ 
        self.Instanceof(D, head(s), r); s := tail(s)
      end
    fi
  end)
end

class ClassInfoObj extends object
  pub cl : integer;
  pub subcls : Seq integer;
endclass ClassInfo extends ClassInfoObj
  pub super : integer;
end

```

Figure 3.5: The Classes *CdsHierarchy* and *ClassInfo*

The constant pool *CP* is a heterogeneous list of constants; these constants are entries obtained through an index given by the symbol table Φ . Entries are objects of three different types, as shown in Figure 3.4. An entry containing an integer value is an object of type *DataInt*; boolean values are encapsulated in instances of *DataBool*; an entry containing an object of type *objData* holds what is the *RVM*'s representation of a fresh object of a given class. The symbol table Φ maps the elements declared in the source program to addresses in the constant pool *CP*, such that $CP[\Phi_C]$ holds the entry of a corresponding fresh object of class *C* in the constant pool.

3.3 The Normal Form

The *RVM* is characterised by a normal form, which is an interpreter-like program modelling a cyclic mechanism which executes one instruction at a time (Figure 3.6).

As a valid program in ROOL, the normal form consists of a sequence of class declarations (CDS_{RVM}) followed by a main command named *I*. The class declarations define the classes that

yield the description of the *RVM* components. The main command describes the behavior of our target machine executing a compiled program: an iterated execution of a sequence of bytecodes represented by the set of guarded commands *GCS*. Every cycle fetches the next bytecode instruction to be executed and then invokes an associated sequence of ROOL commands to simulate its effect on the internal data structures of the virtual machine.

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

```

cdsRVM • var PC : Int; S, F : Stack      •
           S := new Stack; F := new Stack;
           PC := s;
           while PC ≥ s ∧ PC < f → if GCS fi end
           end

```

Figure 3.6: The ROOL Interpreter (*cds_{RVM}* • *I*)

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

In order to illustrate an instance of our Interpreter, we give the following example.

```

cdsRVM • var PC : Int; S, F : Stack      •
           S := new Stack; F := new Stack;
           PC := s;
           while PC ≥ s ∧ PC < s + 7 →
             if rvm.pc = s → Load( $\Psi_x$ )
               [] rvm.pc = s + 2 → Ldc( $\Phi_3$ )
               [] rvm.pc = s + 4 → Mul
               [] rvm.pc = s + 5 → Store( $\Psi_y$ )
             fi
           end
           end

```

In the above example, we have a ROOL program in our normal form. In the main command the target code is stored as a set of guarded commands (*GCS*). The first instruction to be executed is the one which has the guard $PC = s$; when it is executed, the variable whose address is indicated by Ψ_x , in the memory storage M , is pushed onto the operand stack, and then, the PC is incremented by 2. The next instruction to be executed is $Ldc(3)$, which pushes onto the operand stack the constant whose index to the constant pool is given by Φ_3 . At this point, the PC was incremented by two. The next instruction to be selected is the one whose guard is $PC = s + 4$; it pops two integers from the operand stack, multiplies them and pushes the result back onto the stack; this time, the PC is incremented by one because this instruction has no argument, thus it occupies only one bytecode. Finally, the last instruction to be executed is $Store(\Psi_y)$; it pops an integer from the operand stack and stores it in the memory storage M , in the position indicated by the index Ψ_y . So, the overall effect of the execution of this program is just the multiplication an integer variable by a constant, followed by the assignment of the resulting value to another integer variable.

3.4 Instruction Set

In this section, we show how the instructions of our virtual machine are defined. The computation in the virtual machine is centralised 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.

Most of the data structure used to implement our interpreter employs sequences of objects. We assume the following operators that deal with sequences:

Definition 3.4.1 (Sequence operators)

$\langle \rangle$	<i>the empty sequence</i>
$\langle x \rangle$	<i>the singleton sequence with element x</i>
$X \frown Y$	<i>the concatenation of sequence X with sequence Y</i>
$head(X)$	<i>the leftmost element of sequence X</i>
$last(X)$	<i>the rightmost element of sequence X</i>
$front(X)$	<i>the sequence without the last element of X</i>
$tail(X)$	<i>the sequence without the head element of X</i>
$\#X$	<i>the number of elements of X</i>

□

Arbitrary results are obtained when *head*, *last*, *front* and *tail* are applied to empty sequences. In the following we present some laws of sequences.

Law 3.4.1 (*laws of sequences*)

- (1) $head(\langle x \rangle \frown X) = x = last(X \frown \langle x \rangle)$
- (2) $front(X \frown \langle x \rangle) = X = tail(\langle x \rangle \frown X)$
- (3) If X is non-empty then $(head(X) \frown X) = X = (front(X) \frown last(X))$
- (4) $X := \langle x \rangle \cup X; X := tail(X) = \mathbf{skip}$

□

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 3.4.2 (Instruction set - Miscellaneous)

$$\begin{aligned}
 nop & \stackrel{def}{=} PC := PC + 1 \\
 cmpeq & \stackrel{def}{=} S.s := \langle (\mathbf{new} \text{DataBool}; \\
 & \quad Info : (head(tail(S.s)) = head(S.s))) \rangle \frown tail(tail(S.s)); \\
 & \quad PC := PC + 1
 \end{aligned}$$

□

The instruction with opcode *nop* (no operation) has no effect, except for the program counter (PC) increment.

To execute *cmpeq* two values are popped from the operand stack and one is compared against the other. If they are equal then *true* is pushed onto the operand stack, otherwise *false* is pushed.

Definition 3.4.3 (Pushing and popping values onto/from the operand stack)

$$\begin{aligned}
 ldc(j) & \stackrel{def}{=} S.s := \langle CP[j] \rangle \frown S.s; \quad PC := PC + 2 \\
 load(n) & \stackrel{def}{=} S.s := \langle M[n] \rangle \frown S.s; \quad PC := PC + 2 \\
 store(n) & \stackrel{def}{=} M[n] := head(S.s); \quad S.s := tail(S.s); \quad PC := PC + 2 \\
 aconstnull & \stackrel{def}{=} S.s := \langle \mathbf{null} \rangle \frown S.s; \quad PC := PC + 1
 \end{aligned}$$

□

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 actually 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 a index to a local variable.

Some opcodes by themselves indicate a type and constant value to push. The instruction with opcode *aconstnull* pushes a **null** value onto the operand stack.

Definition 3.4.4 (Unary and Binary Operators)

$$\begin{aligned}
uop &\stackrel{def}{=} S.s := \langle (\mathbf{new} \text{ DataInt}; \text{ Info} : \mathbf{uop} (\text{DataInt head}(S.s)).\text{Info}) \rangle \frown \text{tail}(S.s); \\
&\quad PC := PC + 1 \\
bop &\stackrel{def}{=} S.s := \langle (\mathbf{new} \text{ DataInt}; \text{ Info} : (\text{DataInt head}(S.s)).\text{Info} \mathbf{bop} \\
&\quad (\text{DataInt head}(\text{tail}(S.s))).\text{Info}) \rangle \frown \text{tail}(\text{tail}(S.s)); \\
&\quad PC := PC + 1
\end{aligned}$$

□

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

Definition 3.4.5 (Branch Instructions)

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

□

A *goto* instruction always branches. It causes execution to jump to next instruction determined by the parameter *k*. The *cgoto* is a conditional *goto*. If the value stored at the top of the operand

stack holds, execution passes to the next instruction, otherwise execution continues at the specified instruction.

Observe that the *PC* increment depends on the number of operands demanded by the instruction. The more operands are needed, more bytecodes are necessary to codify the instruction. For instance, when the instruction needs no operand, as in *bop*, the *PC* is incremented by 1 because 1 bytecode is enough to codify the opcode, whereas in the case *load n*, an additional bytecode is necessary to represent the operand that indicates the location *n* in the storage *M*, thus the *PC* is rather incremented by 2.

Definition 3.4.6 (Object creation and manipulation)

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

□

The instruction *new* creates a new class instance. In *RVM*, instead of traversing the representation of the source program class hierarchy, determining and recording in 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*'s 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 the object which belongs to the concrete data space of the target machine. Once this object is obtained, it is pushed onto the operand stack.

When the instruction *putfield* is executed, two values are popped from the operand stack. The first one is the object whose attribute is to be set. The second one is the value to be assigned to this attribute. The instruction *putfield(j)* selects and modifies the value of the attribute indicated by the index *j*. Finally, a copy of the modified object is pushed onto the operand stack.

The execution of *getfield* pops from the operand stack only one value: the object whose attribute value is to be obtained. The argument *j* is an index into the constant pool that identifies the corresponding attribute. The instruction *getfield(j)* gets the value of the attribute denoted by *j* in the *RVM* representation of the object, and pushes it onto the operand stack.

Definition 3.4.7 (Type testing)

The instruction *instanceof* tests if an object belongs to a type. The argument *j* is an integer value given by the function Φ that identifies a class. The global variable *Cls* holds an image of the class

hierarchy declared in the source program. From this data structure we can determine if the object belongs to a class C or any subclass of C . Below we define the method *Instanceof*.

```

instanceof( $j$ )  $\stackrel{def}{=}$  var  $obj : objData, r : DataBool$  •
     $obj := head(S.s); S.s := tail(S.s);$ 
     $Cls.Instanceof(Obj, j, r);$ 
     $S.s := \langle r \rangle \frown S.s; PC := PC + 2$ 
end

```

□

Definition 3.4.8 (Method invocation)

```

save( $o$ )  $\stackrel{def}{=}$   $F.s := \langle o \rangle \frown F.s$ 
restore( $o$ )  $\stackrel{def}{=}$   $o := head(F.s); F.s := tail(F.s)$ 
invoke( $m$ )  $\stackrel{def}{=}$   $PC := PC + 2; F.s := \langle (\mathbf{new} \text{ } DataInt; Info : PC) \rangle \frown F.s; PC := m$ 
return  $\stackrel{def}{=}$   $PC := (DataInt \text{ } head(F.s)).Info; F.s := tail(F.s)$ 

```

□

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 . Then, 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 off the frame stack F . The execution flow returns to the next instruction after the invocation that originated the method call.

Chapter 4

Compilation Process

In our approach, the design of a compiler is a constructive proof that an arbitrary program in ROOL is refined by the corresponding program in the normal form. In order to carry out the compilation task, we effectively use provably correct reduction theorems as rewrite rules.

The compilation process is divided into five phases: Class Pre-compilation, Redirection of Method Calls, Simplification of Expressions and Some Source Transformations, 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 a main compilation theorem links the intermediate steps and states the outcome of the entire process.

The reduction theorems are proved correct from the basic algebraic laws of ROOL and the compilation rules introduced in this chapter. We make a distinction between laws and rules. The laws presented in Chapter 2 assert general properties of the language constructors, whereas rules serve the special purpose of compilation.

Since the reduction theorems for all phases are proved correct from the basic laws of ROOL, they corroborate the correctness of the compilation process. The correctness of the compiler follows from the correctness of each compilation rule.

In each of the sections 4.1 to 4.5 we introduce the reduction theorem for the relevant phase and its associated compilation rules. Finally, in the last section we present the proof of the main theorem. The proof of each compilation rule is presented in Appendix A.

4.1 Class Pre-compilation

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

Theorem 4.1.1 (Class Pre-compilation) *Let $cds \bullet c$ be a program in ROOL containing only executable constructs, correct with respect to its syntax and its static typing of expressions and statements. Then, there is a program $cds' \bullet c'$ such that*

$$cgs \bullet c \sqsubseteq cds' \bullet c'$$

where (1) the main command c' differs from c only by the introduction of cast in all its expressions; (2) the class declarations in cgs' are obtained from cgs , changing the visibility of all attributes to public; (3) further, method redefinitions in cgs are eliminated in cgs' ; and (4) all targets of calls and attributes accesses are cast with their static types, according to Law 2.2.88. $\square$

This compilation phase comprises three steps. The first one changes the visibility of the attributes in cgs , and the second one 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 cgs , laws of classes are applied to eliminate redefinitions; as an effect of this strategy, dynamic binding of methods and references to **super** are eliminated. Before we detail each of these steps, we briefly present a reduction strategy for ROOL programs that is useful in the context of our work.

4.1.1 A strategy for normal form reduction

In [13], a normal form for ROOL programs and a corresponding reduction strategy are presented. It follows the same approach used for other programming paradigms [37]; the only intention is to show that a set of laws is comprehensive in the precise sense that it can be used to obtain an equivalent program written with a small subset of constructors.

In order to show that a comprehensive set of laws has been proposed for ROOL, a strategy to reduce an arbitrary executable program to a normal form has been proposed in [13]. The normal form is defined by the following characteristics:

- it preserves each class declaration in cgs , but with an empty body;
- the class **object** is the only one with explicitly declared elements; namely it includes only public attribute declarations, each with either primitive type or **object**. It takes the form of a recursive record;
- since no methods are allowed, the main command c is similar to an imperative program, in spite of the fact that their normal form preserves object-oriented features such as subtype hierarchy, object creation, and type test; finally, c does not allow type casts.

This reduction strategy shows that the laws presented in Section 2.2.2 are sufficient to move upwards the method declarations of m in the hierarchy of classes towards the **object** class. Doing so, complex issues such as dynamic binding are handled in a separate way, using, for instance, Law 2.2.83.

Adopted as a measure of completeness for a set of proposed algebraic laws, their strategy might resemble a compilation process, but it is not. For instance, the use of *copy rule* to in-line the body of methods is not acceptable for the design of compilers, because this may substantially increase the size of the target code in case the program contains a large number of calls. Usually, most compilers use control mechanisms to allow a single compilation of a method body; we follow this view in our work.

Even so, we make use of the set of laws in [13] to accomplish the desired outcome for this compilation phase. We do not exactly follow their strategy, moving up all attributes and method declarations towards the **object** class, but we rely on part of their strategy. Therefore, though similar, our approach has the following difference: using the laws of classes from left to right, we move up method and attribute declarations associated with redefined methods, until we reach a point in the class hierarchy in which the method redefinition is eliminated. Typically, this happens before we reach the **object** class. Furthermore, we do not eliminate casts nor the method declarations. In the next phase the method declarations are moved to a single class (*L*).

4.1.2 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; otherwise, an error can arise when we move the method declaration upwards in the class hierarchy. Therefore, we need to change the declarations of the attributes to make them all public. Even though this is not a good idea from a software engineering point of view, this does not change the behaviour of a complete program, especially considering that our aim here is compilation rather than, for instance, refactoring [78, 40, 24]. To perform these transformations, we use Laws 2.2.78, and 2.2.79. These laws were first presented in [14] and later in [13].

In order to illustrate the impact of our compilation strategy on the source program, we reduce an executable source program to our normal form, showing step by step which transformations are imposed by each compilation phase. From here on, the program presented in Figure 1.3, in Chapter 1, is regarded as an example to illustrate the transformations performed by our compilation strategy.

Applying Laws 2.2.78, and 2.2.79 to our example, we change the declaration of *dir* and *len* in *Step*, and *previous* in *Path* to make them public. Changing the visibility of attributes is necessary to guarantee that the following transformations are allowed, or rather than the application of other laws presented in Section 2.2.2 is possible.

4.1.3 Introducing trivial casts

Applying Law 2.2.88, we introduce type casts to produce a uniform program text in which all targets of calls and attributes accesses are cast with their static types. The purpose of this step is to explicitly annotate the program text with the declared type of each target. Casting is necessary

when we use expressions in contexts where an object value of a given type is expected. Introducing casts in the original program, however, has no effect since they do not change the behaviour of the program. During the elimination of method redefinitions, casts play a fundamental role because, in the presence **self**, when we move methods upwards in the class hierarchy, the behaviour of the program can be modified.

In Figure 4.1, we present the intermediate program obtained from the source program, after the application of the transformations explained so far.

```

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

```

Figure 4.1: Program obtained after introducing trivial casts

Note that when we cast the occurrence of **super** in *getLength*, the context yields *Step* as the type of the target object of this call. Like Java, ROOL does not allow casts to appear in targets of assignment and result parameters.

4.1.4 Elimination of method redefinition in our example

As an example, consider the method *getLength*, which is first declared in the class *Step* and redeclared in the class *Path*. Before moving this method up in the class hierarchy, we have to eliminate

references to **super**, otherwise we would end up referring to a nonexistent method. The elimination of **super** relies on Law 2.2.81, which is basically a copy rule. Applying this rule, we obtain the class *Path* described in Figure 4.2.

```

class Path extends Step
  pub previous : Path;
  meth addStep  $\hat{=}$  (val d, l : Int •
    (Path self).previous := (Path self); (Path self).setDirection(d);
    (Path self).setLength(l) )
  meth getLength  $\hat{=}$  (res l : Int •
    var aux : Int •
      if ((Path self).previous = null) → aux := 0
        □ (Path self).previous <> null → (Path self).previous.getLength(aux)
      fi;
      (res l : Int; • l := (Step self).len )(l); l := l + aux;
    end )
end

```

Figure 4.2: Class *Path* without references to **super**

At this point, we can apply Law 2.2.80 to move the public attribute *previous* from class *Path* to its superclass *Step*, and then, apply Law 2.2.83, to transform the classes *Path* and *Step* as shown in Figure 4.3.


```

class Step
  pub dir, len : Int;
  pub previous : Path;
  meth setDirection  $\hat{=}$  (val d : Int; • (Step self).dir := d )
  meth setLength  $\hat{=}$  (val l : Int; • (Step self).len := l )
  meth getLength  $\hat{=}$  (res l : Int; •
    if (self is Path) → l := (Step self).len
      var aux : Int •
        if ((Path self).previous <> null) →
          (Path self).previous.getLength(aux)
          □ ((Path self).previous = null) → aux := 0
        fi; (res l : Int; • l := (Step self).len )(l); l := l + aux;
      end
    □ ¬(self is Path) → l := (Step self).len
  fi)
end
class Path extends Step
  meth addStep  $\hat{=}$  (val d, l : Int •
    (Path self).previous := (Path self); (Path self).setDirection(d);
    (Path self).setLength(l) )
end

```

Figure 4.3: Method *getLength* as a method without redefinitions

Now, the above class declarations have a single method declaration for *getLength*, which can be used to define the declaration of its associated method in class *L* to be introduced later.

4.1.5 Proof of Theorem 4.1.1

The program $cds' \bullet c$ in Theorem 4.1.1 can be obtained, first of all, by the exhaustive application of Laws 2.2.78 and 2.2.79 to change the visibility of all attributes. Then, Law 2.2.88 is applied to introduce trivial casts. After that, applying Law 2.2.81, we establish the necessary conditions to use Law 2.2.85 to eliminate the clause **super**. Finally, the application of Law 2.2.83 allows us to merge a method declaration and its redefinition into a single declaration in the superclass; the exhaustive application of this law eliminates method redefinition in our program.

Proof:

$$\begin{aligned}
& cds \bullet c \\
= & \{Law\ 2.2.78\} \text{ (change visibility: from protected to public)} \\
& \{Law\ 2.2.79\} \text{ (change visibility: from private to public)} \\
& \{Law\ 2.2.88\} \text{ (cast introduction in expressions)} \\
& \{Law\ 2.2.81\} \text{ (introduce method redefinition)} \\
& \{Law\ 2.2.85\} \text{ (eliminate **super**)} \\
& \{Law\ 2.2.80\} \text{ (move attribute to superclass)} \\
& \{Law\ 2.2.83\} \text{ (move original method to superclass)} \\
& cds' \bullet c'
\end{aligned}$$

◇

These transformations yield the expected outcome for this phase, since they are sufficient to convert the source program $cds \bullet c$ into $cds' \bullet c'$.

4.2 Redirecting method calls

In this phase, first we introduce a particular class L , which plays a fundamental role in our strategy; it includes elements that later will allow the elimination of the source method declarations. Moreover, the introduction of L allows arguments of method calls to be passed using the operand stack S , adapting the invocation of methods to work in the way it is defined in the RVM . As explained in Chapter 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. 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 set of class declarations cds_{RVM} , which is not referenced by the source program in the beginning, but throughout the compilation process, some commands that will be introduced in L and in the main command c will reference methods, attributes, and types defined in cds_{RVM} . For this reason, when we start the process of restructuring the code, introducing class L , the need to introduce cds_{RVM} arises.

The outcome of this phase of compilation is summarised by Theorem 4.2.1: the compilation rules applied in this phase are sufficient to end up with a program where all calls are to methods in L .

Theorem 4.2.1 (Redirection of method calls) *Let $cds \bullet c$ be a program where all attributes in cds are public, every possible cast was introduced, and all methods declared in cds have no redefinitions, then there is a program $cds_{RVM}, cds', L' \bullet c'$, such that $cds \bullet c \sqsubseteq cds_{RVM}, L', cds' \bullet c'$, where*

cds' contains only the attribute declarations of *cds*, and the calls in *c* to methods in *cds* are replaced in *c'* with calls to the corresponding methods in *L*. $\square$

In order to transform each method call that appears in the program into one to an associated method in *L*, we first copy all method bodies declared in *cds* to their associated methods declared in *L*. The previous compilation phase establishes the conditions that allow the definition of *L* without introducing errors. 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 to establish that their arguments have to be pushed onto and popped from the operand stack *S*. Therefore, when a redirected method is invoked, along with the arguments that appear in the original method call, a copy of its target object is also pushed onto the operand stack. When the execution of a method is completed, the results of its execution are reflected on the operand stack, including a copy of its target object, possibly modified.

4.2.1 The class *L*

The introduction of class *L* is totally justified by Law 2.2.77. In principle, any class that is not referenced in the program can be introduced. Later, when redirecting the method calls, we need to prove the compilation rule that allows this transformation. Then, it will be clear why the class *L* is suitable.

For each method *m* declared in *cds*, we introduce the corresponding method *lm* in *L* in such a way that *lm* simulates the behaviour of *m*.

When we copy and arrange all method bodies declared in *cds*, inserting in the class *L* the method declarations from *cds*, our purpose is to set the context to modify each method call present in the source program to an uniform pattern, using an operand stack. After the Redirection of Method Calls, this stack will be used by the body of an invoked method to obtain its arguments and its target object, and to return the results of its execution. The intention behind the introduction of an operand stack associated with each method call is to mimic the way the invoke instructions are defined in the *JVM* (and in *RVM*).

Once a method *lm* is invoked, we need to guarantee that the method body associated with the original method call is executed. Every method in *L* whose parameters are passed by value shares a common pattern, which is described in Figure 4.4, whereas those methods whose parameters are passed by result share the pattern depicted in Figure 4.5.

In these 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 return arguments. Since our language does not support value-result parameters (nor sharing), two stacks are needed: S_{in} to receive the arguments, and S_{out} to return the results derived from the execution of *pc* when applied to its arguments. In Figure 4.4, the argument *v* is passed by value to the

```

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

```

Figure 4.4: The pattern of a method with value parameters in the class L

```

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

```

Figure 4.5: The pattern of a method with result parameters in the class L

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 case of Figure 4.5, the argument r is passed by result to the parametrised command pc , imitating the same behavior of a method call whose target is o and result argument is r .

For methods that are not redefined, the parametrised command in lm is exactly the same we find in the definition of m , except for the replacement of **self** by the variable o . This substitution is syntactically allowed because the occurrences of **self** are cast, as already explained.

Although it is not necessary to justify the definition of L incrementally, for didactic reasons, we explain below how the construction of L can be done introducing its methods one at a time. As an example, consider the method *getLength*. In our first compilation phase, its attributes became public, all possible casts were introduced, and its redefinition was eliminated. Following the pattern previously presented in Figure 4.4, we introduce a new method named as *lgetLength* in L , having its body described in Figure 4.6.

In a similar way, for each method declared in cds , we introduce the corresponding method in L , until we get a one-to-one correspondence with all methods in cds to methods in L .

Observe that in the method declaration in Figure 4.6, there is still a call to the method *getLength* declared in cds . In the next section, we show how we redirect method calls of our example in order to make all method declarations in cds useless.

```

meth lgetLength   $\hat{=}$   (val Sin : Stack; res Sout : Stack •
    var o : Object; l : int • Pop(Sin, o);
    (res l : Int; •
    if (o is Path) →
        var aux : Int •
        if ((Path o).previous = null) → aux := 0
            □ ((Path o).previous <> null) →
                (Path o).previous.getLength(aux)
        fi;
        (res l : Int; • l := (Step o).len )( l ); l := l + aux;
    end
    □ ¬(o is Path) → l := (Step o).len
    fi ) (l); Pushi(Sin, l); Push(Sin, o); Sout := Sin
end)

```

Figure 4.6: Method *lgetLength* declared in *L*

4.2.2 Rules

For each parameter passing mechanism, we have an associated compilation rule. The difference between them resides in the type of the parametrisation. The next two rules establish the transformations related to the Redirection of Method Calls to methods in *L*. Multiple parametrisation of different kinds are handled by combining the effects of these rules.

In order to clarify the following rules, we introduce abbreviations of sequence of commands over our operand stack, which is an instance of class *Stack* (Section 3.2), to manipulate non-primitive variables.

Definition 4.2.1 (Abbreviations to manipulate non-primitive variables through the operand stack)

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

□

In the abbreviations *Bop*(*S*) and *Uop*(*S*), **bop** and **uop** stand for arbitrary binary and unary operators, respectively. There are versions of these abbreviations for each primitive type. In the following, we present those for the integer and boolean type.

The abbreviation $Putfd(S, C.t)$ removes two elements from the top of the operand stack S . The first one is the object of class C whose attribute t is to be set. The second one is the value to be assigned to this attribute. A copy of the modified object is pushed onto the operand stack.

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

Definition 4.2.2 (Abbreviations — integer variables through the operand stack)

$$\begin{aligned}
Push_i(S, x) &\stackrel{def}{=} S.s := \langle (\mathbf{new} \text{ DataInt}; \text{ Info} : x) \rangle \frown S.s \\
Pop_i(S, x) &\stackrel{def}{=} x := (\text{DataInt head}(S.s)).\text{Info}; \quad S.s := \text{tail}(S.s) \\
Bop_i(S) &\stackrel{def}{=} S.s := \langle (\text{DataInt head}(\text{tail}(S.s))).\text{Info} \mathbf{bop} \\
&\quad (\text{DataInt head}(S.s)).\text{Info} \rangle \frown \text{tail}(\text{tail}(S.s)) \\
Uop_i(S) &\stackrel{def}{=} S.s := \langle \mathbf{uop} (\text{DataInt head}(S.s)).\text{Info} \rangle \frown \text{tail}(S.s)
\end{aligned}$$

□

Definition 4.2.3 (Abbreviations — boolean variables through the operand stack)

$$\begin{aligned}
Push_b(S, x) &\stackrel{def}{=} S.s := \langle (\mathbf{new} \text{ DataBool}; \text{ Info} : x) \rangle \frown S.s \\
Pop_b(S, x) &\stackrel{def}{=} x := (\text{DataBool head}(S.s)).\text{Info}; \quad S.s := \text{tail}(S.s) \\
Bop_b(S) &\stackrel{def}{=} S.s := \langle (\text{DataBool head}(\text{tail}(S.s))).\text{Info} \mathbf{bop} \\
&\quad (\text{DataBool head}(S.s)).\text{Info} \rangle \frown \text{tail}(\text{tail}(S.s)) \\
Uop_b(S) &\stackrel{def}{=} S.s := \langle \mathbf{uop} (\text{DataBool head}(S.s)).\text{Info} \rangle \frown \text{tail}(S.s) \\
Is(S, C) &\stackrel{def}{=} S.s := \langle (\text{DataBool}; \text{ info} : (\text{head}(S.s) \mathbf{is} C)) \rangle \frown \text{Tail}(S.s)
\end{aligned}$$

□

The abbreviation $Is(S, C)$ removes an element from the top of the operand stack S and tests if it has the type C . The result of this test is encapsulated in an instance of $DataBool$ and pushed onto S . When manipulating primitive values through the operand stack we encapsulate them in instances of classes declared in CDS_{RVM} . For instance, integer values are encapsulated in instances of $DataInt$.

For conciseness, in the rules presented from here on, we consider mainly the non-primitive types, since versions for primitive types are similar. The following rule deals with method calls with result parameter.

Rule 4.2.1 (Redirecting a call with a result parameter)

Consider that the following class declaration

```

class  $C$  extends  $D$ 
   $ads$ 
  meth  $m \stackrel{\wedge}{=} pc$ 
   $ops$ 
end

```

is included in cds and $cds, A \triangleright le : C$. Then, there is the following associated method declaration in the class L

```

 $cds_{RVM} cds, L \triangleright$  meth  $lm \stackrel{\wedge}{=} (\text{val } S_{in} : Stack; \text{res } S_{out} : Stack \bullet$ 
  var  $o : Object; r : T \bullet$ 
   $Pop(S_{in}, o); pc[o/\text{self}](r);$ 
   $Push(S_{in}, r); Push(S_{in}, o); S_{out} := S_{in}$ 
end

```

such that

```

 $cds_{RVM} cds L, N \triangleright$ 
   $((C)le).m(x) \sqsubseteq \{le \neq \text{null} \wedge le \neq \text{error}\}$ 
  var  $S : Stack; V : L \bullet$ 
   $S := \text{new } Stack; V := \text{new } L;$ 
   $Push(S, le); V.lm(S, S);$ 
   $Pop(S, le); Pop(S, x);$ 
end

```

provided

($\rightarrow$) (1) m is declared in cds and has one 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 $\square$

For the redirection of a method call with a result parameter, 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 , then, le is pushed onto S . After calling the lm method, the value of the result argument is popped from S and assigned to x ; the resulting target 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. For simplicity, from now on we omit the context in the rules.

Rule 4.2.2 (Redirecting call with value parameter)

Consider that the following class declaration

```

class  $C$  extends  $D$ 
   $ads$ 
  meth  $m \stackrel{\wedge}{=} pc$ 
   $ops$ 
end

```

is included in cds and $cds, A \triangleright le : C$. Then, there is the following associated method declaration in the class L

```

 $cds_{RVM} cds, L \triangleright$  meth  $lm \stackrel{\wedge}{=} (\text{val } S_{in} : Stack; \text{res } S_{out} : Stack \bullet$ 
  var  $o : Object; v : T \bullet$ 
   $Pop(S_{in}, o); Pop(S_{in}, v); pc[o/self](v);$ 
   $Push(S_{in}, o); S_{out} := S_{in}$ 
end)

```

such that

```

 $cds_{RVM} cds L, N \triangleright$ 
   $((C)le).m(x) \sqsubseteq \{le \neq \text{null} \wedge le \neq \text{error}\}$ 
  var  $S : Stack; V : L \bullet$ 
   $S := \text{new } Stack; V := \text{new } L;$ 
   $Push(S, x); Push(S, le); V.lm(S, S);$ 
   $V.lm(S, S); Pop(S, le)$ 
end

```

provided

($\rightarrow$) (1) m is declared in cds and has one 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 . $\square$

The above rule is similar to the previous one, except for the role of the argument x . In this rule, x represents a value argument, hence it is pushed onto the operand stack before lm is called.

Combining the effect of the above two rules, we can deal with method calls with multiple parametrisation of (possibly) different kinds. All value arguments must be pushed onto the stack before the method call, whereas the result arguments must be popped from the stack after the method call completes. The target object le is the only exception; it always behaves as a value and as a result argument.

As a desired effect of applying the Rules 4.2.1 and 4.2.2 to redirect method calls, the method declarations in cds become useless. Therefore, based on Law 2.2.84, they can be eliminated.

4.2.3 Moving attributes down

From the theoretical point of view, moving down the attributes to the classes they were originally declared is totally irrelevant because it does not influence the proof of our theorems. Nevertheless, if we do not move the attribute declarations to the classes they were initially declared, we would lose the data structure defined by the programmer. As an immediate consequence, a possible increment in the number of attributes in the objects would occur, incurring in a waste of memory space allocated to hold instances of objects in our virtual machine. Certainly, it is not an acceptable practice for a realistic compiler. In [35] we describe a compilation strategy that keeps the attributes in the classes they were originally declared, but unfortunately, it has proven difficult to formalise.

The decision of moving attributes up has simplified our design and reasoning. On the other hand, unlike [35] we need to move them back to their original classes. The strategy adopted here is to record the path (sequence of class names) from the original class of each attribute to the **object** class.

The original class of an attribute a is given by $Class(a)$, where $Class$ is a function defined by structural induction in cds . We also use a relation H to represent the class hierarchy, such that $A \leq_{cds} B$ iff $B H A$ (B is related to A via H).

To guide the application of Law 2.2.80 we define the algorithm presented in Figure 4.7.

- For each class C in cds (starting from **object**)
 - For each attribute a in C

move a to the immediate subclass of C , say X , provided that

$$(X = Class(a)) \vee (X H^+ Class(a))$$

Figure 4.7: An algorithm to guide the application of Law 2.2.80

The algorithm considers every class in cds , starting from the object class, and every attribute a in C , then, applying Law 2.2.80 from right to left, it moves a to the immediate subclass of C , provided that this subclass is the original class of a or there is a path from this subclass to the defined class of a .

4.2.4 Redirecting calls in our example

In our example, calls to the method *getLength* have the parameter passed by result. When the call $p.getLength(out)$ in the main command of our program example is redirected using the Rule 4.2.1; the following variable block is obtained.

```

var  $p$  :  $Path$  •
   $p := \text{new } Path$ ;
  var  $S$  :  $Stack$ ;  $V$  :  $L$  •  $S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push_i(S, north)$ ;  $Push_i(S, l_1)$ ;  $Push(S, p)$ ;  $V.laadStep(S, S)$ ;  $Pop(S, p)$ 
  end
  var  $S$  :  $Stack$ ;  $V$  :  $L$  •  $S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push_i(S, east)$ ;  $Push_i(S, l_2)$ ;  $Push(S, p)$ ;  $V.laadStep(S, S)$ ;  $Pop(S, p)$ 
  end
  var  $S$  :  $Stack$ ;  $V$  :  $L$  •  $S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push_i(S, south)$ ;  $Push_i(S, l_3)$ ;  $Push(S, p)$ ;  $V.laadStep(S, S)$ ;  $Pop(S, p)$ 
  end
  var  $S$  :  $Stack$ ;  $V$  :  $L$  •  $S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push_i(S, west)$ ;  $Push_i(S, l_4)$ ;  $Push(S, p)$ ;  $V.laadStep(S, S)$ ;  $Pop(S, p)$ 
  end
  var  $S$  :  $Stack$ ;  $V$  :  $L$  •  $S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push(S, p)$ ;  $V.lgetLength(S, S)$ ;  $Pop(S, p)$ ;  $Pop_i(S, out)$ 
  end
end

```

Figure 4.8: Main command obtained after redirecting method calls

```

var  $S$  :  $Stack$ ;  $V$  :  $L$  •  $S := \text{new } Stack$ ;  $V := \text{new } L$ ;
   $Push(S, p)$ ;  $V.lgetLength(S, S)$ ;  $Pop(S, p)$ ;  $Pop_i(S, out)$ 
end

```

As a result of applying Rules 4.2.1 and 4.2.2, for each method call in the main command and in the class L , a variable block is introduced; they are further manipulated 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 manipulation 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. In Figure 4.8, we show the resulting main command of our example obtained when the phase of Redirection of Method Calls is finished.

The method *getLength* uses recursion to find the length of a robot's path. The redirection of a recursive method call does not incur in any overhead. Any recursion is automatically embedded in recursive calls to the associated method lm in L . For instance, consider the recursive call $(Path\ o.previous).getLenth(aux)$ appearing in Figure 4.6, when we rewrite this method call, we obtain the following.

```

var  $S : Stack$ ;  $V : L \bullet S := \mathbf{new} \text{ } Stack$ ;  $V := \mathbf{new} \text{ } L$ ;
     $Push(S, o.previous)$ ;  $V.lgetLength(S, S)$ ;  $Pop(S, o.previous)$ ;  $Pop_i(S, aux)$ 
end

```

There is a recursive call to the method *lgetLength*.

The resulting Class *L* obtained in this compilation phase appears in Figure 4.9.

4.2.5 Proof of Theorem 4.2.1

The proof of Theorem 4.2.1 starts with the introduction of class *L* and *cds_{RVM}* by the application of Law 2.2.77. Then, based on the combination of Rules 4.2.1 and 4.2.2, all calls found in the main command and in class *L* are replaced with calls to methods declared in class *L*. Therefore, the methods declared in *cds* become useless and are eliminated using Law 2.2.84.

Finally, applying Law 2.2.80, from right to left, according with the information recorded by the function *attClass*, we move the attributes to the class where they were originally declared.

Proof:

$$\begin{aligned}
& cds \bullet L \\
= & \{Law\ 2.2.77\} \text{ (Class elimination)} \\
& cds_{RVM}, cds\ L \bullet c \\
= & \{Rule\ 4.2.1\} \langle Redirecting\ call\ with\ result\ parameter \rangle, \\
& \{Rule\ 4.2.2\} \langle Redirecting\ call\ with\ value\ parameter \rangle, \text{ and} \\
& \{Law\ 2.2.84\} \text{ (method elimination)} \\
& \{Law\ 2.2.80\} \text{ (move attribute to superclass) (according to Algorithm in Figure 4.7)} \\
& cds_{RVM}, cds, L' \bullet c'
\end{aligned}$$

◇

These transformations are sufficient to reach the expected outcome of this phase.

4.3 Simplification of expressions and some source transformations

The main objective of this phase is to eliminate nested expressions that appear in assignments and guards. At this point, we also eliminate parametrised commands, and transform conditionals into *if-then-elses*. The objective is to format the commands into patterns closely related to the ones used to define the instructions. The expected outcome for this compilation phase is stated by Theorem 4.3.1.

Theorem 4.3.1 (Simplification of Expressions) *Let $cds_{RVM}, cds, L \bullet c$ be a program, then there is a program $cds_{RVM}, cds, L' \bullet c'$ such that*

```

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

```

Figure 4.9: Class L generated in the phase of Redirection of Method Calls

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

where c' and L' obey the following conditions:

- each assignment operates through the operand stack;
- each boolean expression is a variable or a negation of a variable;
- there is no parametrised recursive calls;
- there is no parametrised command; and
- every conditional is written as an if-then-else.

□

Basically, the task of eliminating nested expressions in a source program 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, in this phase other transformations are performed. We eliminate parametrised commands, simplify guards and conditionals, and eliminate unnecessary sequences of *Push* and *Pop*. In Section 4.3.7 we show the effect of these transformations into our example. The elimination of parametrised commands is the first step in this phase. When a parametrised commands are eliminated a variable block is introduced.

4.3.1 Eliminating parametrised commands

In order to eliminate parametrised commands, such as those appearing in each method declared in L , we rely on Laws 2.2.19 and 2.2.18. Further manipulation is needed to simplify the resulting command. To illustrate how we perform these transformations, consider the parametrised command that appears in the method *lsetDirection*, in Figure 4.9. Based on the basic laws of ROOL, we can reduce it in the following way.

$$\begin{aligned}
& (\mathbf{val} \ d : Int; \bullet (Step \ o).dir := d)(x) \\
= & \{Law \ 2.2.18\} \text{ (Pcom elimination-val)} \\
& \mathbf{var} \ l : Int; \bullet l := x; (Step \ o).dir := l \ \mathbf{end} \\
= & \{Law \ 2.2.54\} \text{ (; := substitution)} \\
& \mathbf{var} \ l : Int; \bullet l := x; (Step \ o).dir := x \ \mathbf{end} \\
= & \{Law \ 2.2.10\} \text{ (var-; right dist)} \\
& \mathbf{var} \ l : Int; \bullet l := x \ \mathbf{end}; (Step \ o).dir := x \\
\sqsubseteq & \{Law \ 2.2.11\} \text{ (var- := final value)} \\
& \mathbf{var} \ l : Int; \bullet \mathbf{end}; (Step \ o).dir := x \\
= & \{Law \ 2.2.6\} \text{ (var elim)} \\
& (Step \ o).dir := x
\end{aligned}$$

In summary, we eliminate the parametrised command. Then, we substitute the expression assigned to $(\text{Step } o).\text{dir}$. At this point, we can contract the scope of the variable block, allowing the assignment to $(\text{Step } o).\text{dir}$ to be out of it. In doing so, the assignment to l becomes irrelevant because it is just before the end of its scope and, therefore, it can be eliminated. Finally, considering that l is never used, we can eliminate its declaration.

Similar transformations are applied to each parametrised command in L . In Figure 4.10, we show the resulting class L obtained after the elimination of parametrised commands.

4.3.2 Parametrised Recursion

In this section we deal with the elimination of parametrised recursive calls that might appear in the methods declared in L . They are eliminated by defining a recursive method lm with the use of the recursive command **rec** $X \bullet c$ **end**, using a strategy similar to that presented in [87] for parametrised recursive programs. In the following, we explain how we perform this transformation.

The basic intuitive interpretation of the recursive command is that of *recursive unfoldings* [9]. During the execution of c , whenever the statement X is encountered, it is equivalent to the replacement of X with the whole recursive command **rec** $X \bullet c$ **end**.

A body of a parametrised recursive method lm is a parametrised command, and all recursive calls are of the form $le.lm(t)$, for some target object le and actual parameter t . The meaning of a parametrised recursive method is given by the fixed point (Law 2.2.67) and the least fixed point (Law 2.2.68) laws.

Consider a recursive parametrised method lm declared in L , having in its body a code fragment with a recursive call, as follows.

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

Then, each parametrised recursive call, such as the above $o.lm(S, S)$, is replaced by a code fragment with a parameterless recursive call to X , in such a way that the recursive call $o.lm(S, S)$ becomes as follows.

$$\mathbf{dvar} \ S_{in}, S_{out} : \text{Stack}; \ S_{in} := S; \ X; \ S := S_{out}; \ \mathbf{dend} \ S_{in}, S_{out}$$

A non-recursive call gives the initial values of S_{in} and S_{out} . After that, when a parameterless recursive call is activated, the initial value of S_{in} , which is the value parameter, is given by S . Similarly, when returning from a call, the current value of S_{out} , which is the result parameter, is copied to S . Note that to achieve the same behavior when the recursive call is activated, the use of dynamic declarations are essential. Actually, the fact of the matter is that they represent an abstract stack which simulates the way parameters are implemented in practice. For each recursive call, the value of the arguments are stored in an activation record [3] in a run time stack; this stack is also used to store local variables. Theorem 4.3.2 formalises this transformation.

```

class L
  meth lsetDirection  $\hat{=}$  (val  $S_{in} : Stack$ ; res  $S_{out} : Stack \bullet$ 
    var  $o : Object$ ;  $x : int \bullet Pop(S_{in}, o)$ ;  $Pop_i(S_{in}, x)$ ;
    (Step  $o$ ).dir :=  $x$ ;  $Push(S_{in}, o)$ ;  $S_{out} := S_{in}$ 
  end)
  meth lsetLength  $\hat{=}$  (val  $S_{in} : Stack$ ; res  $S_{out} : Stack \bullet$ 
    var  $o : Object$ ;  $x : int \bullet Pop(S_{in}, o)$ ;  $Pop_i(S_{in}, x)$ ;
    (Step  $o$ ).len :=  $x$ ;  $Push(S_{in}, o)$ ;  $S_{out} := S_{in}$ 
  end)
  meth laddStep  $\hat{=}$  (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  $o$ );
    var  $S : Stack$ ;  $V : L \bullet S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push_i(S, x)$ ;  $Push(S, o)$ ;  $V.lsetDirection(S, S)$ ;  $Pop(S, o)$ ;
  end
    var  $S : Stack$ ;  $V : L \bullet S := \text{new } Stack$ ;  $V := \text{new } L$ ;
     $Push_i(S, y)$ ;  $Push(S, o)$ ;  $V.lsetLength(S, S)$ ;  $Pop(S, o)$ ;
  end;  $Push(S_{in}, o)$ ;  $S_{out} := S_{in}$ 
  end)
  meth lgetLength  $\hat{=}$  (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  $S : Stack$ ;  $V : L \bullet S := \text{new } Stack$ ;  $V := \text{new } L$ ;
           $Push(S, o.previous)$ ;  $V.lgetLength(S, S)$ ;
           $Pop(S, o.previous)$ ;  $Pop_i(S, aux)$ 
        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

```

Figure 4.10: Class L after the elimination of parametrised commands

Theorem 4.3.2 (Parameters of recursion)

$$\begin{aligned} & \text{rec } X \bullet (\text{val } x : T; \text{ res } y : T \bullet p) \\ \sqsubseteq & \\ & \text{val } x : T; \text{ res } y : T \bullet (\text{rec } X \bullet p[(\text{dpar } x, y : T \bullet X)/X]) \end{aligned}$$

where

$$(\text{dpar } x, y : T \bullet X)(s, t) \stackrel{\text{def}}{=} \text{dvar } x, y : T; \ x := s; \ X; \ t := y; \ \text{dend } x, y \quad \square$$

Proof:

$$\begin{aligned} & RHS \sqsubseteq RHS \\ \equiv & \{Law2.2.65\} (\text{rec fixed point}) \\ & \left(\text{val } x : T; \text{ res } y : T \bullet \right. \\ & \quad \left. p[(\text{dpar } x, y : T \bullet (\text{rec } X \bullet p[(\text{dpar } x, y : T \bullet X)/X] \text{ end})) / X] \right) \sqsubseteq RHS \\ \equiv & \{Law2.2.28\} (\text{dvar; dend conversion}), \{Law2.2.4\} (\text{var association}), \\ & \{Law2.2.18\} (\text{Pcom elimination-val}), \text{ and } \{Law2.2.19\} (\text{Pcom elimination-res}) \\ & \left(\text{val } x : T; \text{ res } y : T \bullet \right. \\ & \quad \left. p[(\text{val } x : T; \text{ res } y : T \bullet (\text{rec } X \bullet p[(\text{dpar } x, y : T \bullet X)/X] \text{ end})) / X] \right) \sqsubseteq RHS \\ \equiv & \{\text{Note that the above left-hand side is a function of RHS}\} \\ & (\text{val } x : T; \text{ res } y : T \bullet p[RHS/X]) \sqsubseteq RHS \\ \Rightarrow & \{Law2.2.66\} (\text{rec least fixed point}) \\ & LHS \sqsubseteq RHS \end{aligned}$$

◇

Observe that the proof of the above theorem relies on the use of dynamic variable declarations. As explained in Chapter 2, **dvar** and **dend** operate on an implicit stack. Therefore, due to the dynamic declarations the occurrences of x and y in **dpar** are free. Hence, the application of the fixed point law in the first step of the proof does not cause the renaming of x and y . As the result in the first step, the recursive call is replaced with the entire recursive method. At this point of the proof, it is not possible to distinguish between a static and a dynamic declaration of x and y , the necessary conditions are satisfied, and therefore, the conversion can be made. The least fixed point law establishes the final result.

In our target machine, we have an explicit run time stack, named F . The following rule is a corollary of the above theorem, and allows us to introduce the frame stack F , replacing the implicit stack with *Push* and *Pop*.

Rule 4.3.1 (Stack implementation — Parameters of recursion))

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

where

$$\begin{aligned} (\text{dpar } x, y : T \bullet X)(s, t) &\stackrel{\text{def}}{=} \text{Push}(F, x); \text{Push}(F, y); \\ &x := s; X; t := y; \\ &\text{Pop}(F, y); \text{Pop}(F, x) \end{aligned}$$

□

Proof: From Law 2.2.27 ((*Push*, *Pop*) – (**dvar**, **dend**) conversion) we transform the dynamic constructors, introducing the stack F and replacing **dvar** and **dend** with *push* and *pop*, respectively; then, Theorem 4.3.2 (Parameters of recursion) establishes the parameter elimination. *Diamond*

In our example, the only method that requires the elimination of parametrised recursion is *lgetLength*. In Figure 4.15, we present the transformed version of this recursive method, in which the recursion is defined solely with the use of the recursive command **rec** $X \bullet c$ **end**. At this point, in the body of this method, there are no calls of the form $o.lgetLength(S, S)$, because its name was eliminated in its body. Nevertheless, the same effect is obtained with the use of the recursive command. Whenever the statement X is encountered, the recursive unfolding simulates the recursive call for this method.

4.3.3 Simplifying Guards

The boolean expressions appearing in **while** and **if** commands may be arbitrarily nested. They need to be simplified to match the patterns used to reduce these commands in the Control Elimination phase. The next theorem shows how the conditions of an **if** command can be simplified.

Rule 4.3.2 (Conditions of **if**)

If x_i do not occur in b_i nor in p_i , for $1 \leq i \leq n$, then

$$\begin{array}{l} \text{if } \bigwedge_{1 \leq i \leq n} b_i \rightarrow p_i \text{ fi} \quad \sqsubseteq \quad \text{var } x_1, \dots, x_n : \text{boolean} \bullet \\ \quad x_1 := b_1; \dots; x_n := b_n; \\ \quad \text{if } \bigwedge_{1 \leq i \leq n} x_i \rightarrow p_i \text{ fi;} \\ \text{end} \end{array}$$

□

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 4.3.5. In our example, the boolean expression (*o is Step*) is assigned to a boolean variable b_2 , and b_2 replaces the expression in the conditional.

The following theorem shows how the condition of a **while** command can be simplified.

Rule 4.3.3 (Condition of **while**) *If x does not occur in b nor in p ;*

$$\begin{array}{l} \text{while } b \bullet p \text{ end} \quad \sqsubseteq \quad \text{var } x : \text{boolean} \bullet \\ \quad x := b; \text{ while } x \bullet p; x := b \text{ end} \quad \square \\ \text{end} \end{array}$$

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

4.3.4 Transforming Conditionals

In the Control Elimination phase, 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 2.2.45 allows us to have simpler compilation rules to deal with conditionals.

4.3.5 Rewriting Assignments

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

Rule 4.3.4 (Assignment to a variable)

If S does not occur in e or f

$$\begin{array}{l} (x := e) = \text{var } S : \text{Stack}; \bullet \\ \quad S := \text{new Stack}; \text{ Push}(S, e); \text{ Pop}(S, x) \\ \text{end} \end{array}$$

□

In the above rule, for each assignment, a variable block declaring S is introduced, S is initialised, the expression e is pushed onto the operand stack, and then, the value is popped from the top of

the stack and assigned to the variable x . The expression e above may be arbitrarily nested; we need to further simplify it to achieve a simpler form, in which e consists of a variable, an attribute selection, or a constant.

The next rule handles $Push(S, e)$, when e is a binary expression.

Rule 4.3.5 (Binary operator)

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

where $Bop(S)$ represents the application of an arbitrary binary operator to the two topmost elements of the operand stack. $\square$

The nested expression in the method call $Push(S, e \textbf{ bop } f)$ 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 $Push(S, e)$, whose argument is an application of a unary operator.

Rule 4.3.6 (Unary operator)

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

where $Uop(S)$ represents the application of an arbitrary unary operator to the element at the top of the operand stack. $\square$

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

Rule 4.3.7 (Pushing an attribute)

If $cds_{RVM} cds, N \triangleright le : C$

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

$\square$

The expression le above may be arbitrarily nested; we need to further simplify it to achieve a simpler form, applying this rule again on $Push(S, le)$. The abbreviation $Getfd(S, C.t)$ is described in Definition 4.2.1.

Rule 4.3.8 (Popping to an attribute)

If $cds_{RVM} cds, N \triangleright le : C$

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

□

We use $Push(S, le)$ and $Pop(S, le)$ because the expression le may be arbitrarily nested, and then may need further simplification to achieve a simpler form. The abbreviation $Putfd(S, C.t)$ is described in Definition 4.2.1.

Object creation is just another instance of assignment.

Rule 4.3.9 (Object Creation - Simplification)

$$\begin{aligned} (x := \mathbf{new} C) = & \mathbf{var} S : Stack; \bullet \\ & S := \mathbf{new} Stack; Push(S, \mathbf{new} C); Pop(S, x) \\ & \mathbf{end} \end{aligned}$$

□

In this case, we need to use $Push(S, \mathbf{new} C)$ to push this object onto the operand stack. Observe that $Push(S, \mathbf{new} C)$ does not need to be simplified anymore.

When the guards are simplified (Section 4.3.3), type testing is rewritten to be an expression that is assigned to a boolean variable. Therefore, the rule for type test is just a specific case of the rule for assignment in which the boolean type of the target of the assignment must be considered.

Rule 4.3.10 (Type test - Simplification)

$$\begin{aligned} (x := (o \text{ is } C)) = & \mathbf{var} S : Stack; \bullet \\ & S := \mathbf{new} Stack; Push(S, o); Is(S, C); Pop_b(S, x) \\ & \mathbf{end} \end{aligned}$$

□

Observe that the object o in $Push(S, o)$ might not be a simple object, 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 the type C . Moreover, the result of this test is encapsulated in an instance of *DataBool* and pushed onto S . This instance of *DataBool* has one attribute named *Info*, which has a primitive 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 the boolean variable x .

4.3.6 Irrelevant sequences of *Push* and *Pop*

During this phase of compilation several variable blocks are introduced containing sequences of *Push* and *Pop* involving the operand stack S . Then, expanding the scope of the variable blocks, we are able to apply laws to eliminate unnecessary assignments to initialise the variables S and V , as well as sequences of *Push* and *Pop*.

The next lemma asserts that an assignment does not affect the result stack.

Lemma 4.3.1 (Assignment does not affect the resulting stack)

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

□

Remember that the initialiser declared in the class *Stack* (Figure 4.3.4) assigns an empty sequence to its attribute just after a creation of an object. Therefore, we have the following Lemma.

Lemma 4.3.2 (Initial empty stack)

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

$$\text{where } empty(S) \stackrel{def}{=} S.s = \langle \rangle$$

□

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 .

In the variable block introduced by Rule 4.3.4, the assumption $\{empty(S)\}$ is 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 following rule states that $\text{Push}(S, o)$ in sequential composition with $\text{Pop}(S, o)$ has no effect whatsoever. During compilation, such an irrelevant pair of *Push* and *Pop* can be eliminated using this rule.

Rule 4.3.11 (*Push*; *Pop* effect)

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

□

Nevertheless, when we have $\text{Pop}(S, x)$ followed by $\text{Push}(S, x)$, the following lemma states that this has the same effect as assigning the head of S to x .

Lemma 4.3.3 (*Pop*; *Push* effect)

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

□

Therefore, the pair $(Pop(S, x), Push(S, x))$ is not a simulation, despite that there is another rule that allows us to eliminate $Pop(S, x)$ followed by $Push(S, x)$. This particular case is formalised by the following rule:

Rule 4.3.12 ($Pop(S, x); Push(S, x)$) — assignment

$$Pop(S, x); Push(S, x); V.lm(S, S); Pop(S, x) = V.lm(S, S); Pop(S, x)$$

□

Whenever we have $Pop(S, x); Push(S, x)$ in the above context, they can always be eliminated.

4.3.7 Our example

We show next how the main command is transformed. Consider the main command shown in Figure 4.8. Applying Laws 2.2.9 (**var**-, left dist), 2.2.10 (**var**-, right dist), 2.2.12 (**var**-, dist), and 2.2.4 (**var** association) we expand the scope of the variable blocks. Then the elimination of irrelevant assignments to the variables S and V relies on the application of Lemmas 4.3.1 (Assignment does not affect the resulting stack) and 4.3.2 (Initial empty stack), and Laws 2.2.55 ($;\ :=$ commutation), 2.2.57 ($;\ :=$ void assignment), and 2.2.62 ($;\ [b]; \{b\}$ simulation). Doing so, we obtain the code shown in Figure 4.11.

```

var  $p$  : Path •
   $p$  := new Path;
  var  $S$  : Stack;  $V$  :  $L$  •  $S$  := new Stack;  $V$  := new  $L$ ;
    Push $i$ ( $S$ , north); Push $i$ ( $S$ ,  $l_1$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ ); Pop( $S$ ,  $p$ );
    Push $i$ ( $S$ , east); Push $i$ ( $S$ ,  $l_2$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ ); Pop( $S$ ,  $p$ );
    Push $i$ ( $S$ , south); Push $i$ ( $S$ ,  $l_3$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ ); Pop( $S$ ,  $p$ );
    Push $i$ ( $S$ , west); Push $i$ ( $S$ ,  $l_4$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ );
    Pop( $S$ ,  $p$ ); Push( $S$ ,  $p$ )
    V.lgetLength( $S$ ,  $S$ ); Pop( $S$ ,  $p$ ); Pop $i$ ( $S$ , out)
  end
end

```

Figure 4.11: Intermediate main command obtained during the third compilation phase

The main command presented in Figure 4.11 has a sequence of irrelevant *Pop* and *Push* just before the call *V.lgetLength*(S , S). Applying Rule 4.3.12 we eliminate this irrelevant sequence we obtain the code shown in Figure 4.12.

```

var  $p$  : Path •
   $p$  := new Path;
  var  $S$  : Stack;  $V$  :  $L$  •  $S$  := new Stack;  $V$  := new  $L$ ;
    Push $i$ ( $S$ , north); Push $i$ ( $S$ ,  $l_1$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ ); Pop( $S$ ,  $p$ );
    Push $i$ ( $S$ , east); Push $i$ ( $S$ ,  $l_2$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ ); Pop( $S$ ,  $p$ );
    Push $i$ ( $S$ , south); Push $i$ ( $S$ ,  $l_3$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ ); Pop( $S$ ,  $p$ );
    Push $i$ ( $S$ , west); Push $i$ ( $S$ ,  $l_4$ ); Push( $S$ ,  $p$ ); V.laadStep( $S$ ,  $S$ );
    V.lgetLength( $S$ ,  $S$ ); Pop( $S$ ,  $p$ ); Pop $i$ ( $S$ , out)
  end
end

```

Figure 4.12: Resulting main command obtained after the third compilation phase

The only change observed in the methods *lsetDirection* and *lsetLength* is caused by the application of Rule 4.3.4. In Figure 4.13 we can see how the assignments to *o.dir* and *o.len* were transformed. Those transformations have been applied to the corresponding methods in Figure 4.10. Similarly, we can verify that the major changes occurred in *laddStep*, in Figure 4.14, were due to the application of Rule 4.3.4.

```

meth lsetDirection  $\hat{=}$  (val  $S_{in} : Stack$ ; res  $S_{out} : Stack$  •
  var  $o : Object$ ;  $x : int$  •  $Pop(S_{in}, o)$ ;  $Pop_i(S_{in}, x)$ ;
  var  $S : Stack$  •  $S := \mathbf{new} Stack$ ;
     $Push(S, x)$ ;  $Pop(S, (Step\ o).dir)$ 
  end;  $Push(S_{in}, o)$ ;  $S_{out} := S_{in}$ 
end)

meth lsetLength  $\hat{=}$  (val  $S_{in} : Stack$ ; res  $S_{out} : Stack$  •
  var  $o : Object$ ;  $x : int$  •  $Pop(S_{in}, o)$ ;  $Pop_i(S_{in}, x)$ ;
  var  $S : Stack$  •  $S := \mathbf{new} Stack$ ;
     $Push(S, x)$ ;  $Pop(S, (Step\ o).len)$ 
  end;  $Push(S_{in}, o)$ ;  $S_{out} := S_{in}$ 
end)

```

Figure 4.13: Methods *lsetDirection* and *lsetLength* after the third compilation phase

```

meth laddStep  $\hat{=}$  (val  $S_{in} : Stack$ ; res  $S_{out} : Stack$  •
  var  $o : Object$ ;  $x, y : int$  •  $Pop(S_{in}, o)$ ;  $Pop_i(S_{in}, x)$ ;  $Pop_i(S_{in}, y)$ ;
  var  $S : Stack$ ;  $V : L$  •  $S := \mathbf{new} Stack$ ;  $V := \mathbf{new} L$ ;
     $Push(S, (Path\ o); )$ ;  $Pop(S, (Path\ o).previous)$ ;
     $Push_i(S, x)$ ;  $Push(S, o)$ ;  $V.lsetDirection(S, S)$ ;  $Pop(S, o)$ ;
     $Push_i(S, y)$ ;  $Push(S, o)$ ;  $V.lsetLength(S, S)$ ;  $Pop(S, o)$ ;
  end;  $Push(S_{in}, o)$ ;  $S_{out} := S_{in}$ 
end)

```

Figure 4.14: Method *laddStep* after the after the third compilation phase


```

meth lgetLength  $\hat{=}$  (val Sin : Stack; res Sout : Stack •
  var F : Stack •
    F := new Stack;      rec X •
      var x, aux : Int; o : Object; x1, x2 : Boolean; S : Stack; V : L •
        Pop(Sin, o); S := new Stack; V := new L;
        Push(S, o); Is(S, Path); Popb(S, x1);
        Push(S, o) Getfd(S, Path.previous); Push(S, null); cmpeq(S); Popb(S, x2)
        if x1 →
          if x2 → Push(S, 0); Pop(S, aux)
            Push(S, o); Getfd(S, Path.previous);
          □ ¬(x2) → Push(S, 0); Pop(S, aux)
            Push(S, o); Getfd(S, Path.previous);
            Push(F, Sin); Push(F, Sout);
            Sin := S; X; Sout := S;
            Push(F, Sout); Push(F, Sin);
            Push(S, o); Putfd(S, Path.previous); Pop(S, o); Popi(S, aux)
          fi; Push(S, o); Getfd(S, Step.len); Pop(S, x);
          Push(S, x); Push(S, aux); Add(S); Pop(S, x)
          □ ¬(x1) → Push(S, o); Getfd(S, Step.len); Pop(S, x)
        fi;
      end Pushi(Sin, x); Push(Sin, o); Sout := Sin
    end
  end)
end

```

Figure 4.15: Method *lgetLength* after the third compilation phase

Observing the method *lgetLength* in Figure 4.15, we verify that the parametrised recursion was eliminated with the introduction of the recursive command **rec** $X \bullet c$ **end**. Moreover, all conditionals are in an *if-then-else* style, and all assignments are operated through the operand stack.

4.3.8 Proof of Theorem 4.3.1

The proof of Theorem 4.3.1 is as follows.

Proof: First, combining the application of Laws 2.2.19 and 2.2.18, parametrised commands are eliminated. Then, Rule 4.3.1 yields the elimination of parametrised recursive calls. After that, based on Law 2.2.45 and rules from 4.3.4 to 4.3.3, assignments, conditionals and iteration are transformed. Applying Laws 2.2.10, 2.2.9, 2.2.12 and 2.2.4 the scope of the blocks of variables are expanded. Irrelevant sequences of *Push* and *Pop* are eliminated by the application of Rules 4.3.11 and 4.3.12.

These transformations are sufficient to reach the outcome of this phase. $\diamond$

4.4 Control Elimination

Control Elimination consists of reducing the nested control structure of the source program to a single flat iteration. The outcome of this phase is a program whose control structure is the same as that of our normal form. The next theorem summarises the result.

Theorem 4.4.1 (Control Elimination)

Consider a program

$$cds_{RVM}, cds, L \bullet \mathbf{var} \ v : T \bullet q \ \mathbf{end}$$

where *cds* has only declarations of public attributes, and methods declared in *L* have the following form

$$\mathbf{meth} \ lm \stackrel{\wedge}{=} (\mathbf{val} \ S_{in} : Stack; \ \mathbf{res} \ S_{out} \bullet \mathbf{var} \ w : T \bullet p \ \mathbf{end})$$

The program also satisfies the following conditions: (1) in *p* and in *q* 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 as an *if-then-else*, (5) and there are no parametrised recursive calls (6) nor parametrised commands.

Then, there is a command *r* such that $cds_{RVM}, cds, L \bullet q \sqsubseteq cds_{RVM} \ cds \bullet r$.

The main command *r* has the same control structure as *I* in the normal form $cds_{RVM} \bullet I$, but *r* still operates on the abstract space. $\square$

To accomplish the goal established by this theorem, we apply, to the main command and to L , the compilation rules explained in the following two subsections.

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 stage we introduce the program counter PC .

Here we represent the RVM instructions as abbreviations of sequences of ROOL commands, as described in Section 3.4. Moreover, for convenience, we define an abbreviation for a command which is very similar to the main command of our interpreter, but declares just one of the *RVM* components, the PC counter.

Definition 4.4.1 (Abbreviation for the PC frame)

$$\begin{aligned}
 PC : [s, GCS, f] &\stackrel{def}{=} \text{var } PC : Int; \bullet \\
 &\quad PC := s; \\
 &\quad \text{while } PC \geq s \wedge PC < f \rightarrow \\
 &\quad \quad \text{if } GCS \text{ fi} \\
 &\quad \text{end;} \\
 &\quad \{PC = f\} \\
 &\text{end}
 \end{aligned}$$

□

The following patterns are introduced in this phase. They are very similar to the *RVM*'s instructions, but they still operate on the abstract space.

Definition 4.4.2 (Abbreviations introduced in the Control Elimination phase)

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

□

The abbreviation $Load_{CE}(e)$ pushes an expression onto the operand stack. To pop a value from the operand stack and assign it to a variable, we use the abbreviation $Store_{CE}(x)$. In accordance with the *JVM* instruction set, we use the abbreviation $aconstnull$ to push the constant **null** onto the operand stack. 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, in the following we introduce abbreviations to deal with integer values.

Definition 4.4.3 (Abbreviations introduced in the Control Elimination phase for integer and boolean values)

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

□

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 value and assigns it to a variable; in the same way, $bStore_{CE}$ is the version which deals with boolean values.

Each abbreviation introduced in this phase is almost a bytecode instruction for the *RVM*. The only difference resides in the fact that these abbreviations still refer to variables and classes declared in the source program.

In order to maintain the correspondence with the *JVM*, the PC increment takes into consideration the number of bytecodes used by the associated instruction in the bytecode stream. Therefore, besides the bytecode necessary to denote the instruction itself, an additional bytecode is necessary for each instruction argument. Since the abbreviation $Load_{CE}(e)$ has one argument, two bytecodes are necessary: one for the opcode, and the other one to hold x . Therefore, the PC is incremented by two.

In the following rules, abbreviations such as $Push(S, x)$ are reduced to the form given in Definition 4.4.1. Here we assume that S and x are different from PC and, for this reason, we omit these side conditions in these rules.

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

Rule 4.4.1 (Skip)

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

□

As shown in Definition 4.4.2, the only effect of the *nop* instruction is the *PC* increment.

The next rules deal with the elements of abbreviations introduced in the previous phases. The following rule considers the pattern that loads a variable.

Rule 4.4.2 (Push variable)

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

□

As previously defined, $Load_{CE}(x)$ is an abbreviation of a sequence of ROOL commands that places the value of variable x at the top of the operand stack S , and increments the *PC* by two.

Now we show the reduction of a pattern that stores an integer variable.

Rule 4.4.3 (Pop variable)

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

□

Like the *Load* instruction, the *Store* instruction also requires two bytecodes. Nevertheless, instead of pushing a value onto the operand stack S , a value is popped from S and stored in x .

The following rule deals with object creation.

Rule 4.4.4 (Object Creation)

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

□

The abbreviation $Load_{CE}(\mathbf{new} C)$ pushes a fresh object onto the operand stack.

The following rule addresses type testing.

Rule 4.4.5 (Type Test)

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

□

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

Rule 4.4.6 (Set an attribute)

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

□

The following rule addresses the sequence of commands that pops an object and gets the value of its attribute t and pushes it onto the operand stack.

Rule 4.4.7 (Set an attribute)

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

□

To reduce the method call associated to the binary operator, we use the following rule.

Rule 4.4.8 (Binary Operator)

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

□

The above Bop is an abbreviation of a sequence of ROOL commands that removes two values from the operand stack S , operates on 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. Observe that the Bop introduced in this phase is exactly the same given in the RVM 's instruction set (see Definition 3.4.4).

The next rule deals with the reduction of the method call associated to the unary operator.

Rule 4.4.9 (Unary Operator)

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

□

Similarly to the Bop abbreviation, Uop is also a sequence of ROOL commands. It pops one value from the operand stack S and applies the unary operand on it, pushing the resulting value at the top of S (see Definition 3.4.4).

The following rule shows how a conditional can be transformed to the reduced form when its branches have already been reduced.

Rule 4.4.10 (Conditional)

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

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

□

The guards in R represent a conditional branch in the execution flow, and assign to PC the location of the next instruction to be executed, according to the value in x .

In order to transform an iteration into a reduced command, we assume that its body is already reduced.

Rule 4.4.11 (Iteration)

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

where

$$R = \left(\begin{array}{l} PC = s \rightarrow (\mathbf{if} \ x \rightarrow PC := PC + 1 \\ \quad \square \neg(x) \rightarrow PC := f_0 + 1 \\ \quad \mathbf{fi}) \\ \square GCS \square PC = f_0 \rightarrow (PC := s) \end{array} \right)$$

□

The reduction of sequential composition assumes that both arguments are already in the normal form and the final position (j) of the guarded command set (GCS_1) of the left argument coincides with the initial position of the guarded command set (GCS_2) of the right argument.

Rule 4.4.12 (Sequential composition)

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

□

The resulting normal form has the initial position (s) of the left argument and has the final position (f) of the right argument.

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

Lemma 4.4.1 (*Commands in normal form*)

If w and PC are not free in p then

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

□

The reduction of a method call follows from Lemma 4.4.1.

Rule 4.4.13 (Method Call)

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

□

Observe that the form of the method call in the above rule is similar to those we obtain after performing the Redirection of Method Calls.

Similarly, the rule to reduce assignments is as follows.

Rule 4.4.14 (Assignment)

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

□

The application of the above rules results in a program of the form

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

Applying Law 2.2.4, we can join the variable blocks obtaining a program which is similar to the normal form command, contains its components, but still operates on the abstract data space. This program is described by Definition 4.4.4

Definition 4.4.4 (Abbreviation for the flat iteration)

$$\begin{aligned} w : [s, GCS, f] &\stackrel{def}{=} \text{var } PC : Int; F, S : Stack; v : T \bullet \\ &\quad F := \text{new } Stack; S := \text{new } Stack; PC := s; \\ &\quad \text{while } PC \geq s \wedge PC < f \rightarrow \\ &\quad \quad \text{if } GCS \text{ fi} \\ &\quad \text{end;} \\ &\quad \{PC = f\} \\ &\text{end} \end{aligned}$$

□

where w represents a list of variables comprising the *RVM*'s components PC , S , F , and the list of local variables v declared in the main command or in a method body. Recall that F and S represent the frame stack and operand stack, respectively.

Both F and S are not introduced by the rules in this section. The operand stack S is introduced by the rules of Redirection of Method Calls, and assignment simplifications, whereas 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, we finally produce a program $w_m : [s_m, GCS_m, f_m]$ in the body of each method declared in L , and $w_c : [s_c, GCS_c, f_c]$ in the main command. In Figure 4.16, we present the general form of the resulting program at this stage.

```

CdsRVM, cds,
class L
  {meth lmi  $\hat{=}$  (val Sin : Stack res Sout : Stack •
    rec X • wm : [sm, GCSm, fm] end)*
  end
  • wc : [sc, GCSc, fc] end

```

Figure 4.16: Intermediate program obtained before the elimination of L

There might exist several method declarations in L , each one operating over an assumed disjoint interval established by the pair of values (s_m, f_m) . The main command also operates over 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 4.16 to our normal form, it is necessary to eliminate the class L . This is the objective of the next sections. The major obstacle to reduce class L resides in the calls to methods in L that may exist in GCS_m and in GCS_c (Figure 4.16). Recursive parametrised calls were eliminated in Section 4.3.2. Nevertheless, recursive methods might exist, implemented using the recursive command **rec** X • p **end**. To eliminate the class L , we handle recursion and method calls separately, so that the overall task becomes modular.

4.4.2 Recursive commands in L

The strategy adopted here follows the reduction of recursion found in [87]. We simply adapted it to our notation.

Before we introduce the reduction rule that eliminates the command **rec** X • p **end** in the recursive methods of L , some abbreviations are introduced to help structure the proof. The left-hand side of the reduction rule is given by the following recursive command.

Consider a recursive method lm , possibly declared in L ,

$$\mathbf{meth} \ lm \triangleq (\mathbf{val} \ S_{in} : \mathit{Stack}; \mathbf{res} \ S_{out} : \mathit{Stack} \bullet \mathit{LHS})$$

where

$$\mathit{LHS} = \mathbf{rec} \ X \bullet v : [s, \left(\begin{array}{l} b_1 \rightarrow X; \ PC := PC + 1 \\ \square \ b_2 \rightarrow q \end{array} \right), f] \mathbf{end}$$

The body of the above recursive command has the same structure of the main command of the normal form, except for the recursive calls. For conciseness, here we assume the existence of only one recursive call to X to be eliminated. Therefore, there are no calls to X to be eliminated in q . Nevertheless, the rule can be easily generalised to deal with an arbitrary number of calls.

In the previous section, the run time stack F was necessary to save the parameters when a recursive call was performed. At this point of the compilation process, it is also necessary to store the values of the local variables declared in the method body whenever a recursive call is simulated.

In *MID* bellow, when a recursive method call is simulated, F_v holds the values of the local variables v declared in the method body, including the returning address in PC . Thereafter, we modify the recursion to X , so that we replace an occurrence of X with commands that push the local variables onto the stack F_v before transferring the control to the initial address of the method ($PC = s$).

$$\mathit{MID} = (\mathbf{vres} \ x:T \bullet v, F_v : [s, U, f])$$

$$\text{where } U = \left(\begin{array}{l} b_1 \rightarrow PC := PC + 1; \ \mathit{Push}(F_v, v); \ PC := s + 1; \\ \square \ (PC = f_0 \rightarrow PC := f; \\ \qquad \text{If } \neg(\mathit{empty}(F_v)) \rightarrow \mathit{Pop}(F_v, v) \ \mathbf{fi}) \\ \square \ b_2 \rightarrow q \end{array} \right)$$

where f_0 is the penultimate location, just before f .

Recall that v consists of an arbitrary list of variables, including those data variables declared by the programmer in the source program. Since the emphasis here is on the control structure of the program, we do not distinguish between the source variables and the variables introduced in the preceding compilation phases.

Since the frame stack F_v is an instance of Stack , we know it is initially empty. Therefore, checking whether F_v is empty or not, it is possible to distinguish between the exit from a recursive call and a call originated in the main command. When the program counter reaches the address f and the frame stack F_v is not empty, it means that the exit from a recursive call is being performed. Then, control is resumed by popping the local variables off from F_v and assigning them to v . For this reason, PC gets the address of the instruction that follows the terminated call. Finally, the

control returns to the main command when PC reaches the address f and the frame stack F_v is empty.

As an alternative implementation of LHS , we present the following program:

$$RHS = v, F_v : [s, Z, f]$$

$$\text{where } Z = \left(\begin{array}{l} (PC = s) \rightarrow PC := f; \text{ Push}(F_v, v); \text{ PC} := s + 1; \\ \square \quad b_1 \rightarrow PC := PC + 1; \text{ Push}(F_v, v); \text{ PC} := s + 1; \\ \square \quad (PC = f_0) \rightarrow \text{Pop}(F_v, v) \\ \square \quad b_2 \rightarrow q \end{array} \right)$$

where f_o is the penultimate location, just before before f .

Analysing RHS , we note that its first action is to push PC onto the stack F_v with a value that satisfies the exit condition for the loop. The main advantage of the above implementation resides in the fact that it avoids the test of the emptiness of F_v in order to activate the exit condition since the last value to be popped makes the value of PC to be $f + 1$. This fact makes RHS a more suitable option for a low-level implementation.

The following two lemmas are used to prove that $LHS \sqsubseteq RHS$. To do so, it is convenient to show that $LHS \sqsubseteq MID$ and that $MID \sqsubseteq RHS$

Lemma 4.4.2 (Symbolic execution of U) *Let $d = ((PC \geq s) \wedge (PC < f))$.*

$$\begin{array}{l} \{PC = s\}; \text{ while } d \rightarrow \text{ if } U \text{ fi end} \\ \sqsubseteq \\ [\text{empty}(F_v); \text{ MID}; \text{ PC} := \text{PC} + 1; \text{ while } d \rightarrow \text{ if } U \text{ fi end} \end{array}$$

□

Lemma 4.4.3 (Symbolic execution of Z) *Let $d = ((PC \geq s) \wedge (PC < f))$.*

$$\begin{array}{l} \{PC = s\}; \text{ while } d \rightarrow \text{ if } Z \text{ fi end} \\ \sqsubseteq \\ [\text{empty}(F_v)]; \text{ MID}; \text{ PC} := f; \text{ while } d \rightarrow \text{ if } Z \text{ fi end} \end{array}$$

□

Since the proofs of these lemmas are similar, only the proof of the Lemma 4.4.2 is presented in Appendix A. In the following we prove the elimination of the recursive command.

Rule 4.4.15 (Recursion elimination)

Provided LHS , MID and RHS are defined as above, X is not free in q , and F_v occurs only where explicitly shown, then

$$LHS \sqsubseteq RHS$$

□

Proof: ($LHS \sqsubseteq MID$)

MID

$\sqsubseteq \{Law\ 2.2.74\}$ (**while** – **if** replace guarded command) and

$\{Lemma\ 4.4.2\}$ (Symbolic execution of U)

$$w, F : [s, \left(\begin{array}{l} b_1 \rightarrow [empty(F)];\ MID;\ PC := r; \\ \square\ (PC = f_0 \rightarrow PC := f; \\ \quad\quad\quad If\ \neg(empty(F)) \rightarrow Pop(F, w)\ \mathbf{fi}) \\ \square\ b_2 \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.40\}$ (**if** - Guarded command introduction)

$$w, F : [s, \left(\begin{array}{l} b_1 \rightarrow [empty(F)];\ MID;\ PC := r; \\ \square\ (PC = f_0 \rightarrow PC := f; \\ \square\ b_2 \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.62\}$ ($;$ $[b]$; $\{b\}$ simulation)

$$w, F : [s, \left(\begin{array}{l} b_1 \rightarrow [empty(F)];\ \{empty(F)\};\ MID; \\ \quad\quad\quad PC := r;\ \{empty(F)\}; \\ \square\ b_2 \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.62\}$ ($;$ $[b]$; $\{b\}$ simulation) and $\{Law\ 2.2.64\}$ ($;$ $:=$ commute assertion)

$$w, F : [s, \left(\begin{array}{l} b_1 \rightarrow MID;\ PC := r;\ [empty(F)];\ \{empty(F)\}; \\ \square\ b_2 \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.62\}$ ($[b]$; $\{b\} = [b] \sqsupseteq \mathbf{skip}$) and $\{Law\ 2.2.6\}$ (**var** elim)

$$w, F : [s, \left(\begin{array}{l} b_1 \rightarrow MID;\ PC := r \\ \square\ b_2 \rightarrow q \end{array} \right), f]$$

$\Rightarrow \{Law\ 2.2.66\}$ (**rec** least fixed point)

LHS

$(MID \sqsubseteq RHS):$

$$\begin{aligned}
& RHS \\
& \sqsubseteq \{Law\ 2.2.74\} \text{ (**while** – **if** replace guarded command) } \text{ and } \\
& \quad \{Lemma\ 4.4.3\} \text{ (Symbolic execution of Z)} \\
& w, F : [s, \left(\begin{array}{l} (PC = s) \rightarrow \{empty(F)\}; MID; PC := f; \\ \square \ b_1 \rightarrow PC := PC + 1; Push(F, w); PC := s + 1; \\ \square \ (PC = f_0) \rightarrow Pop(F, w) \\ \square \ b_2 \rightarrow q \end{array} \right), f] \\
& \sqsubseteq \{Law\ 2.2.40\} \text{ (**if** - Guarded command introduction)} \\
& \quad w, F : [s, (PC = s) \rightarrow \{empty(F)\}; MID; PC := f], f] \\
& \sqsubseteq \{Law\ 2.2.62\} (; [b]; \{b\} \text{ simulation}) \\
& \quad w, F : [s, (PC = s) \rightarrow [empty(F)]; \{empty(F)\}; MID; PC := f; \{empty(F)\}], f] \\
& \sqsubseteq \{Law\ 2.2.62\} ([b]; \{b\} = [b]) \text{ and } \{Law\ 2.2.64\} (; := \text{commute assertion}) \\
& \quad w, F : [s, (PC = s) \rightarrow MID; PC := f; [empty(F)]; \{empty(F)\}], f] \\
& \sqsubseteq \{Law\ 2.2.62\} ([b]; \{b\} = [b] \sqsupseteq \textbf{skip}) \text{ and } \{Law\ 2.2.6\} (\textbf{var elim}) \\
& \quad w : [s, (PC = s) \rightarrow MID; PC := f; [empty(F)]; \{empty(F)\}], f] \\
& \Rightarrow \{Lemma\ 4.4.1\} \text{ (Commands in normal form)} \\
& MID
\end{aligned}$$

◇

4.4.3 Nested Normal Form

At this point of the compilation process, we have a program in which every method, as much as the main command, has its body in the same format of the normal form main command. Nevertheless, the main command can have an arbitrary number of calls that remain to be eliminated.

So far, each method body, as well as the main command, has been compiled into a separate segment of code. To eliminate calls we replace them with the corresponding method body. Nevertheless, this is only an intermediate step because we discard this unnecessary increase in the target code by arranging the code segments. 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.

To illustrate the elimination of calls, consider the program described in Figure 4.17. For conciseness, in this program we assume the existence of just one method in L .

```

CdsRVM, cds,
class  $L$ 
  meth  $lm \stackrel{\wedge}{=} (\text{val } S_{in} : \text{Stack} \text{ res } S_{out} : \text{Stack} \bullet$ 
     $v : [s_m, GCS_m, f_m] \text{ end})$ 
end
 $\bullet$ 
 $w : [s_c, \left( \begin{array}{l} b_1 \rightarrow o.lm(S, S); \text{ PC} := \text{PC} + 1; \\ \square \dots \\ \square b_n \rightarrow o.lm(S, S); \text{ PC} := \text{PC} + 1; \\ \square b \rightarrow q \end{array} \right), f_c]$ 

```

Figure 4.17: Intermediate program before the elimination of method calls

In order to eliminate the calls to the method in L we use Laws 2.2.86 and 2.2.77 to copy the body of each method to the main command and eliminate L , producing an equivalent program with nested normal form commands. Also, other laws are necessary to eliminate parametrised commands that might arise when Law 2.2.86 is applied. To illustrate these transformations, we present in Figure 4.18 the program obtained from the one in Figure 4.17.

```

CdsRVM, cds  $\bullet$ 
 $w : [s_c, \left( \begin{array}{l} b_1 \rightarrow v : [s_m, GCS_m, f_m]; \text{ PC} := \text{PC} + 1; \\ \square \dots \\ \square b_n \rightarrow v : [s_m, GCS_m, f_m]; \text{ PC} := \text{PC} + 1; \\ \square b \rightarrow q \end{array} \right), f_c]$ 

```

Figure 4.18: Program with nested normal form main commands

The next rule formalises the elimination of nested normal form commands, allowing us to keep just one copy of the code of a method body. Whenever a body of a method is to be executed, the corresponding return address is saved in a fresh variable, say w (or it can be pushed onto the Frame stack F), and the start address of the code of the method is assigned to PC . When the method execution completes, the return address is copied back into PC .

Rule 4.4.16 (Nested normal form)

If w does not occur free in the left-hand side of the following inequation and r_i is not changed by $v : [s_0, b_0 \rightarrow p, f_0]$ then

$$v : [s, \left(\begin{array}{l} b_1 \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \text{ } PC := r_1) \\ \square \quad \dots \\ \square \quad b_n \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \text{ } PC := r_n) \\ \square \quad b \rightarrow q \end{array} \right), f] \sqsubseteq v, w : [s, T, f]$$

where

$$T = \left(\begin{array}{l} b_1 \rightarrow (w := r_1; \text{ } PC := K_{b_0}) \\ \square \quad \dots \\ \square \quad b_n \rightarrow (w := r_n; \text{ } PC := K_{b_0}) \\ \square \quad f_o \rightarrow PC := w \\ \square \quad b_o \rightarrow p \square b \rightarrow q \end{array} \right)$$

□

The address where the method begins is represented by K_{b_0}

4.4.4 Our example

In Figure 4.19, 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 a sequence or guarded commands corresponding whose guards indicate its location in the bytecode stream.

In Figure 4.20, we describe the sequence of guard that form the main command segment.

The segment associated to the method *lgetLength* is presented in Figure 4.21.

4.4.5 Proof Theorem 4.4.1

The first step consists in the application of Rules 4.4.1 — 4.4.14, together with Law 2.2.4 to produce commands in form of a flat iteration (Definition 4.4.4). The form of some rules allows an infinite number of applications, as the *RHS* contains the *LHS*, but each of these rules is applied only once for each source command. The second step is the elimination of class L . To achieve this objective, Rule 4.4.15 is applied to eliminate those recursive commands that implement recursive methods. After that, calls to methods in L can be eliminated, using Law 2.2.86. Once all calls are eliminated, we can discard the class L using Law 2.2.77. To eliminate the parametrised command introduced by the application of Law 2.2.86, we use the combination of Laws 2.2.18 and 2.2.19. Finally, Rule 4.4.16 is used to eliminate the nested normal form commands and avoid the code replication. ◇

```

 $cds_{RVM} \quad cds \bullet \quad \text{var } PC : Int; \quad S, F : Stack, w : T \quad \bullet$ 
 $S := \text{new } Stack; \quad F := \text{new } STack;$ 
 $PC := s;$ 
while  $PC \geq s \wedge PC < f \rightarrow$ 
  if  $PC = s \rightarrow PC := k_5$ 
     $\square PC = k_1 \rightarrow lsetDirection$ 
     $\vdots$ 
     $\square PC = k_2 \rightarrow lsetLength$ 
     $\vdots$ 
     $\square PC = k_3 \rightarrow laddStep$ 
     $\vdots$ 
     $\square PC = k_4 \rightarrow lgetlength$ 
     $\vdots$ 
     $\square PC = k_5 \rightarrow \text{Main Command}$ 
  fi
end
end

```

Figure 4.19: General structure of our example after the Control Elimination phase

```

 $\vdots$ 
 $\square PC = k_5 \rightarrow iLoad_{CE}(north)$ 
 $\square PC = k_5 + 02 \rightarrow iLoad_{CE}(l_1)$ 
 $\square PC = k_5 + 04 \rightarrow Load_{CE}(p)$ 
 $\square PC = k_5 + 06 \rightarrow PC := PC + 2; \quad Push_i(F, PC); \quad PC := k_3 \% \{invoke \quad laadStep\}$ 
 $\square PC = k_5 + 08 \rightarrow Store_{CE}(p)$ 
 $\square PC = k_5 + 10 \rightarrow iLoad_{CE}(east)$ 
 $\square PC = k_5 + 12 \rightarrow iLoad_{CE}(l_2)$ 
 $\square PC = k_5 + 14 \rightarrow Load_{CE}(p)$ 
 $\square PC = k_5 + 16 \rightarrow PC := PC + 2; \quad Push_i(F, PC); \quad PC := k_3 \% \{invoke \quad laadStep\}$ 
 $\square PC = k_5 + 18 \rightarrow Store_{CE}(p)$ 
 $\square PC = k_5 + 20 \rightarrow iLoad_{CE}(south)$ 
 $\square PC = k_5 + 22 \rightarrow iLoad_{CE}(l_3)$ 
 $\square PC = k_5 + 24 \rightarrow Load_{CE}(p)$ 
 $\square PC = k_5 + 26 \rightarrow PC := PC + 2; \quad Push_i(F, PC); \quad PC := k_3 \% \{invoke \quad laadStep\}$ 
 $\square PC = k_5 + 28 \rightarrow Store_{CE}(p)$ 
 $\square PC = k_5 + 30 \rightarrow iLoad_{CE}(west)$ 
 $\square PC = k_5 + 32 \rightarrow iLoad_{CE}(l_4)$ 
 $\square PC = k_5 + 34 \rightarrow Load_{CE}(p)$ 
 $\square PC = k_5 + 36 \rightarrow PC := PC + 2; \quad Push_i(F, PC); \quad PC := k_3 \% \{invoke \quad laadStep\}$ 
 $\square PC = k_5 + 38 \rightarrow PC := PC + 2; \quad Push_i(F, PC); \quad PC := k_4 \% \{invoke \quad lgetLength\}$ 
 $\square PC = k_5 + 40 \rightarrow Store_{CE}(p)$ 
 $\square PC = k_5 + 42 \rightarrow iStore_{CE}(out)$ 

```

Figure 4.20: Sequence of guarded commands corresponding to the Main Command


```

⋮
□  $PC = k_4$  →  $Store_{CE}(o_4)$ 
□  $PC = k_4 + 02$  →  $Load_{CE}(o_4)$ 
□  $PC = k_4 + 04$  →  $IsOf_{CE}(Path)$ 
□  $PC = k_4 + 06$  →  $bStore_{CE}(x_1)$ 
□  $PC = k_4 + 08$  →  $Load_{CE}(o_4)$ 
□  $PC = k_4 + 10$  →  $Putfield_{CE}(Path.previous)$ 
□  $PC = k_4 + 12$  →  $aconstrnull$ 
□  $PC = k_4 + 14$  →  $cmpeq$ 
□  $PC = k_4 + 16$  →  $bStore_{CE}(x_2)$ 
□  $PC = k_4 + 18$  →  $bLoad_{CE}(x_1)$ 
□  $PC = k_4 + 20$  → if  $x_1 \rightarrow PC := PC + 2$  □  $\neg(x_1) \rightarrow PC := k_4 + 84$  fi
□  $PC = k_4 + 22$  →  $bLoad_{CE}(x_2)$ 
□  $PC = k_4 + 24$  → if  $x_1 \rightarrow PC := PC + 2$  □  $\neg(x_1) \rightarrow PC := k_4 + 32$  fi
□  $PC = k_4 + 26$  →  $Load_{CE}(o_4)$ 
□  $PC = k_4 + 28$  →  $Getfield_{CE}(Path.previous);$ 
□  $PC = k_4 + 30$  →  $PC := 68$ 
□  $PC = k_4 + 32$  →  $iLoad_{CE}(0)$ 
□  $PC = k_4 + 34$  →  $iStore_{CE}(aux)$ 
□  $PC = k_4 + 36$  →  $Load_{CE}(o_4);$ 
□  $PC = k_4 + 38$  →  $Getfield_{CE}(Path.previous);$ 
□  $PC = k_4 + 40$  →  $Push_i(F, aux)$ 
□  $PC = k_4 + 42$  →  $Push_i(F, x)$ 
□  $PC = k_4 + 44$  →  $Push_i(F, x_2)$ 
□  $PC = k_4 + 46$  →  $Push_i(F, x_1)$ 
□  $PC = k_4 + 48$  →  $Push(F, o_4)$ 
□  $PC = k_4 + 50$  →  $PC := PC + 2; Push_i(F, PC); PC := k_4$ 
□  $PC = k_4 + 52$  →  $Pop(F, o_4)$ 
□  $PC = k_4 + 54$  →  $Pop_i(F, x_1)$ 
□  $PC = k_4 + 56$  →  $Pop_i(F, x_2)$ 
□  $PC = k_4 + 58$  →  $Pop_i(F, x)$ 
□  $PC = k_4 + 60$  →  $Pop_i(F, aux)$ 
□  $PC = k_4 + 62$  →  $Putfield_{CE}(Path.previous)$ 
□  $PC = k_4 + 64$  →  $Store_{CE}(o_4);$ 
□  $PC = k_4 + 66$  →  $iStore_{CE}(aux);$ 
□  $PC = k_4 + 68$  →  $Load_{CE}(o_4);$ 
□  $PC = k_4 + 70$  →  $Getfield_{CE}(Path.previous);$ 
□  $PC = k_4 + 70$  →  $iStore_{CE}(x);$ 
□  $PC = k_4 + 74$  →  $iLoad_{CE}(x);$ 
□  $PC = k_4 + 76$  →  $iLoad_{CE}(x);$ 
□  $PC = k_4 + 78$  →  $add;$ 
□  $PC = k_4 + 80$  →  $iStore_{CE}(x);$ 
□  $PC = k_4 + 82$  →  $PC := 88$ 
□  $PC = k_4 + 84$  →  $Load_{CE}(o_4);$ 
□  $PC = k_4 + 86$  →  $Getfield_{CE}(Step.len);$ 
□  $PC = k_4 + 88$  →  $iLoad_{CE}(x);$ 
□  $PC = k_4 + 90$  →  $Load_{CE}(o_4);$ 
□  $PC = k_4 + 92$  →  $Pop(F, PC) \% \{Return\}$ 

```

Figure 4.21: Sequence of guarded commands corresponding to the Method *lgetLength*

4.5 Data Refinement

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

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

Theorem 4.5.1 (Data Refinement) *Consider a program of the form $cds_{RVM}, cds \bullet r$, where cds contains only public attribute declarations, and r has the form of Definition 4.4.4. Then*

$$\hat{\Psi}_w (cds_{RVM}, cds \bullet r) \sqsubseteq cds_{RVM} \bullet I$$

□

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 is required to link the data model used by the *RVM* with the object model of the source program. To do so, symbol tables are necessary to relate the concrete (*RVM*) and the abstract (source program) data spaces.

To carry out the change of data representation, we use the distributivity properties of the function $\hat{\Psi}$ presented later. It is a polymorphic function (built from the symbol tables) that applies to programs and commands, and distributes over the commands in the main command.

To perform the necessary change of data representation, the simulation function $\hat{\Psi}$ *reclassifies* [33, 89] the objects stored in the variables referenced by the main command. Reclassification enables an object to change its class membership, preserving its identity, by creating a concrete representation of its attribute values.

Consider the classes declared in Figure 4.24. Given a variable x whose static type is B , after the execution of the following code fragment

```
 $x := \mathbf{new} \ B;$ 
 $x.k := 50; \ x.l := 26$ 
 $x.z := \mathbf{new} \ A; \ x.z.k := 5$ 
```

a fresh object of type B is assigned to x , and its attributes are initialised. The resulting object stored in x is presented in Figure 4.22,

Figure 4.22: An object in the abstract space

In Figure 4.23 we present the abstract instance of the object in x . Observe that 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 3.1, the class *Data* specialises into two subclasses: *PriData* and *ObjData*. Object are stored in *objData* instances whereas primitive values are instances of *PriData*. The class *PriData* specialises into 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 the 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* consists of a sequence of *Data*, where each element is the concrete representation of an attribute whose type is 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 . Figure 4.23 is the concrete representation of the object stored in x .

Differently from [33] and [89], where an object may attain many types according to trees of class hierarchies, the object reclassification presented in our compilation process has the following characteristics: all objects in the concrete space are instances of the class *Data* or of its subclasses;

Figure 4.23: An object in the concrete space

besides, abstract instances do not coexist with concrete instances, except during the moment in which the 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. Thus, 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, because their purpose is just to allow the mapping between the abstract and concrete space. We introduce them merely to simplify the description of our Data Refinement phase.

To illustrate how these methods are structured, consider the class declarations in Figure 4.24.

Each class has its specific *encode* method specialised in building the abstract instance from the corresponding concrete instance. Every *encode* method has an element of class *ObjData* as a formal value parameter (D). An object of *objData* has the attribute *att*, which is a sequence of *Data*, where each element is a concrete instance of an associated attribute in the abstract space. The length of this sequence is exactly the number of attributes of an object whose type is indicated by the attribute *cl* of each *objData* instance. Hence, there is a one to one correspondence between them. Based on *att* and *cl*, the encode method retrieves the values of each attribute of an object in the abstract space.

where 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 2.2.84 (method elimination).

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 the example of Figure 4.24, the value of the attribute k declared in the class A is retrieved by calling the *encode* method declared in class A , using the clause **super**. The term $D.att[\Phi_{B,l}]$ holds the value of the attribute l , declared in class B . In case of this primitive attribute, its value is encapsulated in an instance of *IntData*.

Since z is a non-primitive attribute, there are two possible types for z : it can be an instance of class A or a instance of class B . Further, 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 the $D.att[\Phi_{B,z}]$ is different from **null**, the call **self.z.encode**($D.att[\Phi_{B,z}]$) fulfills 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 4.24 have as a result formal parameter D , which is an instance of *ObjData*. The attribute *cl* is assigned according to the current type of the target object in which the *decode* call is executed. 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 $D.att[\Phi_{B,l}]$.

```

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

```

Figure 4.24: An example of *encode* and *decode* methods

In the case of 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, we invoke the *decode* method associated with the type of this attribute 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, through the use of the clause **super**. When this call completes, the resulting sequence ($D_s.att$) is concatenated with the current sequence ($D.att$).

The next lemma is useful to establish 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 compilation 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 4.5.1 (Encode; Decode)

$$(1) \ o.encode(M[\Psi_o]); \ o.decode(M[\Psi_o]) = o.encode(M[\Psi_o])$$

$$(2) \ o.decode(M[\Psi_o]); \ o.encode(M[\Psi_o]) = o.decode(M[\Psi_o])$$

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.

In our approach to Data Refinement, the commands **dvar** and **dend** play an essential role in defining 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 used by Back [5, 8]. The *encode* and *decode* methods are used to convert objects between the concrete and the abstract data space. The usage of the *decoding* and the *encoding* blocks is more general. 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 with 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 *decode* method is called to build its concrete instance.

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 4.5.1 (Encoding block $\hat{\Psi}_{o,i}$)

```

 $[M[\Psi_o].cl = \Phi_C]$ 
dvar  $o : C; \ i : Int;$ 
  if  $M[\Psi_o] = \mathbf{null} \rightarrow o := \mathbf{null}$ 
     $\square \neg(M[\Psi_o] = \mathbf{null}) \rightarrow o := \mathbf{new} \ C; \ o.encode(M[\Psi_o])$ 
  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 $[M[\Psi_o].cl = 'C']$ prevents us from using conditionals to test the type of the variable o . Otherwise, we would have to consider the whole context of classes in which o is declared.

The next *decoding* block retrieves the concrete space from the abstract space. Similarly, once the concrete space is initialised, the data space of the source program is terminated.

Definition 4.5.2 (Decoding Block $\hat{\Psi}_{o,i}^{-1}$)

```

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

```

□

Analysing the *encoding* and *decoding* blocks, it is clear that there is an asymmetry between the definitions of the *encode* and *decode* methods. The *encode* deals only with the attributes, whereas the *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 pair $(\hat{\Psi}, \hat{\Psi}^{-1})$ is a *simulation*. The next theorem formalises the relationship between $\hat{\Psi}$ and $\hat{\Psi}^{-1}$.

Theorem 4.5.2 ($(\hat{\Psi}, \hat{\Psi}^{-1})$ simulation)

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

□

Like in [53] and in [87], we use the first component of a simulation as a function.

Definition 4.5.3 (Simulation Function)

Let $(\hat{\Psi}_w, \hat{\Psi}_w^{-1})$ be a simulation. We use $\hat{\Psi}$ itself as a function defined by

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

□

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

Definition 4.5.4 (Simulation as substitution)

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

□

The function $\hat{\Psi}_w$ does not affect the classes used to define our interpreter (cds_{RVM}), nor the components of our target machine, nor commands that have no references to variables or classes affected by $\hat{\Psi}_w$.

In an analogous way, any command that has no reference to any class or variable declared in the source program is not affected. For instance, the **skip** command is not affected.

Rule 4.5.1 (**skip** - Data Refinement)

$$\hat{\Psi}(\mathbf{skip}) \sqsubseteq \mathbf{skip}$$

□

The following two Rules address the distributivity properties of $\hat{\Psi}$ over an assignment.

Rule 4.5.2 (*Assignment- $\hat{\Psi}_{x,w}$ - Data Refinement*)

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

□

In the above case, x is replaced with the corresponding machine location $M[\Psi_x]$.

Proof:

$$\begin{aligned} & \hat{\Psi}_{x,w} (x := e) \\ = & \{ \text{Definition 4.5.3} \} \text{ (Simulation Function), and} \\ & \{ \text{Law 2.2.21} \} \text{ (dvar; dend change scope)} \\ & \text{dvar } x, w : T_x, T_w; \ x := M[\Psi_x]; \ w := M[\Psi_w]; \ x := e; \ \text{dend } M; \ \hat{\Psi}_{x,w}^{-1} \\ = & \{ \text{Law 2.2.53} \} (; := \text{combination}), \\ & \{ \text{Law 2.2.2} \} (; := \text{identity}), \text{ and} \\ & \{ \text{Definition 4.5.4} \} \text{ (Simulation as substitution)} \\ & \text{dvar } x, w : T_x, T_w; \ x := \hat{\Psi}_{x,w} (e); \ w := M[\Psi_w]; \ \text{dend } M; \ \hat{\Psi}_{x,w}^{-1} \\ \sqsubseteq & \{ \text{Definition 4.5.2} \} \text{ (Decoding Block } \hat{\Psi}_{x,w}^{-1} \text{) and} \\ & \{ \text{Law 2.2.25} \} \text{ (dend; dvar simulation)} \\ & \text{dvar } x, w : T_x, T_w; \ x := \hat{\Psi}_{x,w} (e); \ w := M[\Psi_w]; \ M[\Psi_x] := x; \ M[\Psi_w] := w; \ \text{dend } x, w \\ = & \{ \text{Law 2.2.53} \} (; := \text{combination}), \\ & \{ \text{Law 2.2.2} \} (; := \text{identity}), \text{ and} \\ & \{ \text{Law 2.2.21} \} \text{ (dvar; dend change scope)} \\ & \text{dvar } x, w : T_x, T_w; \ x := \hat{\Psi}_{x,w} (e); \ w := M[\Psi_w]; \ \text{dend } x, w; \ M[\Psi_x] := \hat{\Psi}_{x,w} (e) \\ = & \{ \text{Law 2.2.23} \} \text{ (dend- := final value) and} \\ & \{ \text{Law 2.2.25} \} \text{ (dend; dvar simulation)} \\ & M[\Psi_x] := \hat{\Psi}_{x,w} (e) \end{aligned}$$

◇

In the next Rule, the target of the assignment is not affected by the function $\hat{\Psi}$.

Rule 4.5.3 (*Assignment - $\hat{\Psi}_w$ - Data Refinement*) If x is not in w

$$\hat{\Psi}_w (x := e) \sqsubseteq x := \hat{\Psi}_w (e)$$

□

Proof: Similar to Lemma 4.5.2

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

Rule 4.5.4 (*Sequential Composition - Data Refinement*)

$$\hat{\Psi}_w (p; q) \sqsubseteq \hat{\Psi}_w (p); \hat{\Psi}_w (q)$$

□

When applied to constructors that deal with control, like the conditional, $\hat{\Psi}$ distributes over the components of these commands.

Rule 4.5.5 (Conditional - Data Refinement)

$$\begin{aligned} & \hat{\Psi}_w (\mathbf{if} \ b \ \rightarrow \ p \ \square \ \neg(b) \ \rightarrow \ q \ \mathbf{fi}) \\ \sqsubseteq & \\ & \mathbf{if} \ \hat{\Psi}_w (b) \ \rightarrow \ \hat{\Psi}_w (p) \ \square \ \hat{\Psi}_w (\neg(b)) \ \rightarrow \ \hat{\Psi}_w (q) \ \mathbf{fi} \end{aligned}$$

where b is a boolean expression. □

The next rules deal with abbreviations introduced in the previous compilation phase. The following rule asserts the effect of $\hat{\Psi}$ over the abbreviation $Load_{CE}(x)$.

Rule 4.5.6 (Load variable - Data Refinement)

$$\hat{\Psi}_w (Load_{CE}(x)) \sqsubseteq Load(\Psi_x)$$

□

Recall that the definition of $Load_{CE}(x)$ is

$$S.s := \langle x \rangle \wedge S.s; \ PC := PC + 2$$

whereas the definition of the instruction $Load(\Psi_x)$ is as follows.

$$S.s := \langle M[\Psi_x] \rangle \wedge S.s; \ PC := PC + 2$$

where the index Ψ_x refers to the location that holds the value of x in M . While the $Load_{CE}(x)$ pushes an abstract instance onto S , the instruction $Load(\Psi_x)$ pushes the concrete counterpart.

The next rule shows the effect of $\hat{\Psi}$ over an abbreviation $Store(x)$.

Rule 4.5.7 (Store variable - Data Refinement)

$$\hat{\Psi}_w (Store_{CE}(x)) \sqsubseteq Store(\Psi_x)$$

□

The following rule addresses the abbreviation used to push a constant onto the operand stack.

Rule 4.5.8 (Load Constant - Data Refinement)

$$\hat{\Psi}_w (Load_{CE}(a)) \sqsubseteq Ldc(\Phi_a)$$

□

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

The rule below shows how we deal with object creation.

Rule 4.5.9 (Object Creation - Data Refinement)

$$\hat{\Psi}_w (Load_{CE}(\mathbf{new} \ C)) \quad \sqsubseteq \quad new(\Phi_C)$$

□

After the simplification phase, each object creation is refined to an abbreviation like $Load(\mathbf{new} \ C)$. It has as parameter an expression that retains a reference to C , a class declared in the source program. The function $\hat{\Psi}$ eliminates this source reference, replacing $Load(\mathbf{new} \ C)$ with another abbreviation 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.

The following rule deals with type testing.

Rule 4.5.10 (Instanceof - Data Refinement)

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

□

The instruction $Instanceof(\Phi_C)$ pops an object from the operand stack. If this object is an instance of C or one 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 rely on the context of classes declared in the source program. In the following we present its definition.

```

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

```

where the method $Instanceof$, declared in Cls (Figure 3.5, 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 the subclass of C . The term Φ_C indicates the object ($ClsData$) that represent the class C in Cls .

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

Rule 4.5.11 (Getfield - Data Refinement)

$$\hat{\Psi}_w (Getfield(C.t)) \sqsubseteq Getfield(\Phi_{C.t});$$

□

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; the abbreviation $Getfield(\Phi_{C.t})$ gets the value corresponding to the attribute t and pushes it onto the top of the operand stack S .

The next rule deals with setting a value of an object attribute.

Rule 4.5.12 (Putfield - Data Refinement)

$$\hat{\Psi}_w (Putfield(C.t)) \sqsubseteq Putfield(\Phi_{C.t})$$

□

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

As an illustration, Figure 4.25 presents the resulting segment of guarded commands that represents the sequence of bytecode instructions of method *lgetLength*. The resulting program has the same structure of that presented in Figure 4.19. The classes and variables declared in the source program are eliminated. The program operates exclusively on the concrete space.

In Figure 4.25, we present the segment referent to the method *lgetLength*.

4.5.1 Proof Theorem 4.5.1

The application of Rules 4.5.1 — 4.5.12 is sufficient to obtain a program that operates exclusively on the concrete space of the target machine. Then we apply Law 2.2.77 to eliminate the class declarations in cds , as they are not referenced any longer. ◇

4.6 The Compilation Process

Theorem 1.5.1 asserts that we can transform $\hat{\Psi} (cds \bullet c)$ into $cds_{RVM} \bullet w : [s, GSC, f]$. This theorem links the intermediate steps and establishes the correctness for the entire compilation process.

From Theorem 4.1.1 (Class Pre-compilation), we can transform $cds \bullet c$, eliminating method redefinitions, introducing casts and changing attribute visibility to public. Performing this transformations, the program $cds_{pc} \bullet c_{pc}$ is obtained.

```

⋮
□  $PC = k_4$  →  $store(\Psi_{o_4})$ 
□  $PC = k_4 + 02$  →  $load(\Psi_{o_4})$ 
□  $PC = k_4 + 04$  →  $instanceof(\Phi_{Path})$ 
□  $PC = k_4 + 06$  →  $store(\Psi_{x_1})$ 
□  $PC = k_4 + 08$  →  $load(\Psi_{o_4})$ 
□  $PC = k_4 + 10$  →  $Putfield(\Phi_{Path.previous})$ 
□  $PC = k_4 + 12$  →  $aconstrnull$ 
□  $PC = k_4 + 14$  →  $cmpeq$ 
□  $PC = k_4 + 16$  →  $bstore(\Psi_{x_2})$ 
□  $PC = k_4 + 18$  →  $bLoad_{CE}(\Psi_{x_1})$ 
□  $PC = k_4 + 20$  →  $cgoto(k_4 + 84)$ 
□  $PC = k_4 + 22$  →  $bLoad_{CE}(x_2)$ 
□  $PC = k_4 + 24$  →  $cgoto(k_4 + 32)\mathbf{fi}$ 
□  $PC = k_4 + 26$  →  $Load_{CE}(\Psi_{o_4})$ 
□  $PC = k_4 + 28$  →  $Getfield(\Phi_{Path.previous});$ 
□  $PC = k_4 + 30$  →  $goto(68)$ 
□  $PC = k_4 + 32$  →  $ldc(\Phi_0)$ 
□  $PC = k_4 + 34$  →  $store(\Psi_{aux})$ 
□  $PC = k_4 + 36$  →  $load(\Psi_{o_4});$ 
□  $PC = k_4 + 38$  →  $getfield(\Phi_{Path.previous});$ 
□  $PC = k_4 + 40$  →  $save(\Psi_{aux})$ 
□  $PC = k_4 + 42$  →  $save(\Psi_x)$ 
□  $PC = k_4 + 44$  →  $save(\Psi_{x_2})$ 
□  $PC = k_4 + 46$  →  $save(\Psi_{x_1})$ 
□  $PC = k_4 + 48$  →  $save(\Psi_{o_4})$ 
□  $PC = k_4 + 50$  →  $invoke(k_4)$ 
□  $PC = k_4 + 52$  →  $restore(\Psi_{o_4})$ 
□  $PC = k_4 + 54$  →  $restore(\Psi_{x_1})$ 
□  $PC = k_4 + 56$  →  $restore(\Psi_{x_2})$ 
□  $PC = k_4 + 58$  →  $restore(\Psi_x)$ 
□  $PC = k_4 + 60$  →  $restore(\Psi_{aux})$ 
□  $PC = k_4 + 62$  →  $putfield(\Phi_{Path.previous})$ 
□  $PC = k_4 + 64$  →  $store(\Psi_{o_4});$ 
□  $PC = k_4 + 66$  →  $store(\Psi_{aux});$ 
□  $PC = k_4 + 68$  →  $load(\Psi_{o_4});$ 
□  $PC = k_4 + 70$  →  $getfield(\Phi_{Path.previous});$ 
□  $PC = k_4 + 70$  →  $store(\Psi_x);$ 
□  $PC = k_4 + 74$  →  $load(\Psi_x);$ 
□  $PC = k_4 + 76$  →  $load(Psi_x);$ 
□  $PC = k_4 + 78$  →  $add;$ 
□  $PC = k_4 + 80$  →  $store(\Psi_x);$ 
□  $PC = k_4 + 82$  →  $goto(88)$ 
□  $PC = k_4 + 84$  →  $load(\Psi_{o_4});$ 
□  $PC = k_4 + 86$  →  $getfield(\Phi_{Step.len});$ 
□  $PC = k_4 + 88$  →  $load(\Psi_x);$ 
□  $PC = k_4 + 90$  →  $load(\Psi_{o_4});$ 
□  $PC = k_4 + 92$  →  $Pop(F, PC) \% \{Return\}$ 

```

Figure 4.25: Sequence of bytecode instructions corresponding to the Method *lgetLength*

$\vdots$
 $\square PC = k_5 \rightarrow ldc(\Phi_{north})$
 $\square PC = k_5 + 02 \rightarrow load(\Psi_{l_1})$
 $\square PC = k_5 + 04 \rightarrow load(\Psi_p)$
 $\square PC = k_5 + 06 \rightarrow invoke(k_3) \% \{invoke\ laadStep\}$
 $\square PC = k_5 + 08 \rightarrow store(\Psi_p)$
 $\square PC = k_5 + 10 \rightarrow ldc(\Phi_{east})$
 $\square PC = k_5 + 12 \rightarrow load(\Psi_{l_2})$
 $\square PC = k_5 + 14 \rightarrow load(\Phi_p)$
 $\square PC = k_5 + 16 \rightarrow invoke(k_3) \% \{invoke\ laadStep\}$
 $\square PC = k_5 + 18 \rightarrow store(\Psi_p)$
 $\square PC = k_5 + 20 \rightarrow ldc(\Phi_{south})$
 $\square PC = k_5 + 22 \rightarrow load(\Psi_{l_3})$
 $\square PC = k_5 + 24 \rightarrow load(\Psi_p)$
 $\square PC = k_5 + 26 \rightarrow invoke(k_3) \% \{invoke\ laadStep\}$
 $\square PC = k_5 + 28 \rightarrow store(\Psi_p)$
 $\square PC = k_5 + 30 \rightarrow ldc(\Phi_{west})$
 $\square PC = k_5 + 32 \rightarrow load(\Psi_{l_4})$
 $\square PC = k_5 + 34 \rightarrow load(\Psi_p)$
 $\square PC = k_5 + 36 \rightarrow invoke(k_3) \% \{invoke\ laadStep\}$
 $\square PC = k_5 + 38 \rightarrow invoke(k_4) \% \{invoke\ lgetLength\}$
 $\square PC = k_5 + 40 \rightarrow store(\Psi_p)$
 $\square PC = k_5 + 42 \rightarrow store(\Psi_{out})$

Figure 4.26: Main command after the Data Refinement

Theorem 4.2.1 (Redirection of method calls) establishes that

$$cds_{pc} \bullet c_{pc} \sqsubseteq cds_{RVM}, cds_{rmc} \bullet crmc$$

At this point, the class L together with cds_{RVM} (the class declarations of our 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 4.3.1 (Simplification of Expressions and some source transformations), which states that we can obtain the program $cds_{RVM}, cds, L_s \bullet c_s$ from the program $cds_{RVM}, cds_{rmc} L; \bullet crmc$, which is the outcome of the second compilation phase. 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 is no parametrised command and every conditional is written as *if-then-else*.

At this point, based on from Theorem 4.4.1 (Control elimination), we transform the nested control structure of the program $cds_{RVM}, cds, L_s \bullet c_s$ into a single flat iteration, obtaining the program $cds_{RVM}, cds_{rmc} \bullet v : [s, GSC, f]$, which is a program with the same control structure of our normal form, but still operating on the abstract data space.

Finally, from Theorem 4.5.1 (Data Refinement), using the distributive properties of the simulation function $\hat{\Psi}$, built from the symbol tables, 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 is a normal form program $cds_{RVM} \bullet I$

Proof Theorem 1.5.1:

$$\begin{aligned}
& \hat{\Psi} (cds \bullet c) \\
\sqsubseteq & \{Theorem\ 4.1.1\} \text{ (Class Pre-compilation)} \\
& \hat{\Psi} (cds_{pc} \bullet c_{pc}) \\
\sqsubseteq & \{Theorem\ 4.2.1\} \text{ (Redirection of method calls)} \\
& \hat{\Psi} (cds_{RVM}, cds_{rmc} \bullet crmc) \\
\sqsubseteq & \{Theorem\ 4.3.1\} \text{ (Simplification of Expressions and some source transformations)} \\
& cds_{RVM}, cds, L_s \bullet c_s \\
\sqsubseteq & \{Theorem\ 4.4.1\} \text{ (Control Elimination)} \\
& \hat{\Psi} (cds_{RVM}, cds_{rmc} \bullet v : [s, GSC, f]) \\
\sqsubseteq & \{Theorem\ 4.5.1\} \text{ (Data Refinement)} \\
& cds_{RVM} \bullet I
\end{aligned}$$

◇

This complete algebraic formalisation of the compilation process for an object-oriented language is the main result of this work.

Chapter 5

Final Considerations

A systematic development of correct compilers is still an active and difficult research topic. Due to the intricacy of the task, many approaches focused on the development of mathematical models to ensure the consistency of the compilation process. In order to preserve behaviour the translation from source to target code must be carried out in a disciplined way. Using formal methods as a design tool the desired discipline can be imposed.

Formal methods approaches are twofold: either based on verification, which requires the target code to be proved to correct given the semantics of the source program, or based on construction, where the source program is transformed into another by the application of laws.

In this thesis we adopted a constructive approach to compilation based on algebraic laws. These laws are suitable not only for compilation but they are expressive enough to formally derive behaviour preserving program transformations, as illustrated in [13] through the derivation of provably-correct refactorings.

Here (and in [36, 34, 35]) we presented a set of compilation rules along with their derivation 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 [19]. 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 specification statements of Morgan's refinement calculus [68]. Differently from most practical object-oriented languages, which are based on reference semantics, ROOL adopts a copy semantics. So we do not model references or sharing.

Nevertheless, ROOL is a programming language sufficiently similar to Java to be used in meaningful case studies. For instance, its simplified semantics still allows us to reason about the subset of Java programs that do not have reference aliasing. Study of this language seem to represent a significant advance on related work in the literature.

We use this algebraic framework to characterise a correct compiler for the executable subset of ROOL, extending the approach described in [87]. Following this algebraic approach, we addressed the correct implementation of object-oriented features. Both the source (ROOL) and the target

(a simplified model of the *JVM*, called here *RVM*) languages adopted here are far more complex than those described in [87].

Each transformation performed by the compilation rules brings the source program closer to our particular normal form. The product of the compilation process is just a program in the same language (the normal form) from which we can capture the sequence of generated bytecodes of the target machine.

By yielding a detailed compilation of an example, we illustrated how each compilation phase is carried out, showing how each algebraic transformation corroborates the generation of a sequence of bytecodes for our target machine (*RVM*). This work can also be taken as a realistic case study that describes how a rich set of provable laws for object-oriented programming can be used to compile a executable subset of ROOL.

Due to the dynamic binding of methods, reasoning about object-oriented languages becomes a major challenge. Dynamic binding is handled with the elimination of method redefinitions, based on the reduction strategy presented in [13]. In the first compilation phase, we apply the laws of classes to move up the declarations of redefined methods, so that all method redefinitions are eliminated. Therefore, we no longer needed to worry about the dynamic binding of methods in the rest of the compilation process.

In order to reduce the source program to the normal form, the classes declared in the source program have to be eliminated. To remove them, our strategy relies on the introduction of an auxiliary class *L*, which includes a concrete representation of all the source methods in terms of the elements of the ROOL virtual machine. A call to a source method is replaced with a call to the corresponding method in *L*. Based on this fact, we adapted the reduction rules devised to perform procedure elimination, described in [87].

One advantage of this semantic style is abstraction. In comparison with the other approaches to semantics, the algebraic approach does not require the construction of an explicit mathematical model. The semantics of the language is characterised by its laws (equations and inequations). As a consequence, extensions and modifications to the semantics imply the alteration of only those laws whose elements were affected by the changes. In case of denotational semantics [71, 81], the impact would require alterations on its mathematical model and, therefore, revision of every semantic clause. In the case of operational semantics [76], an extension of the language would cause a severe revision of the proofs, given that proofs are normally constructed by structural induction or by induction on the length of derivations or on the depth of trees. Nevertheless, we are not advocating the superiority of the algebraic approach over the other approaches to semantics. Different semantic approaches are suitable for different purposes. For instance, some programming language concepts, such as nondeterminism, are defined more easily in one approach than in another. Therefore, each formalism has its appropriate area of application.

One common criticism to the algebraic style is that merely postulating algebraic laws can give rise to complex and unexpected interactions between programming constructions; this can be

avoided by linking the algebraic semantics with a mathematical model in which the laws can be verified. Nevertheless, **dvar** and **dend** are not defined in the weakest precondition semantics of ROOL. In this case, we adopted the algebraic semantics defined in [87].

The semantics of ROOL is given by means of predicate transformers. We presented laws that deal with imperative features and object-oriented features. The correctness of these laws is justified by the weakest precondition calculus. The majority of these laws have been proved correct with respect to the weakest precondition semantics for the language and simulation laws. These proofs can be found in [19, 14, 12, 24].

We believe our work is a modest contribution to the field of compiler design and correctness, extending the approach presented in [87] to compile an object-oriented language based on sequential Java into a sequence of bytecodes for a virtual machine. Nevertheless, there is much to be done to generalise this approach to even more complex source and target languages. In the next section we discuss related work. In Section 5.2 we finish with a critical analysis and indicate some ways in which our work can be extended.

5.1 Related Work

In the literature, one can find several approaches related to the design of correct compilers. The majority deals with procedural languages, as the algebraic approach described in [72]. In the next sections we briefly present related works within the formalisation of Java and the *JVM*, and other approaches that characterise compilation as program refinement.

5.1.1 Approaches for object-oriented languages

There are many approaches concerning the formalisation of Java and the *JVM*. Nevertheless, these approaches are based on verification instead of calculation. In [15], using Abstract State Machines (*ASM*), a compilation scheme of Java programs to *JVM* code is presented and the work turns out to be a case study for mechanical verification of a compiler correctness proof.

A large sequential sublanguage of Java is described in [76]. Its abstract syntax, type system, well-formedness conditions, and an operational evaluation semantics are formalised as *ASM* models. Based on this formalization, type soundness is expressed and proved. Using the theorem prover Isabelle/HOL all definitions and proofs are formally verified.

Also, based on *ASM*, a structured sequence of mathematical models for static and dynamic behaviour of the programming language Java is defined in [90]. These definitions help to clarify and explore the official description [44, 60]. The Java and *JVM* are decomposed into language layers and security modules, dividing the overall definition and the verification problem into a series of tractable subproblems. This 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. These *ASM* models are used to establish under what assumptions Java programs can

be proved type safe, and successfully verified, and correctly executed when compiled to *JVM* code. Moreover, these models are executable with AsmGofer [88], which allows one to execute an *ASM* step by step, giving the opportunity to observe the current state in each step. Thus, this work is highly indicated to support the appropriate understanding of Java programs and of what can be expected when these programs are running on the *JVM*.

Compilation as refinement of programs

Closely related to Sampaio’s approach to compilation [87] is the work reported in [59], which presents a formal compilation model for real-time programs. As we do in this thesis, the program compilation is also characterised as sequence of equivalence-preserving transformations. The source language for this compilation formalism consists of a small, imperative programming language with special real-time statements, whereas the target language for the compilation process 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 [46, 47].

More recently, Wildman [96] presented a case study program compilation verification from imperative programs to assembly code. A compilation strategy is presented as a data refinement. The compiled code is data refined by calculation, as we do here.

Another work focused on using refinement as a compilation model is presented in [95]. In this work the conventional framework of program refinement is extended down to the assembler level. Compilation is regarded as a phase in the total 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 [1]), which is part of the .NET framework [54, 82]. Unfortunately, this case study does not comprise object-oriented features, and we are not aware of any other work, based on calculation, in this direction.

5.2 Future Work

Our work can be extended in several directions. Further work is needed towards the mechanisation of this approach. Its algebraic nature makes the mechanisation easier, allowing the use of a term rewrite system as a tool for specification, verification, and prototype implementation. Another issue for investigation is the extension of our source and/or target languages with more complex constructors. The investigation of optimising transformations which could be applied both to the source and to the target code would be an important complement to this work.

5.2.1 Mechanisation

Although it is not in the scope of our work, it is relatively easy to implement a prototype compiler by coding the rules in a term rewriting system such as OBJ [87]. As our theorems have the form of rewrite rules, we could use a term rewrite system both to verify the proof of our compilation rules and to carry out compilation automatically. In this way, a prototype compiler can be obtained as a by-product of its own proof of correctness. Moreover, the mechanisation of our approach to compilation would yield a formal specification of all concepts involved in it.

The work of Meseguer [23, 21, 66, 22] provides support to inequational reasoning, which is essential for the implementation of our prototype; the reason is that, in general, our rules are expressed as refinement, rather than equality. In this framework, a logic called *rewriting logic* and a language called Maude, which is based on this logic, are described. Maude provides additional features to support concurrency and object orientation.

The mechanisation of our approach would bring a better insight into our hand proofs, because many aspects are often left out in an informal presentation. For example, in a manual proof, a transformation is usually justified by citing the necessary laws; the details of how the law is actually used to perform the transformation are often neglected. The mechanisation would help to uncover (possible) hidden assumptions and the omission of references to laws necessary to justify some proof steps.

Our algebraic system is both non-terminating and non-confluent. An example of a non-terminating rule is Rule 4.3.11; further, if the reductions are not applied in a given order, it might lead a given input program to different normal forms. For instance, a permutation in the order of the compilation phases may require us to repeat a phase already accomplished, and this would result in different outcomes. Therefore the proof steps cannot be carried out completely automatically; the application of the laws requires our guidance. Even so, we believe that the automated proofs would be safer and (hopefully) less laborious than their manual versions in that it frees the user from handwriting long terms resulting from the application of laws.

5.2.2 Code optimisation

In terms of code optimisation, we just provide the rules related to the elimination of irrelevant sequence 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 in the context of our target machine is the replacement of sequences of bytecode instructions with other sequences known to execute more efficiently.

5.2.3 Language extensions

An immediate topic for further research is the extension of our reduction strategy to target a similar language that has a reference instead of copy semantics. Some of the algebraic laws related to imperative features rely on copy semantics and, therefore, must be revised in the context of a reference semantics. On the other hand, laws for the object-oriented of ROOL do not rely on copy semantics.

Dealing with pointers will have an impact on our virtual machine and on the target language. In such a scenario, two memory locations share the same object if they store the same pointer to the location where this object is stored in the heap. This certainly implies in a better use of memory resources, because fresh objects will not be created. In the context of a reference semantics, the bytecode instructions are also affected. For instance, consider the instruction *Putfield*. In our *RVM*, after the execution of this instruction, the resulting operand stack has the modified object at the top. Typically, this object is stored in some memory location, to make effective the intended attribute assignment. In a reference semantics context, after the execution of a *Putfield* instruction, we no longer need to have the affected object at the top of operand stack because when an object is affected, all variables that share that object reference the same modified object.

A further topic of investigation resides in the extension of our source language with more complex structures. For example, we can extend ROOL to comprise a larger subset of Java, combining both object-orientation and concurrency aspects.

A remarkable example that handles parallelism, communication and external choice is in [48], where 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 the process of eliminating concurrency is entirely described in terms of our source language, a low level implementation can be obtained because this program can be reduced to normal form.

Appendix A

Proofs of Compilation Rules for ROOL

In this appendix, we present the proofs of the compilation rules presented in Chapter 4. The proofs of related lemmas and theorems are also presented here.

A.1 Lemmas

The lemmas presented here are referenced only by the proofs in the next sections.

The next lemma establishes that a fresh object of type *Stack* is always *empty*.

Lemma A.1.1 (Initial empty stack)

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

$$\textit{where empty}(S) \stackrel{\textit{def}}{=} (S.s = \langle \rangle)$$

Proof :

$$\begin{aligned} & S := \mathbf{new} \textit{Stack} \\ = & \text{From definition of Class } \textit{Stack} \text{ (Figure 3.2)} \\ & S := \mathbf{new} \textit{Stack} = S := \mathbf{new} \textit{Stack}; \{ \textit{empty}(S) \} \end{aligned}$$

◇

The following lemma states that the assignment does not affect the resulting stack.

Lemma A.1.2 (Assignment does not affect the resulting stack)

$$\{ S = X \}; S.\textit{Push}(e); S.\textit{Pop}(x) \sqsubseteq \{ S = X \}; S.\textit{Push}(e); S.\textit{Pop}(x); \{ S = X \}$$

Proof:

$$\begin{aligned}
& RHS \\
= & \{Definition\ 4.2.1\} \langle \text{Methods declared in the class Stack} \rangle \\
& \{S = X\}; \ S.s := \langle e \rangle \wedge S.s; \ x := head(S.s); \ S.s := tail(S.s); \ \{S = X\} \\
\\
= & \{Law\ 2.2.54\} (; := \text{substitution}) \\
& \{S = X\}; \ S.s := \langle e \rangle \wedge S.s; \ x := e; \ S.s := tail(S.s); \ \{S = X\} \\
\\
= & \{Law\ 2.2.55\} (; := \text{commutation}) \\
& \{S = X\}; \ S.s := \langle e \rangle \wedge S.s; \ S.s := tail(S.s); \ \{S = X\}; \ x := e \\
\\
= & \{Law\ 2.2.63\} (; := \text{commute assumption}) \\
& \{S = X\}; \ \{S = X\}; \ x := e \\
\\
= & \{Law\ 2.2.59\} (\text{assumption combination}) \\
& \{S = X \wedge S = X\}; \ x := e \\
\\
= & \{Law\ 4.3.4\} \langle \text{Assignment of a variable} \rangle \\
& \{S = X\}; \ S.s := \langle e \rangle \cup S.s; \ S.s := tail(S.s); \ x := e \\
\\
= & \{Law\ 2.2.55\} (; := \text{commutation}) \\
& \{S = X\}; \ S.s := \langle e \rangle \cup S.s; \ x := e; \ S.s := tail(S.s) \\
\\
= & \{Law\ 2.2.54\} (; := \text{substitution}) \\
& \{S = X\}; \ S.s := \langle e \rangle \cup S.s; \ x := head(S.s); \ S.s := tail(S.s) \\
\\
= & \{Definition\ 4.2.1\} \langle \text{Abbreviations — operand stack} \rangle \\
& RHS
\end{aligned}$$

◇

The next lemma allows an specific permutation in a sequence of *push* and *pop*.

Lemma A.1.3 (Combining two pairs of Pushs and Pops)

$$\begin{aligned}
& Push(S, e_1); Pop(S, z); Push(S, e_2); Pop(S, y) \\
= & \\
& Push(S, e_1); Push(S, e_2); Pop(S, y); Pop(S, z)
\end{aligned}$$

Proof:

$$\begin{aligned}
& Push(S, e_1); Pop(S, z); Push(S, e_2); Pop(S, y) \\
= & \{Definition\ 4.2.1\} \langle \text{Abbreviations — operand stack} \rangle \\
& S.s := \langle e_1 \rangle \wedge S.s; z := head(S.s); S.s := tail(S.s); \\
& S.s := \langle e_2 \rangle \wedge S.s; y := e_2; S.s := tail(S.s) \\
= & \{Law\ 2.2.54\} (; := \text{substitution}) \\
& \{Law\ 2.2.63\} (; := \text{commute assumption}) \\
& S.s := \langle e_1 \rangle \wedge S.s; y := e_2; z := head(S.s); S.s := tail(S.s); \\
& S.s := \langle e_2 \rangle \wedge S.s; S.s := tail(S.s) \\
= & \{Law\ 3.4.1\} (\text{laws of sequences}) \\
& S.s := \langle e_1 \rangle \wedge S.s; y := e_2; z := head(S.s); S.s := tail(S.s); \mathbf{skip} \\
= & \{Law\ 2.2.46\} (; - \mathbf{skip} \text{ unit}) \\
& S.s := \langle e_1 \rangle \wedge S.s; \mathbf{skip}; y := e_2; z := head(S.s); S.s := tail(S.s) \\
= & \{Law\ 3.4.1\} (\text{laws of sequences}) \\
& \{Law\ 2.2.63\} (; := \text{commute assumption}) \\
& S.s := \langle e_1 \rangle \wedge S.s; S.s := \langle e_2 \rangle \wedge S.s; y := e_2; S.s := tail(S.s); \\
& z := head(S.s); S.s := tail(S.s) \\
= & \{Law\ 2.2.54\} (; := \text{substitution}) \\
& S.s := \langle e_1 \rangle \wedge S.s; S.s := \langle e_2 \rangle \wedge S.s; y := head(S.s); S.s := tail(S.s); \\
& z := head(S.s); S.s := tail(S.s) \\
= & \{Definition\ 4.2.1\} \langle \text{Abbreviations — operand stack} \rangle \\
& Push(S, e_1); Push(S, e_2); Pop(S, y); Pop(S, z)
\end{aligned}$$

◇

The next lemma consider a particular case of reduction of sequential composition. Here the guarded command set of the right argument includes that of the left argument.

Lemma A.1.4 (Sequential composition)

$$PC : [s, b_1 \rightarrow p, s_0]; PC : [s_0, \left(\begin{array}{c} b_1 \rightarrow p \\ \square \quad b_2 \rightarrow q \end{array} \right), f] \sqsubseteq PC : [s, \left(\begin{array}{c} b_1 \rightarrow p \\ \square \quad b_2 \rightarrow q \end{array} \right), f]$$

Proof:

LHS

$\{Definition 4.4.1\}$ and $\{Law\ 2.2.12\}$

```
var  $PC : Int$  •  
   $PC := s$ ;  
  while  $b_1 \rightarrow$  if  $b_1 \rightarrow p$  fi end;  
   $\{PC = s_o\}$   
   $PC := s_o$ ;  
  while  $(b_1 \vee b_2) \rightarrow$  if  $b_1 \rightarrow p \sqcap b_2 \rightarrow q$  fi end;  
   $\{PC = f\}$   
end
```

$\sqsubseteq \{Law\ 2.2.58\}, \{Law\ 2.2.62\}$

```
var  $PC : Int$  •  
   $PC := s$ ;  
  while  $b_1 \rightarrow$  if  $b_1 \rightarrow p$  fi end;  
  while  $(b_1 \vee b_2) \rightarrow$  if  $b_1 \rightarrow p \sqcap b_2 \rightarrow q$  fi end;  
   $\{PC = f\}$   
end
```

$\sqsubseteq \{Law\ 2.2.73\}$

```
var  $PC : Int$  •  
   $PC := s$ ;  
  while  $b_1 \rightarrow$  if  $b_1 \rightarrow p \sqcap b_2 \rightarrow q$  fi end;  
  while  $(b_1 \vee b_2) \rightarrow$  if  $b_1 \rightarrow p \sqcap b_2 \rightarrow q$  fi end;  
   $\{PC = f\}$   
end
```

$\sqsubseteq \{Law\ 2.2.76\}$

```
var  $PC : Int$  •  
   $PC := s$ ;  
  while  $(b_1 \vee b_2) \rightarrow$  if  $b_1 \rightarrow p \sqcap b_2 \rightarrow q$  fi end;  
   $\{PC = f\}$   
end
```

$\sqsubseteq \{Definition\ 4.4.1\}$

RHS

◇

Extending the guarded command set by introducing new guarded commands leads to refinement.

Lemma A.1.5 (*Guarded command introduction*)

$$PC : [s, b_1 \rightarrow p, f] \sqsubseteq PC : [s, \left(\begin{array}{c} b_1 \rightarrow p \\ \square \quad b_2 \rightarrow q \end{array} \right), f]$$

Proof:

$$\begin{aligned} & \text{RHS} \\ & \sqsubseteq \{ \text{Law 2.2.46} \} \text{ and } \{ \text{Rule 4.4.1} \} \\ & PC : [s, b_1 \rightarrow p, f]; \quad PC : [f, \left(\begin{array}{c} b_1 \rightarrow p \\ \square \quad b_2 \rightarrow q \end{array} \right), f] \\ & \sqsubseteq \{ \text{Lemma A.1.4} \} \\ & \text{LHS} \end{aligned}$$

◇

The next lemma is used to eliminate a conditional whose branches are reduction form programs with identical components, except for the initial value for PC .

Lemma A.1.6 (*Conditional elimination*)

If PC is not free in b then

$$\begin{aligned} & \text{if } b \rightarrow PC : [s_1, R, f] \\ & \quad \square \neg b \rightarrow PC : [s_2, R, f] \\ & \text{fi} \end{aligned} \sqsubseteq PC : [s, R, f]$$

$$\text{Where } R = \left(\begin{array}{c} PC = s \rightarrow \text{if } b \rightarrow PC := s_1 \quad \square \neg b \rightarrow PC := s_2 \text{ fi} \\ \square \quad b_1 \rightarrow p \end{array} \right)$$

Proof:

$$\begin{aligned} & \text{RHS} \\ & = \{ \text{Definition 4.4.1} \} \text{ and } \{ \text{Law 2.2.61} \} \\ & \text{var } PC : \text{Int} \bullet \\ & \quad PC := s; \quad [PC = s]; \\ & \quad \text{while } PC \geq s \wedge PC < f \rightarrow \text{if } R \text{ fi end;} \\ & \quad \{ PC = f \} \\ & \text{end} \end{aligned}$$

$= \{Law\ 2.2.71\}$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
if R **fi**
while $PC \geq s \wedge PC < f \rightarrow$ **if** R **fi end;**
 $\{PC = f\}$
end

$\sqsubseteq \{Law\ 2.2.37\}$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
if $b \rightarrow PC := s_1 \square \neg b \rightarrow PC := s_2$ **fi**
while $PC \geq s \wedge PC < f \rightarrow$ **if** R **fi end;**
 $\{PC = f\}$
end

$= \{Law\ 2.2.50\}$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
if $b \rightarrow PC := s_1;$
while $PC \geq s \wedge PC < f \rightarrow$ **if** R **fi end;**
 $\{PC = f\}$
 $\square \neg b \rightarrow PC := s_2;$
while $PC \geq s \wedge PC < f \rightarrow$ **if** R **fi end;**
 $\{PC = f\}$
fi
end

$= \{Law\ 2.2.61\} \text{ and } \{Law\ 2.2.51\}$
var $PC : Int \bullet$
if $b \rightarrow PC := s; PC := s_1;$
while $PC \geq s \wedge PC < f \rightarrow$ **if** R **fi end;**
 $\{PC = f\}$
 $\square \neg b \rightarrow PC := s; PC := s_2;$
while $PC \geq s \wedge PC < f \rightarrow$ **if** R **fi end;**
 $\{PC = f\}$
fi
end

$$\begin{aligned}
&= \{Law\ 2.2.53\} \text{ and } \{Law\ 2.2.8\} \\
&\quad \mathbf{if}\ b \rightarrow \\
&\quad \quad \mathbf{var}\ PC : Int \bullet \\
&\quad \quad \quad PC := s_1; \\
&\quad \quad \quad \mathbf{while}\ PC \geq s \wedge PC < f \rightarrow \mathbf{if}\ R \mathbf{fi}\ \mathbf{end}; \\
&\quad \quad \quad \{PC = f\} \\
&\quad \quad \mathbf{end} \\
&\quad \square \neg b \rightarrow \\
&\quad \quad \mathbf{var}\ PC : Int \bullet \\
&\quad \quad \quad PC := s_2; \\
&\quad \quad \quad \mathbf{while}\ PC \geq s \wedge PC < f \rightarrow \mathbf{if}\ R \mathbf{fi}\ \mathbf{end}; \\
&\quad \quad \quad \{PC = f\} \\
&\quad \quad \mathbf{end} \\
&\quad \mathbf{fi} \\
&= \{Definition\ 4.4.1\} \\
&\quad RHS
\end{aligned}$$

◇

The following lemma states that if the unique effect of the first guarded command to be executed is to set a new value to PC , we can substitute the expression for the initial value of PC in the reductional form program.

Lemma A.1.7 (*Void initial state*)

$$PC : [s_0, \left(\square \ b \rightarrow p \right), f] \sqsupseteq PC : [s, \left(\square \ b \rightarrow p \right), f]$$

Proof: Lwhileifunfold

$$\begin{aligned}
&LHS \\
&= \{Law\ 2.2.72\} \ (\mathbf{while} - \mathbf{if}\ \mathbf{unfold}) \\
&\quad \mathbf{var}\ PC : Int \bullet \\
&\quad \quad PC := s_0; \\
&\quad \quad \mathbf{while}\ (b \vee PC = s_0) \rightarrow \\
&\quad \quad \quad \mathbf{if}\ \left(\square \ b \rightarrow p \right) \mathbf{fi} \\
&\quad \quad \mathbf{end}; \\
&\quad \quad \{PC = f\} \qquad \mathbf{end} \\
&\sqsupseteq \{Law\ 2.2.13\} \ (\mathbf{var} - := \text{initial value}) \\
&\quad RHS
\end{aligned}$$

◇

The next lemma is used in the proof of Rule 4.4.15.

Lemma 4.4.2 (Symbolic execution of U):

Let $d = ((PC \geq s) \wedge (PC < f))$.

$$\begin{aligned} & \{PC = s\}; \textbf{ while } d \rightarrow \textbf{ if } U \textbf{ fi end} \\ \sqsupseteq & \{empty(F_v); MID; PC := PC + 1; \textbf{ while } d \rightarrow \textbf{ if } U \textbf{ fi end} \end{aligned}$$

□

The proof of Lemma 4.4.2 is similar to the proof of Lemma 5.1 in [87] (Appendix B). This proof uses the following abbreviations introduced in Section 4.4.2.

$$U = \left(\begin{array}{l} b_1 \rightarrow PC := PC + 1; \text{ Push}(F_v, v); PC := s + 1; \\ \square (PC = f_0 \rightarrow PC := f; \\ \quad \text{If } \neg(empty(F_v)) \rightarrow Pop(F_v, v) \textbf{ fi}) \\ \square b_2 \rightarrow q \end{array} \right)$$

$$Z = \left(\begin{array}{l} (PC = s) \rightarrow PC := f; \text{ Push}(F_v, v); PC := s + 1; \\ \square b_1 \rightarrow PC := PC + 1; \text{ Push}(F_v, v); PC := s + 1; \\ \square (PC = f_0) \rightarrow Pop(F_v, v) \\ \square b_2 \rightarrow q \end{array} \right)$$

We also need to define the following abbreviation.

$$T = \left(\begin{array}{l} b \rightarrow PC := f; \text{ Push}(F_v, v); PC := s + 1; \\ \square (PC = f_0 \rightarrow PC := f; \\ \quad \text{If } length(F_v) > 1 \rightarrow Pop(F_v, v) \textbf{ fi}) \\ \square b_o \rightarrow q \end{array} \right)$$

The following lemma establishes that a normal form program with guarded command set U is refined by a normal form program obtained from this one, by replacing U with T and the empty sequence with a sequence with one element.

Lemma A.1.8 (Lift of sequence variables) *If U and T are define as above, and k occurs only where it is explicitly shown below, then*

$$v, F_v : [s, U, f] \sqsubseteq v, F_v : [s, T, f]$$

□

Proof: Due to the requirement of the stack to be non-empty in the right-hand side, the data space of the two programs are different. Therefore, since the above can be regarded as a data refinement

problem, we define a simulation in order to compare the above programs.

Let $\Theta \stackrel{def}{=} \mathbf{dvar} \ F_w : Stack, w : T;$
 $\quad \{ \neg empty(F_v) \};$
 $\quad F_w := front(F_v);$
 $\quad w := last(F_v);$
 $\quad \mathbf{dend} \ F_v$
and $\Theta^{-1} \stackrel{def}{=} \mathbf{dvar} \ F_v : Stack;$
 $\quad F_v := F_w \frown \langle w \rangle$
 $\quad \mathbf{dend} \ F_w, w$

In the same way presented in Section 4.5, to define the simulation function we use $\Theta(p)$ to denote $\Theta; \ p; \ \Theta^{-1}$. The following properties are true of Θ and Θ^{-1} :

- (1) (Θ, Θ^{-1}) is a simulation.
- (2) $\Theta(push(F_w, v)) \sqsubseteq push(F_v, v)$
- (3) $\Theta(pop(F_w, v)) \sqsubseteq pop(F_v, v)$
- (4) $\Theta([\neg empty(F_w)]) \sqsubseteq [\#F_v > 1]$
- (5) Let $d_1 = (b \vee (c_o \wedge empty(F_w)) \vee b_0)$ and $d_2 = (b \vee (c_o \wedge (\#F_v > 1)) \vee b_0)$. Then
 $\Theta(\mathbf{while} \ d_1 \rightarrow \mathbf{if} \ S[F_w/F_v] \mathbf{fi} \ \mathbf{end}) \sqsubseteq \mathbf{while} \ d_2 \rightarrow \mathbf{if} \ U \mathbf{fi} \ \mathbf{end}$

$$\begin{aligned}
(1) \quad & \Theta; \Theta^{-1} \\
& = \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}), \\
& \quad \{Law\ 2.2.55\} (; := \text{commutation}) \text{ and} \\
& \quad \{Law\ 3.4.1\} (\text{laws of sequences}) \\
& \quad \mathbf{dvar} F_w : Stack, w : T; \{ \neg empty(F_v) \}; F_w := front(F_v); \\
& \quad \quad w := last(F_v); \\
& \quad \mathbf{dend} F_w, w \\
& \sqsubseteq \{Law\ 2.2.23\} (\mathbf{dend-} := \text{final value}), \\
& \quad \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}) \text{ and} \\
& \quad \{Law\ 2.2.62\} (; [b]; \{b\} \text{ simulation}) \\
& \quad \mathbf{skip} \\
& = \{Law\ 2.2.23\} (\mathbf{dend-} := \text{final value}), \\
& \quad \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}) \text{ and} \\
& \quad \{Law\ 2.2.60\} (; := \text{void assumption}) \\
& \quad \mathbf{dvar} F_v : Stack, F_v := F_w \frown \langle w \rangle; \{ \neg empty(F_v) \}; \mathbf{dend} F_v \\
& \\
& = \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}), \\
& \quad \{Law\ 2.2.55\} (; := \text{commutation}) \text{ and} \\
& \quad \{Law\ 3.4.1\} (\text{laws of sequences}) \\
& \quad \Theta^{-1}; \Theta
\end{aligned}$$

◇

$$\begin{aligned}
(2) \quad & \Theta(push(F_w, v)) \\
& = \Theta; push(F_w, v); \Theta^{-1} \\
& = \{Law\ 2.2.21\} (\mathbf{dvar}; \mathbf{dend} \text{ change scope}) \text{ and} \\
& \quad \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}) \\
& \quad \mathbf{dvar} F_w : Stack, w : T; \{ \neg empty(F_v) \}; F_w := front(F_v); \\
& \quad \quad w := last(F_v); push(F_w, v); F_v := F_w \frown \langle w \rangle; \\
& \quad \mathbf{dend} F_w, w \\
& = \{Law\ 2.2.55\} (; := \text{commutation}) \text{ and} \\
& \quad \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}) \\
& \quad \{ \neg empty(F_v) \}; push(F_v, v) \\
& \sqsubseteq \{Law\ 2.2.62\} (; [b]; \{b\} \text{ simulation}) \\
& \quad push(F_v, v)
\end{aligned}$$

◇

$$(3) \quad \text{Similar to (2).}$$

(4) Similar to (2).

(5) From (1) — (4) and the distributivity of Θ over iteration.

Then we have:

$$\begin{aligned}
& \text{RHS} \\
\sqsubseteq & \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}) \text{ and } (5) \\
& \mathbf{var} F_v : Stack, v : T \bullet \\
& \quad v := a_0; F_v := \langle k \rangle; \mathbf{dvar} F_w : Stack, w : T; \{ \neg \text{empty}(F_v) \}; \\
& \quad F_w := \text{front}(F_v); w := \text{last}(F_v); \\
& \quad \mathbf{while} d_1 \rightarrow \mathbf{if} U[F_w/F_v] \mathbf{fi} \mathbf{end}; F_v := F_w \frown \langle w \rangle; \mathbf{dend} F_w, w; \{c_o \wedge F_v = \langle k \rangle\} \\
& \mathbf{end} \\
= & \{Law\ 2.2.60\} (; := \text{void assumption}) \text{ and} \\
& \{Law\ 2.2.55\} (; := \text{commutation}) \\
& \mathbf{var} F_v : Stack, v : T \bullet \\
& \quad v := a_0; F_v := \langle k \rangle; \mathbf{dvar} F_w : Stack, w : T; F_w := \langle \rangle; w := k; \\
& \quad \mathbf{while} d_1 \rightarrow \mathbf{if} U[F_w/F_v] \mathbf{fi} \mathbf{end}; F_v := F_w \frown \langle w \rangle; \mathbf{dend} F_w, w; \{c_o \wedge F_v = \langle k \rangle\} \\
& \mathbf{end} \\
= & \{Law\ 2.2.55\} (; := \text{commutation}) \\
& \{Law\ 2.2.23\} (\mathbf{dend-} := \text{final value}) \text{ and} \\
& \{Law\ 2.2.25\} (\mathbf{dend}; \mathbf{dvar} \text{ simulation}) \\
& \mathbf{var} F_v : Stack, v : T \bullet \\
& \quad v := a_0; F_v := \langle k \rangle; \mathbf{dvar} F_w : Stack; F_w := \langle \rangle; \\
& \quad \mathbf{while} d_1 \rightarrow \mathbf{if} U[F_w/F_v] \mathbf{fi} \mathbf{end}; F_v := F_w \frown \langle k \rangle; \mathbf{dend} F_w; \{c_o \wedge F_v = \langle k \rangle\} \\
& \mathbf{end} \\
= & \{Law\ 2.2.63\} (; := \text{commute assumption}) \text{ and} \\
& \{Law\ 2.2.23\} (\mathbf{dend-} := \text{final value}) \\
& \mathbf{var} F_v : Stack, v : T \bullet \\
& \quad v := a_0; F_v := \langle k \rangle; \mathbf{dvar} F_w : Stack; F_w := \langle \rangle; \\
& \quad \mathbf{while} d_1 \rightarrow \mathbf{if} U[F_w/F_v] \mathbf{fi} \mathbf{end}; \{c_o \wedge \text{empty}(F_w)\}; \mathbf{dend} F_w; \\
& \mathbf{end}
\end{aligned}$$

$$\begin{aligned}
& \sqsupseteq \{Law\ 2.2.13\} (\mathbf{var-} := \text{initial value}) \text{ and} \\
& \{Law\ 2.2.6\} (\mathbf{var\ elim}) \\
& \mathbf{var}\ F_v : Stack, v : T \bullet \\
& \quad v := a_0; \mathbf{dvar}\ F_w : Stack; F_w := \langle \rangle; \\
& \quad \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U[F_w/F_v] \mathbf{fi\ end}; \{c_o \wedge \text{empty}(F_w)\}; \mathbf{dend}\ F_w; \\
& \mathbf{end} \\
& = \{Law\ 2.2.28\} (\mathbf{dvar}; \mathbf{dend}\ \text{conversion}) \text{ and} \\
& \{Law\ 2.2.23\} (\mathbf{dend-} := \text{final value}) \\
& LHS
\end{aligned}$$

◇

In doing so, now we can prove Lemma 4.4.2. In the following we repeat the inequation to be proved.

Consider $d_1 = (b \vee (c_0 \wedge \neg \text{empty}(F_v)) \vee b_0)$, then we have:

$$\begin{aligned}
& [b]; \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end} \sqsupseteq \{\text{empty}(F_v)\}; MID; v := r; \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end} \\
& [b]; \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end} \\
& \sqsupseteq \{Law\ 2.2.72\} (\mathbf{while} - \mathbf{if}\ \text{unfold}) \text{ and} \\
& \{Law\ 2.2.62\} ([b]; \{b\} = [b] \sqsupseteq \mathbf{skip}) \\
& v := r; \text{push}(F_v, v); v := a_0; \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end} \\
& = \{Law\ 2.2.76\} (\mathbf{while}\ \text{sequence}) \text{ and} \\
& \text{Let } d_2 = (b \vee (c_0 \wedge \#F_v > 1) \vee b) \\
& v := r; \text{push}(F_v, v); v := a_0; \mathbf{while}\ d_2 \rightarrow \mathbf{if}\ T \mathbf{fi\ end}; \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end} \\
& \sqsupseteq \{Law\ 2.2.62\} (\{b\} \sqsubseteq \mathbf{skip}) \text{ and} \\
& \{Law\ 2.2.72\} (\mathbf{while} - \mathbf{if}\ \text{unfold}) \\
& v := r; \text{push}(F_v, v); v := a_0; \mathbf{while}\ d_2 \rightarrow \mathbf{if}\ T \mathbf{fi\ end}; \\
& \{c_0 \wedge \#F_v > 1\} \text{pop}(F_v, v); \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end} \\
& = \{Law\ 2.2.6\} (\mathbf{var\ elim}) \text{ and} \\
& \{Definition\ 4.4.4\} \text{Abbreviation for the flat iteration} \\
& v := r; \text{push}(F_v, v); v : [a_0, T, (c_0 \wedge \#F_v > 1)] \text{pop}(F_v, v); \mathbf{while}\ d_1 \rightarrow \mathbf{if}\ U \mathbf{fi\ end}
\end{aligned}$$

$$\begin{aligned}
&= \{Law\ 2.2.7\} \text{ (var rename),} \\
&\quad \text{assuming } w \text{ is fresh and } t' = t[w/v] \\
&\quad v := r; \text{ push}(F_v, v); \quad w : [a'_0, T', (c'_0 \wedge \#F_v > 1)] \text{ pop}(F_v, v); \\
&\quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end} \\
\\
&\sqsubseteq (\{c'_0 \wedge \#F_v > 1\} \sqsubseteq \{c'_0 \wedge \#F_v > 1\}; \{F_v = \langle v \rangle\} = \{c'_0 \wedge F_v = \langle v \rangle\}) \\
&\quad v := r; \text{ push}(F_v, v); \quad w : [a'_0, T', (c'_0 \wedge F_v = \langle v \rangle)] \text{ pop}(F_v, v); \\
&\quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end} \\
\\
&= \{Definition\ 3.4.3\} \text{ Pushing and popping values onto/from the operand stack} \\
&\quad v := r; \quad F_v := \langle v \rangle \frown F_v; \quad w : [a'_0, T', (c'_0 \wedge F_v = \langle v \rangle)] \quad F_v := \langle \rangle; \\
&\quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end} \\
\\
&\sqsubseteq \{Law\ 2.2.62\} (\{b\} \sqsubseteq \mathbf{skip}) \\
&\quad v := r; \quad \{empty(F_v)\}; \quad F_v := \langle v \rangle \frown F_v; \quad w : [a'_0, T', (c'_0 \wedge F_v = \langle v \rangle)] \quad F_v := \langle \rangle; \\
&\quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end} \\
\\
&= \{Law\ 2.2.6\} \text{ (var elim) and} \\
&\quad (\{empty(F_v)\}; \quad F_v := \langle \rangle = \{empty(F_v)\}) \\
&\quad v := r; \quad \{empty(F_v)\}; \quad w, F_v : [a'_0, T', (c'_0 \wedge F_v = \langle v \rangle)] \quad F_v := \langle \rangle; \\
&\quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end} \\
\\
&= \{Lemma\ A.1.8\} \text{ (var elim) and} \\
&\quad \{Law\ 2.2.7\} \text{ (var rename)} \\
&\quad v := r; \quad \{empty(F_v)\}; \quad MID; \quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end} \\
\\
&\sqsubseteq \{Law\ 2.2.63\} (; := \text{commute assumption}) \\
&\quad \{empty(F_v)\}; \quad MID; \quad v := r; \quad \mathbf{while } d_1 \rightarrow \mathbf{if } U \mathbf{fi end}
\end{aligned}$$

◇

A.2 Redirection of method calls — Proofs

Proof of Rule 4.2.1 (Redirecting a call with a result parameter):

Consider that the following class declaration

```
class C extends D
  ads
  meth m  $\hat{=}$  pc
  ops
end
```

is included in *cds* and $cds, A \triangleright le : C$. Then, there is the following associated method declaration in the class *L*

```
class L extends object
  ads
  meth lm  $\hat{=}$  (val Sin : Stack; res Sout : Stack •
    var o : Object; v : T •
      Pop(Sin, o); Pop(Sin, v); pc[o/self](v);
      Push(Sin, o); Sout := Sin
    end) ops
end
```

such that

```
cdsRVM cds L, N  $\triangleright$ 
  ((C)le).m(x)  $\sqsubseteq$  {le  $\neq$  null  $\wedge$  le  $\neq$  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 one value parameter; (2) *S* and *V* are fresh names; (3) *le* is not **null** nor **error**; (4) all attributes referenced by *lm* are public and declared in *cds*; (5) *N* is either **main** or *L* □

The above rule is only applied in the main command and body of methods declared in *L*. There is no need to transform those methods in *cds*.

In our proof, based on the results presented in previous compilation phase, we only need to

consider the case in which $((C)le).m(x)$ is a call to an original method whose target is the object stored in le .

Proof Rule 4.2.1 (Redirecting a call with a result parameter):

$$\begin{aligned}
& ((C)le).m(x) \\
= & \{Law\ 2.2.86\} \text{ (method call elimination)} \\
& \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\};\ pc[le/\mathbf{self}](x); \\
= & \{Law\ 2.2.15\} \text{ (var block-vres)} \\
& \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\}; \\
& \mathbf{var}\ o : Object \bullet \\
& \quad o := (C)le;\ pc[o/\mathbf{self}](x);\ le := (C)o \\
& \mathbf{end} \\
\\
= & \{Law\ 2.2.17\} \text{ (var block-res)} \\
& \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\}; \\
& \mathbf{var}\ o : Object \bullet \\
& \quad o := (C)le; \\
& \quad \mathbf{var}\ a : T \bullet \\
& \quad \quad pc[o/\mathbf{self}](a);\ x := a \\
& \quad \mathbf{end};\ le := (C)o \\
& \mathbf{end} \\
\\
= & \{Law\ 2.2.9\} \text{ (var-; left dist)} \\
& \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\}; \\
& \mathbf{var}\ o : Object \bullet \\
& \quad o := (C)le; \\
& \quad \mathbf{var}\ a : T \bullet \\
& \quad \quad pc[o/\mathbf{self}](a);\ x := a;\ le := (C)o; \\
& \quad \mathbf{end} \\
& \mathbf{end}
\end{aligned}$$

= {*Rule 4.3.4*} (Simplification of assignment)
 { $le \neq \mathbf{null} \wedge le \neq \mathbf{error}$ };
var $o : \mathbf{Object}$ •
 var $S : \mathbf{Stack}$ •
 $S := \mathbf{new Stack}; \text{Push}(S, le); \text{Pop}(S, o)$
 end
 var $a : T$ •
 $pc[o/\mathbf{self}](a);$
 var $S : \mathbf{Stack}$ •
 $S := \mathbf{new Stack}; \text{Push}(S, a); \text{Pop}(S, x)$
 end
 var $S : \mathbf{Stack}$ •
 $S := \mathbf{new Stack}; \text{Push}(S, o); \text{Pop}(S, le)$
 end
 end
end

= {*Lemma 4.3.1*} (Assignment does not affect the resulting stack)
 { $le \neq \mathbf{null} \wedge le \neq \mathbf{error}$ };
var $o : \mathbf{Object}$ •
 var $S : \mathbf{Stack}$ •
 $S := \mathbf{new Stack}; \text{Push}(S, le); \text{Pop}(S, o); \{\mathbf{empty}(S)\}$
 end
 var $a : T$ •
 $pc[o/\mathbf{self}](a);$
 { o is C };
 var $S : \mathbf{Stack}$ •
 $S := \mathbf{new Stack}; \text{Push}(S, a); \text{Pop}(S, x); \{\mathbf{empty}(S)\}$
 end
 var $S : \mathbf{Stack}$ •
 $S := \mathbf{new Stack}; \text{Push}(S, o); \text{Pop}(S, le); \{\mathbf{empty}(S)\}$
 end
 end
end

= {*Law* 2.2.9} (**var**-, left dist), {*Law* 2.2.10} (**var**-, right dist),
 {*Law* 2.2.12} (**var**-, dist), and {*Law* 2.2.4} (**var** association)
 {*le* ≠ **null** ∧ *le* ≠ **error**};

```

var S : Stack •
  S := new Stack; Push(S, le);
  var o : Object; a : T •
    Pop(S, o); {empty(S)}
    pc[o/self](a);
    S := new Stack; Push(S, a); Pop(S, x); {empty(S)}
    S := new Stack; Push(S, o); Pop(S, le); {empty(S)}
  end
end

```

= {*Law* 2.2.55} (; := commutation)
 {*le* ≠ **null** ∧ *le* ≠ **error**};

```

var S : Stack •
  S := new Stack; Push(S, le);
  var o : Object; a : T •
    Pop(S, o); {empty(S)}; S := new Stack;
    pc[o/self](a);
    Push(S, a); Pop(S, x); {empty(S)}; S := new Stack;
    {o is C}; Push(S, o); Pop(S, le); {empty(S)}
  end
end

```

⊆ {*Law* 2.2.57} (; := void assignment) and {*Law* 2.2.62} (; [*b*]; {*b*} simulation)
 {*le* ≠ **null** ∧ *le* ≠ **error**};

```

var S : Stack •
  S := new Stack; Push(S, le);
  var o : Object; a : T •
    Pop(S, o);
    pc[o/self](a);
    Push(S, a); Pop(S, x);
    Push(S, o); Pop(S, le);
  end
end

```


= {*Lemma A.1.3*} ⟨Combining two pairs of Pushes and Pops⟩
 { $le \neq \mathbf{null} \wedge le \neq \mathbf{error}$ };
var $S : \mathit{Stack}$ •
 $S := \mathbf{new} \ \mathit{Stack}; \ \mathit{Push}(S, le);$
 var $o : \mathit{Object}; \ a : T$ •
 $\mathit{Pop}(S, o);$
 $pc[o/\mathbf{self}](a);$
 $\mathit{Push}(S, a); \ \mathit{Push}(S, o); \ \mathit{Pop}(S, le); \ \mathit{Pop}(S, x);$
 end
end

= {*Law 2.2.10*} (**var**-, right dist) and {*Law 2.2.46*} (- skip unit)
 { $le \neq \mathbf{null} \wedge le \neq \mathbf{error}$ };
var $S : \mathit{Stack}$ •
 $S := \mathbf{new} \ \mathit{Stack}; \ \mathit{Push}(S, le);$
 var $o : \mathit{Object}; \ a : T$ •
 $\mathit{Pop}(S, o);$
 $pc[o/\mathbf{self}](a);$
 $\mathit{Push}(S, a); \ \mathit{Push}(S, o);$
 end;
 $\mathit{Pop}(S, le); \ \mathit{Pop}(S, x);$
end

= {*Law 2.2.15*} (**var** block-vres)
 { $le \neq \mathbf{null} \wedge le \neq \mathbf{error}$ };
var $S : \mathit{Stack}$ •
 $S := \mathbf{new} \ \mathit{Stack}; \ \mathit{Push}(S, le);$
 var $S_v : \mathit{Stack}$ •
 var $o : \mathit{Object}; \ a : T$ •
 $S_v := S; \ \mathit{Pop}(S_v, o);$
 $pc[o/\mathbf{self}](a);$
 $\mathit{Push}(S_v, a); \ \mathit{Push}(S_v, o); \ S := S_v$
 end
 end;
 $\mathit{Pop}(S, le); \ \mathit{Pop}(S, x);$
end

= {*Law* 2.2.17} (**var** block-res) and {*Law* 2.2.4} (**var** association)
 {*le* ≠ **null** ∧ *le* ≠ **error**};
var *S* : *Stack* •
 S := **new** *Stack*; *Push*(*S*, *le*);
 var *S_v*, *S_r* : *Stack* •
 var *o* : *Object*; *a* : *T* •
 S_v := *S*; *Pop*(*S_v*, *o*);
 $pc[o/\mathbf{self}](a)$;
 Push(*S_v*, *a*); *Push*(*S_v*, *o*); *S_r* := *S_v*
 S := *S_r*;
 end
 end;
 Pop(*S*, *le*); *Pop*(*S*, *x*);
end

= {*Law* 2.2.18} (Pcom elimination-val) and {*Law* 2.2.19} (Pcom elimination-res)
 {*le* ≠ **null** ∧ *le* ≠ **error**};
var *S* : *Stack* •
 S := **new** *Stack*; *Push*(*S*, *le*);
 (**val** *S_v* : *Stack*; **res** *S_r* : *Stack* •
 var *o* : *Object*; *a* : *T* •
 Pop(*S_v*, *o*);
 $pc[o/\mathbf{self}](a)$;
 Push(*S_v*, *a*); *Push*(*S_v*, *o*); *S_r* := *S_v*
 end
 end)(*S*, *S*);
 Pop(*S*, *le*); *Pop*(*S*, *x*);
end

$\sqsubseteq$ $\{Law\ 2.2.6\}$ (**var** elim), $\{Law\ 2.2.4\}$ (**var** association), and
 $\{Law\ 2.2.13\}$ (**var**- := initial value)
 $\{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\};$
var $S : Stack; \ V : L \bullet$
 $V := \mathbf{new}\ L;$
 $S := \mathbf{new}\ Stack; \ Push(S, le);$
(val $S_v : Stack; \mathbf{res}\ S_r : Stack \bullet$
 $\mathbf{var}\ o : Object; \ a : T \bullet$
 $Pop(S_v, o);$
 $pc[o/\mathbf{self}](a);$
 $Push(S_v, a); \ Push(S_v, o); \ S_r := S_v$
end
end)(S, S);
 $Pop(S, le); \ Pop(S, x);$
end

$\sqsubseteq$ $\{Law\ 2.2.86\}$ (method call elimination)
 $\{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\};$
var $S : Stack; \ V : L \bullet$
 $S := \mathbf{new}\ Stack; \ Push(S, le);$
 $V.lm(S, S);$
 $Pop(S, le); \ Pop(S, x);$
end

◇

Proof Rule 4.2.2 (Value parameter): **Similar to Rule 4.2.1**

A.3 Simplification of expressions and some source transformations — Proofs

Proof of Rule 4.3.4 (Assignment of a variable):

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

□

$$\begin{aligned} & \text{RHS} \\ = & \{Law\ 2.2.6\} (\text{var elim}) \\ & \text{var } S : \text{Stack}; \bullet \\ & \text{end}; \ x := e \\ \\ = & \{Law\ 2.2.9\} (\text{var-; left dist}) \\ & \text{var } S : \text{Stack}; \bullet \\ & \quad x := e \\ & \text{end} \\ \\ \sqsubseteq & \{Law\ 2.2.13\} (\text{var- := initial value}) \text{ and } \{Law\ 2.2.11\} (\text{var- := final value}) \\ & \text{var } S : \text{Stack}; \bullet \\ & \quad S := \text{new Stack}; \ S.s := \langle e \rangle \cap S.s; \ x := e; \ S.s := \text{tail}(S.s) \\ & \text{end} \\ \\ = & \{Law\ 2.2.54\} (; := substitution) \\ & \text{var } S : \text{Stack}; \bullet \\ & \quad S := \text{new Stack}; \\ & \quad S.s := \langle e \rangle \cap S.s; \ x := \text{head}(S.s); \ S.s := \text{tail}(S.s) \\ & \text{end} \\ \\ = & \{Definition\ 4.2.1\} \langle \text{PushandPopdefinitions} \rangle \\ & \text{var } S : \text{Stack}; \bullet \\ & \quad S := \text{new Stack}; \ \text{Push}(S, e); \ \text{Pop}(S, x) \\ & \text{end} \end{aligned}$$

$$\begin{aligned}
&= \{Definition\ 4.2.1\} \langle PushandPopdefinitions \rangle \\
&\quad \mathbf{var}\ S : Stack; \quad \bullet \\
&\quad S := \mathbf{new}\ Stack; \\
&\quad S.s := \langle e \rangle \frown S.s; \ x := head(S.s); \ S.s := tail(S.s) \\
&\quad \mathbf{end} \\
\\
&= \{Law\ 2.2.54\} (\ ; := \mathbf{substitution}) \\
&\quad \mathbf{var}\ S : Stack; \quad \bullet \\
&\quad S := \mathbf{new}\ Stack; \ S.s := \langle e \rangle \frown S.s; \ x := e; \ S.s := tail(S.s) \\
&\quad \mathbf{end} \\
\\
&= \{Law\ 2.2.55\} (\ ; := \mathbf{commutation}) \text{ and } \{Law\ 2.2.9\} (\mathbf{var-; left dist}) \\
&\quad \mathbf{var}\ S : Stack; \quad \bullet \\
&\quad S := \mathbf{new}\ Stack; \ S.s := \langle e \rangle \frown S.s; \ S.s := tail(S.s) \\
&\quad \mathbf{end}; \ x := e \\
\\
&\sqsubseteq \{Law\ 2.2.11\} (\mathbf{var- := final value}) \text{ and } \{Law\ 2.2.6\} (\mathbf{var elim}) \\
&\quad x := e
\end{aligned}$$

◇

Proof of Rule 4.3.5 (Binary Operator):

$$Push(S, e \ \mathbf{bop} \ f) \quad \sqsubseteq \quad Push(S, e); \ Push(S, f); \ Bop(S)$$

□

$$\begin{aligned}
&\text{RHS} \\
&= \{Definition\ 4.2.1\} \langle Push, PopandBopdefinitions \rangle \\
&\quad S.s := \langle e \rangle \frown S.s; \ S.s := \langle f \rangle \frown S.s; \\
&\quad S.s := \langle head(tail(S.s) \ \mathbf{bop} \ head(S.s)) \rangle \frown tail(tail(S.s)) \\
\\
&= \{Law\ 2.2.53\} (\ ; := \mathbf{combination}) \\
&\quad S.s := \langle e \rangle \frown S.s; \\
&\quad S.s := \langle head(tail(\langle f \rangle \frown S.s)) \ \mathbf{bop} \ head(\langle f \rangle \frown S.s)) \rangle \frown tail(tail(\langle f \rangle \frown S.s)) \\
\\
&= \{Law\ 3.4.1\} (\mathbf{laws of sequences}) \\
&\quad S.s := \langle e \rangle \frown S.s; \ S.s := \langle head(S.s) \ \mathbf{bop} \ f \rangle \frown tail(S.s)
\end{aligned}$$

$$\begin{aligned}
&= \{Law\ 2.2.53\} \text{ (; := combination)} \\
&\quad S.s := \langle head(\langle e \rangle \frown S.s) \text{ bop } f \rangle \frown tail(\langle e \rangle \frown S.s) \\
\\
&= \{Law\ 3.4.1\} \text{ (laws of sequences)} \\
&\quad S.s := \langle e \text{ bop } f \rangle \frown S.s \\
\\
&= \{Law\ 2.2.18\} \text{ (Pcom elimination-val)} \text{ and } \{Law\ 2.2.86\} \text{ (method call elimination)} \\
&\quad LHS
\end{aligned}$$

◇

Proof of Rule 4.3.6 (Unary Operator):

$$Push(S, \text{uop } e) \quad \sqsubseteq \quad Push(S, e); \quad Uop(S)$$

□

$$\begin{aligned}
&\quad RHS \\
&= \{Definition\ 4.2.1\} \langle Push, Pop, \text{ and } Uop \text{ definitions} \rangle \\
&\quad S.s := \langle e \rangle \frown S.s; \quad S.s := \langle head(\text{uop } head(S.s)) \rangle \frown tail(S.s) \\
\\
&= \{Law\ 2.2.53\} \text{ (; := combination)} \\
&\quad S.s := \langle head(\text{uop } head(\langle e \rangle \frown S.s)) \rangle \frown tail(\langle e \rangle \frown S.s) \\
\\
&= \{Law\ 3.4.1\} \text{ (laws of sequences)} \\
&\quad S.s := \langle head(\text{uop } e) \rangle \frown S.s \\
\\
&= \{Law\ 2.2.18\} \text{ (Pcom elimination-val)} \text{ and } \{Law\ 2.2.86\} \text{ (method call elimination)} \\
&\quad LHS
\end{aligned}$$

◇

Proof of Rule 4.3.7 (Pushing an attribute): If $cds_{RVM} \text{ cds}, N \triangleright le : C$

$$\begin{aligned}
Push(S, le.t) \quad &\sqsubseteq \quad Push(S, le); \quad Getfd(S, C.t) \\
&\text{end}
\end{aligned}$$

□

$$\begin{aligned}
& \text{Push}(S, le.t) \\
= & \{ \text{Definition 4.2.1} \} \text{ Abbreviations — operand stack} \\
& S.s := \langle le.t \rangle \frown S.s \\
= & \{ \text{Law 2.2.53} \} (; := \text{combination}) \\
& \{ \text{Law 2.2.54} \} (; := \text{substitution}) \\
& S.s := \langle le \rangle \frown S.s \quad S.s := \langle (C \text{ head}(S.s)).t \rangle \frown \text{tail}(s.s) \\
= & \{ \text{Definition 4.2.1} \} \text{ Abbreviations — operand stack} \\
& \text{Push}(S, le); \text{ Getfd}(S, C.t)
\end{aligned}$$

◇

Proof of Rule 4.3.8 (Popping to an attribute):

If $cds, N \triangleright le : C$

$$\begin{aligned}
& \text{Pop}(S, (C \text{ le}).t) \quad \sqsubseteq \quad \text{Push}(S, le); \text{ Putfd}(S, C.t) \\
& \text{end}
\end{aligned}$$

□

Similar to Rule 4.3.7.

Proof of Rule 4.3.9 (Object Creation - Simplification):

$$\begin{aligned}
(x := \text{new } C) = & \text{var } S : \text{Stack}; \bullet \\
& S := \text{new } \text{Stack}; \text{ Push}(S, \text{new } C); \text{ Pop}(S, x) \\
& \text{end}
\end{aligned}$$

□

Similar to Rule 4.3.4

Proof of Rule 4.3.10 (Type test - Simplification):

(Type test - Simplification)

$$\begin{aligned}
(x := (o \text{ is } C)) = & \text{var } S : \text{Stack}; \bullet \\
& S := \text{new } \text{Stack}; \text{ Push}(S, o); \text{ Is}(C); \text{ Pop}_b(S, x) \\
& \text{end}
\end{aligned}$$

□

$$\begin{aligned}
& \text{RHS} \\
= & \{ \text{Law 2.2.6} \} (\text{var elim}) \\
& \text{var } S : \text{Stack}; \bullet \\
& \text{end}; \quad x := (o \text{ is } C) \\
= & \{ \text{Law 2.2.9} \} (\text{var-; left dist}) \\
& \text{var } S : \text{Stack}; \bullet \\
& \quad x := (o \text{ is } C) \\
& \text{end}
\end{aligned}$$

$\sqsubseteq$ $\{Law\ 2.2.13\}$ (**var- := initial value**) and $\{Law\ 2.2.11\}$ (**var- := final value**)
var $S : Stack$; •
 $S := \mathbf{new}\ Stack$;
 $S.s := \langle (DataBool; info : (o\ \mathbf{is}\ C)) \rangle \frown S.s$;
 $x := (o\ \mathbf{is}\ C); S.s := tail(S.s)$
end

$=$ $\{Law\ 2.2.54\}$ (**:= substitution**)
var $S : Stack$; •
 $S := \mathbf{new}\ Stack$;
 $S.s := \langle (DataBool; info : (o\ \mathbf{is}\ C)) \rangle \frown S.s$;
 $x := head(S.s); S.s := tail(S.s)$
end

$=$ $\{Law\ 2.2.53\}$ (**:= combination**)
var $S : Stack$; •
 $S := \mathbf{new}\ Stack$;
 $S.s := \langle o \rangle \frown S.s$;
 $S.s := \langle (head(S.s)\mathbf{is}\ C) \rangle \frown tail(S.s)$;
 $x := head(S.s); S.s := tail(S.s)$
end

$=$ $\{Law\ 2.2.53\}$ (**:= combination**)
 $\{Definition\ 4.2.1\}$ (**Push definition**)
 $\{Definition\ 4.2.3\}$ ($Is(S, C)$ and $Pop_b(S, b)$ definitions)
var $S : Stack$; •
 $S := \mathbf{new}\ Stack; Push(S, o); Is(S, C); Pop_b(S, x)$;
end

$=$ $\{Law\ 2.2.53\}$ (**:= combination**)
 $\{Definition\ 4.2.1\}$ (**Push definition**)
 $\{Definition\ 4.2.3\}$ ($Is(S, C)$ and $Pop_b(S, b)$ definitions)
var $S : Stack$; •
 $S := \mathbf{new}\ Stack$;
 $S.s := \langle o \rangle \frown S.s$;
 $S.s := \langle (head(S.s)\mathbf{is}\ C) \rangle \frown tail(S.s)$;
 $x := head(S.s); S.s := tail(S.s)$
end

= {Law 2.2.53} (; := **combination**)
var $S : Stack$; •
 $S := \mathbf{new} \ Stack$;
 $S.s := \langle (DataBool; info : (o \text{ is } C)) \rangle \cap S.s$;
 $x := head(S.s); \ S.s := tail(S.s)$
end

= {Law 2.2.54} (; := **substitution**)
var $S : Stack$; •
 $S := \mathbf{new} \ Stack$;
 $S.s := \langle (DataBool; info : (o \text{ is } C)) \rangle \cap S.s$;
 $x := (o \text{ is } C); \ S.s := tail(S.s)$
end

= {Law 2.2.55} (; := **commutation**) and {Law 2.2.9} (**var-; left dist**)
var $S : Stack$; •
 $S := \mathbf{new} \ Stack$;
 $S.s := \langle (DataBool; info : (o \text{ is } C)) \rangle \cap S.s$;
 $S.s := tail(S.s)$
end; $x := (o \text{ is } C)$

$\sqsubseteq$ {Law 2.2.11} (**var- := final value**) and {Law 2.2.6} (**var elim**)
 $x := (o \text{ is } C)$

◇

Proof of Rule 4.3.2 (Conditions of if):

If $x_1, \dots, x_n$ **do not occur in** b_i **nor in** p_i ;

if $\bigwedge_{1 \leq i \leq n} b_i \rightarrow p_i$ **fi** $\sqsubseteq$ **var** $x_1, \dots, x_n : boolean$ •
 $x_1 := b_1; \dots; x_n := b_n$;
if $\bigwedge_{1 \leq i \leq n} x_i \rightarrow p_i$ **fi**;
end

□

RHS
= {Law 2.2.53} (; := **combination**)
LHS

◇

Proof of Rule 4.3.3 (Conditions of **while**):

If x does not occur in b nor in p ;

$$\begin{array}{l} \text{while } b \bullet p \text{ end} \quad \sqsubseteq \quad \text{var } x : \text{boolean} \bullet \\ \quad x := b; \text{ while } x \bullet p; x := b \text{ end} \\ \quad \text{end} \end{array}$$

□

Similar to Rule 4.3.2

Proof of Rule 4.3.11 ($Push$; Pop effect):

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

□

$$\begin{array}{l} Push(S, x); Pop(S, x) \\ = \{ \text{Definition 4.2.1} \} \langle Push \text{ and } Pop \text{ definitions} \rangle \\ \quad S.s := \langle x \rangle \wedge S.s; x := head(S.s); S.s := tail(S.s) \\ \sqsubseteq \{ \text{Law 2.2.2} \} (:= \text{identity}), \{ \text{Law 2.2.3} \} (:= \text{symmetry}), \\ \quad \{ \text{Law 2.2.54} \} (; := \text{substitution}) \text{ and } \{ \text{Law 2.2.53} \} (; := \text{combination}) \\ \quad x := head(\langle x \rangle \wedge S.s); S.s := tail(\langle x \rangle \wedge S.s) \\ = \{ \text{Law 3.4.1} \} (\text{laws of sequences}) \\ \quad x := x; S.s := S.s \\ \sqsubseteq \{ \text{Law 2.2.1} \} (:= \text{skip}) \\ \quad \text{skip} \end{array}$$

◇

Proof of Lemma 4.3.3 ($Push$; Pop effect):

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

□

$$\begin{aligned}
& LHS \\
= & \{ \text{Definition 4.2.1} \} \langle \text{Push and Pop definitions} \rangle \\
& x := \text{head}(S.s); \ S.s := \text{tail}(S.s); \ S.s := \langle x \rangle \wedge S.s; \\
\sqsubseteq & \{ \text{Law 2.2.2} \} (:= \text{identity}), \{ \text{Law 2.2.3} \} (:= \text{symmetry}), \\
& \{ \text{Law 2.2.54} \} (; := \text{substitution}) \text{ and } \{ \text{Law 2.2.53} \} (; := \text{combination}) \\
& x := \text{head}(S.s); \ S.s := \langle \text{head}(S.s) \rangle \wedge \text{tail}(S.s); \\
= & \{ \text{Law 3.4.1} \} (\text{laws of sequences}) \\
& x := \text{head}(S.s); \ S.s := S.s \\
\sqsubseteq & \{ \text{Law 2.2.1} \} (:= \text{skip}) \\
& x := \text{head}(S.s)
\end{aligned}$$

◇

Proof of Rule 4.3.12 ($\text{Pop}(S, x); \text{Push}(S, x)$) — assignment:

$$\text{Pop}(S, x); \text{Push}(S, x); \ V.lm(S, S); \text{Pop}(S, x) = V.lm(S, S); \text{Pop}(S, x)$$

□

$$\begin{aligned}
& LHS \\
= & \{ \text{Lemma 4.3.3} \} \langle \text{Pop; Push effect} \rangle \\
& x := \text{head}(S.s); \ V.lm(S, S); \text{Pop}(S, x) \\
= & \{ \text{Law 2.2.6} \} (\text{var elim}) \text{ and } \{ \text{Law 2.2.13} \} (\text{var-} := \text{initial value}) \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s) \text{ end}; \ x := \text{head}(S.s); \ V.lm(S, S); \text{Pop}(S, x) \\
= & \{ \text{Law 2.2.9} \} (\text{var-; left dist}) \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s); \ x := \text{head}(S.s); \ V.lm(S, S); \text{Pop}(S, x) \text{ end} \\
= & \{ \text{Law 2.2.53} \} (; := \text{combination}) \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s); \ x := Y; \ V.lm(S, S); \text{Pop}(S, x) \text{ end} \\
= & \{ \text{Law 2.2.55} \} (; := \text{commutation}) \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s); \ V.lm(S, S); \ x := Y; \text{Pop}(S, x) \text{ end} \\
= & \{ \text{Definition 4.2.1} \} \langle \text{Pop definition} \rangle \\
& \text{var } Y : \text{Object} \bullet \\
& \quad Y := \text{head}(S.s); \ V.lm(S, S); \ x := Y; \ x := \text{head}(S.s); \ S.s := \text{tail}(S.s) \\
& \text{end} \\
= & \{ \text{Law 2.2.53} \} (; := \text{combination}) \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s); \ V.lm(S, S); \ x := \text{head}(S.s); \ S.s := \text{tail}(S.s) \text{ end} \\
= & \{ \text{Definition 4.2.1} \} \langle \text{Pop definition} \rangle \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s); \ V.lm(S, S); \text{Pop}(S, x) \text{ end} \\
= & \{ \text{Law 2.2.10} \} (\text{var-; right dist}) \\
& \text{var } Y : \text{Object} \bullet Y := \text{head}(S.s) \text{ end}; \ V.lm(S, S); \text{Pop}(S, x) \\
\sqsubseteq & \{ \text{Law 2.2.11} \} (\text{var-} := \text{final value}) \text{ and } \{ \text{Law 2.2.4} \} (\text{var association}) \\
& V.lm(S, S); \text{Pop}(S, x)
\end{aligned}$$

◇

A.4 Control elimination — Proofs

Proof of Rule 4.4.1 (skip - Control Elimination):

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

□

$$\begin{aligned}
 & \text{RHS} \\
 = & \{ \text{Definition 4.4.1} \} \text{ (Abbreviation for the } PC \text{ frame)} \text{ and} \\
 & \{ \text{Definition 3.4.2} \} \text{ (Instruction set - Miscellaneous)} \\
 & \mathbf{var } PC : Int \bullet \\
 & \quad PC := s; \\
 & \quad \mathbf{while } PC \geq s \wedge PC < s + 1 \rightarrow \\
 & \quad \quad \mathbf{if } PC = s \rightarrow PC := PC + 1 \mathbf{ fi} \\
 & \quad \mathbf{end}; \{ PC = s + 1 \} \\
 & \mathbf{end} \\
 = & \{ \text{Law 2.2.61} \} (; := \mathbf{void assertion}) \text{ and } \{ \text{Law 2.2.71} \} (\mathbf{while unfold}) \\
 & \mathbf{var } PC : Int \bullet \\
 & \quad PC := s; [PC = s]; \\
 & \quad \mathbf{if } PC = s \rightarrow PC := PC + 1 \mathbf{ fi} \\
 & \quad \mathbf{while } PC \geq s \wedge PC < s + 1 \rightarrow \\
 & \quad \quad \mathbf{if } PC = s \rightarrow PC := PC + 1 \mathbf{ fi} \\
 & \quad \mathbf{end}; \{ PC = s + 1 \}; \\
 & \mathbf{end} \\
 \sqsubseteq & \{ \text{Law 2.2.37} \} (\mathbf{if selection}) \text{ and } \{ \text{Law 2.2.61} \} (; := \mathbf{void assertion}) \\
 & \mathbf{var } PC : Int \bullet \\
 & \quad PC := s; [PC = s]; \\
 & \quad PC := PC + 1; [PC = s + 1]; \\
 & \quad \mathbf{while } PC \geq s \wedge PC < s + 1 \rightarrow \\
 & \quad \quad \mathbf{if } PC = s \rightarrow PC := PC + 1 \mathbf{ fi} \\
 & \quad \mathbf{end}; \{ PC = s + 1 \} \\
 & \mathbf{end}
 \end{aligned}$$

= $\{Law\ 2.2.70\}$ (**while elimination**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $PC := PC + 1; [PC = s + 1]; \{PC = s + 1\}$
end

$\sqsubseteq \{Law\ 2.2.62\} (; [b]; \{b\}$ **simulation**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $PC := PC + 1$
end

= $\{Law\ 2.2.11\}$ (**var- := final value**) and $\{Law\ 2.2.61\} (; := \text{void assertion})$
var $PC : Int \bullet$ **end**

= $\{Law\ 2.2.6\}$ (**var elim**)
skip

◇

Proof of Rule 4.4.2 (Push variable):

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

□

RHS
= $\{Definition\ 4.4.1\}$ (**Abbreviation for the PC frame**) and
 $\{Definition\ 4.4.2\}$ (**Abbreviations introduced in the Control Elimination phase**)
var $PC : Int \bullet$
 $PC := s;$
while $PC \geq s \wedge PC < s + 2 \rightarrow$
 if $PC = s \rightarrow S.s := \langle x \rangle \wedge S.s; PC := PC + 2$ **fi**
 end; $\{PC = s + 2\}$
end

$= \{Law\ 2.2.61\} (\text{; := void assertion}) \text{ and } \{Law\ 2.2.71\} (\text{while unfold})$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
if $PC = s \rightarrow S.s := \langle x \rangle \wedge S.s; PC := PC + 2$ **fi**
while $PC \geq s \wedge PC < s + 2 \rightarrow$
 $\text{if } PC = s \rightarrow S.s := \langle x \rangle \wedge S.s; PC := PC + 2$ **fi**
end; $\{PC = s + 2\};$
end

$\sqsubseteq \{Law\ 2.2.37\} (\text{if selection}) \text{ and } \{Law\ 2.2.61\} (\text{; := void assertion})$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $S.s := \langle x \rangle \wedge S.s; PC := PC + 2; [PC = s + 2];$
while $PC \geq s \wedge PC < s + 2 \rightarrow$
 $\text{if } PC = s \rightarrow S.s := \langle x \rangle \wedge S.s; PC := PC + 2$ **fi**
end; $\{PC = s + 2\}$
end

$= \{Law\ 2.2.70\} (\text{while elimination})$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $S.s := \langle x \rangle \wedge S.s; PC := PC + 2; [PC = s + 2]; \{PC = s + 2\}$
end

$\sqsubseteq \{Law\ 2.2.62\} (\text{; } [b]; \{b\} \text{ simulation})$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $S.s := \langle x \rangle \wedge S.s; PC := PC + 2$
end

$\sqsubseteq \{Law\ 2.2.11\} (\text{var- := final value}) \text{ and } \{Law\ 2.2.9\} (\text{var-; left dist})$
var $PC : Int \bullet$
 $PC := s; [PC = s];$
end;
 $S.s := \langle x \rangle \wedge S.s$

$\sqsubseteq \{Law\ 2.2.61\} (\text{; := void assertion}) \text{ and } \{Law\ 2.2.11\} (\text{var- := final value})$
var $PC : Int \bullet$ **end;**
 $S.s := \langle x \rangle \wedge S.s$

$$\sqsubseteq \{Law\ 2.2.6\} \text{ (var elim)} \\ S.s := \langle x \rangle \frown S.s$$

$$= \{Definition\ 4.2.1\} \text{ (Abbreviations — operand stack)} \\ Push(S, x)$$

◇

Proof of Rule 4.4.3 (Pop variable):

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

□

$$\begin{aligned} & RHS \\ = & \{Definition\ 4.4.1\} \text{ (Abbreviation for the } PC \text{ frame)} \text{ and} \\ & \{Definition\ 4.4.2\} \text{ (Abbreviations introduced in the Control Elimination phase)} \\ & \text{var } PC : Int \bullet \\ & \quad PC := s; \\ & \quad \text{while } PC \geq s \wedge PC < s + 2 \rightarrow \\ & \quad \quad \text{if } PC = s \rightarrow \\ & \quad \quad \quad x := head(S.s); S.s := tail(S.s); PC := PC + 2 \\ & \quad \quad \text{fi} \\ & \quad \text{end; } \{PC = s + 2\} \\ & \text{end} \\ \\ = & \{Law\ 2.2.61\} (; := \text{void assertion}) \text{ and } \{Law\ 2.2.71\} (\text{while unfold}) \\ & \text{var } PC : Int \bullet \\ & \quad PC := s; [PC = s]; \\ & \quad \text{if } PC = s \rightarrow \\ & \quad \quad x := head(S.s); S.s := tail(S.s); PC := PC + 2 \\ & \quad \text{fi} \\ & \quad \text{while } PC \geq s \wedge PC < s + 2 \rightarrow \\ & \quad \quad \text{if } PC = s \rightarrow \\ & \quad \quad \quad x := head(S.s); S.s := tail(S.s); PC := PC + 2 \\ & \quad \quad \text{fi} \\ & \quad \text{end; } \{PC = s + 2\}; \\ & \text{end} \end{aligned}$$

$\sqsubseteq$ $\{Law\ 2.2.37\}$ (**if selection**) and $\{Law\ 2.2.61\}$ (**; := void assertion**)
var $PC : Int \bullet$
 $PC := s; \ [PC = s];$
 $x := head(S.s); \ S.s := tail(S.s); \ PC := PC + 2; \ [PC = s + 2];$
while $PC \geq s \wedge PC < s + 2 \rightarrow$
 if $PC = s \rightarrow$
 $x := head(S.s); \ S.s := tail(S.s); \ PC := PC + 2$
 fi
end; $\{PC = s + 2\}$
end

$=$ $\{Law\ 2.2.70\}$ (**while elimination**)
var $PC : Int \bullet$
 $PC := s; \ [PC = s];$
 $x := head(S.s); \ S.s := tail(S.s); \ PC := PC + 2;$
 $[PC = s + 2]; \ \{PC = s + 2\}$
end

$\sqsubseteq$ $\{Law\ 2.2.62\}$ (**; [b]; {b} simulation**)
var $PC : Int \bullet$
 $PC := s; \ [PC = s];$
 $x := head(S.s); \ S.s := tail(S.s); \ PC := PC + 2;$
end

$\sqsubseteq$ $\{Law\ 2.2.11\}$ (**var- := final value**) and $\{Law\ 2.2.9\}$ (**var-; left dist**)
var $PC : Int \bullet$
 $PC := s; \ [PC = s];$
end;
 $x := head(S.s); \ S.s := tail(S.s)$

$\sqsubseteq$ $\{Law\ 2.2.61\}$ (**; := void assertion**) and $\{Law\ 2.2.11\}$ (**var- := final value**)
var $PC : Int \bullet$ **end;**
 $x := head(S.s); \ S.s := tail(S.s)$

$\sqsubseteq$ $\{Law\ 2.2.6\}$ (**var elim**)
 $x := head(S.s); \ S.s := tail(S.s)$

$$= \{ \text{Definition 4.2.1} \} \text{ (Abbreviations — operand stack)} \\ \text{Pop}(S, x)$$

◇

Proof of Rule 4.4.8 (Binary Operator):

$$\text{Bop}(S) \sqsubseteq PC : [s, (PC = s \rightarrow \text{Bop}), s + 1]$$

□

$$\begin{aligned} & \text{RHS} \\ = & \{ \text{Definition 4.4.1} \} \text{ (Abbreviation for the PC frame)} \text{ and} \\ & \{ \text{Definition 3.4.4} \} \text{ (Unary and Binary Operators)} \\ & \text{var } PC : \text{Int} \bullet \\ & \quad PC := s; \\ & \quad \text{while } PC \geq s \wedge PC < s + 1 \rightarrow \\ & \quad \quad \text{if } PC = s \rightarrow \\ & \quad \quad \quad S.S := \langle (DataInt \text{ head}(S.s)).Info \text{ bop} \\ & \quad \quad \quad (DataInt \text{ head}(\text{tail}(S.s))).Info \rangle \wedge \text{tail}(\text{tail}(S.s)); \\ & \quad \quad \quad PC := PC + 1 \\ & \quad \quad \text{fi} \\ & \quad \text{end; } \{ PC = s + 1 \} \\ & \text{end} \\ = & \{ \text{Law 2.2.61} \} (; := \text{void assertion}) \text{ and } \{ \text{Law 2.2.71} \} (\text{while unfold}) \\ & \text{var } PC : \text{Int} \bullet \\ & \quad PC := s; [PC = s]; \\ & \quad \text{if } PC = s \rightarrow \\ & \quad \quad S.S := \langle (DataInt \text{ head}(S.s)).Info \text{ bop} \\ & \quad \quad (DataInt \text{ head}(\text{tail}(S.s))).Info \rangle \wedge \text{tail}(\text{tail}(S.s)); \\ & \quad \quad PC := PC + 1 \\ & \quad \text{fi} \\ & \quad \text{while } PC \geq s \wedge PC < s + 1 \rightarrow \\ & \quad \quad \text{if } PC = s \rightarrow \\ & \quad \quad \quad S.S := \langle (DataInt \text{ head}(S.s)).Info \text{ bop} \\ & \quad \quad \quad (DataInt \text{ head}(\text{tail}(S.s))).Info \rangle \wedge \text{tail}(\text{tail}(S.s)); \\ & \quad \quad \quad PC := PC + 1 \\ & \quad \quad \text{fi} \\ & \quad \text{end; } \{ PC = s + 1 \}; \\ & \text{end} \end{aligned}$$

$\sqsubseteq$ {Law 2.2.37} (**if selection**) and {Law 2.2.61} (**;** := **void assertion**)

```

var  $PC : Int \bullet$ 
   $PC := s; [PC = s];$ 
   $S.S := \langle (DataInt\ head(S.s)).Info \mathbf{bop}$ 
     $(DataInt\ head(tail(S.s))).Info \rangle \wedge tail(tail(S.s));$ 
   $PC := PC + 1; [PC = s + 1];$ 
  while  $PC \geq s \wedge PC < s + 1 \rightarrow$ 
    if  $PC = s \rightarrow$ 
       $S.S := \langle (DataInt\ head(S.s)).Info \mathbf{bop}$ 
         $(DataInt\ head(tail(S.s))).Info \rangle \wedge tail(tail(S.s));$ 
       $PC := PC + 1$ 
    fi
  end;  $\{PC = s + 1\}$ 
end

```

$=$ {Law 2.2.70} (**while elimination**)

```

var  $PC : Int \bullet$ 
   $PC := s; [PC = s];$ 
   $S.S := \langle (DataInt\ head(S.s)).Info \mathbf{bop}$ 
     $(DataInt\ head(tail(S.s))).Info \rangle \wedge tail(tail(S.s));$ 
   $PC := PC + 1; [PC = s + 1]; \{PC = s + 1\}$ 
end

```

$=$ {Law 2.2.62} (**;** [b]; $\{b\}$ **simulation**)

```

var  $PC : Int \bullet$ 
   $PC := s; [PC = s];$ 
   $S.S := \langle (DataInt\ head(S.s)).Info \mathbf{bop}$ 
     $(DataInt\ head(tail(S.s))).Info \rangle \wedge tail(tail(S.s));$ 
   $PC := PC + 1$ 
end

```

$=$ {Law 2.2.11} (**var- := final value**) and {Law 2.2.9} (**var-; left dist**)

```

var  $PC : Int \bullet$ 
   $PC := s; [PC = s];$ 
end;
 $S.S := \langle (DataInt\ head(S.s)).Info \mathbf{bop}$ 
   $(DataInt\ head(tail(S.s))).Info \rangle \wedge tail(tail(S.s));$ 

```

$$\begin{aligned}
&= \{Law\ 2.2.61\} (\text{;} := \text{void assertion}) \text{ and } \{Law\ 2.2.11\} (\text{var-} := \text{final value}) \\
&\quad \text{var } PC : Int \bullet \text{end;} \\
&\quad S.S := \langle (DataInt\ head(S.s)).Info \text{ bop} \\
&\quad \quad (DataInt\ head(tail(S.s)).Info) \wedge tail(tail(S.s)); \\
&= \{Law\ 2.2.6\} (\text{var elim}) \\
&\quad S.S := \langle (DataInt\ head(S.s)).Info \text{ bop} \\
&\quad \quad (DataInt\ head(tail(S.s)).Info) \wedge tail(tail(S.s)); \\
&= \{Definition\ 4.2.1\} (\text{Abbreviations} \text{ — operand stack}) \\
&\quad Bop(S)
\end{aligned}$$

◇

Proof of Rule 4.4.9 (Unary Operator):

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

□

Similar to Rule 4.4.8

Proof of Rule 4.4.11 (Iteration):

If PC does not occur in b then

$$\begin{aligned}
&\text{while } b \rightarrow \\
&\quad w : [s + 1, GCS, f_0] \sqsubseteq w : [s, R, f_0 + 1] \\
&\text{end} \\
\text{Where } R &= \left(\begin{array}{l} PC = s \rightarrow (\text{if } b \rightarrow PC := PC + 1 \\ \quad \square \neg(b) \rightarrow PC := f_0 + 1 \\ \quad \text{fi}) \\ \square GCS \square PC = f_0 \rightarrow (PC := s) \end{array} \right)
\end{aligned}$$

□

RHS

$\sqsupseteq$ {*Lemma A.1.6*} **⟨Conditional elimination⟩**
if $b \rightarrow w : [s_1, R, f]$
 $\square \neg(b) \rightarrow w : [f, R, f]$
fi

$\sqsupseteq$ {*Lemma A.1.4*} **⟨Sequential composition⟩** and {*Rule 4.4.1*} **⟨skip⟩**
if $b \rightarrow w : [s_1, b_1 \rightarrow p, f_1]; w : [f_1, R, f];$
 $\square \neg(b) \rightarrow \text{skip}$
fi

$\sqsupseteq$ {*Lemma A.1.7*} **⟨Void initial state⟩**
if $b \rightarrow w : [s_1, b_1 \rightarrow p, f_1]; RHS;$
 $\square \neg(b) \rightarrow \text{skip}$
fi

$\sqsupseteq$ {*Law 2.2.66*} **⟨rec least fixed point⟩**
LHS

◇

Proof of Rule 4.4.12 (Sequential composition):

$$w : [s, GCS_1, j]; w : [j, GCS_2, f] \sqsubseteq w : [s, (GCS_1 \square GCS_2), f]$$

□

RHS

$\sqsupseteq$ {*Lemma A.1.4*} **⟨Sequential composition⟩**
 $w : [s, GCS_1, j]; w : [j, GSC_1 \square GCS_2, f]$
 $\sqsupseteq$ {*Lemma A.1.5*} **⟨Guarded command introduction⟩**
 $w : [s, GCS_1, j]; w : [j, GCS_2, f]$

◇

Proof of Rule 4.4.10 (Conditional):

If PC does not occur in b then

if $b \rightarrow w : [s_1, GCS_1, f_0]$
 $\square \neg(b) \rightarrow w : [s_2, GCS_2, f] \sqsubseteq w : [s, R, f]$
fi

$$\text{Where } R = \left(\begin{array}{l} PC = s \rightarrow (\text{if } b \rightarrow PC := s_1 \square \neg(b) \rightarrow PC := s_2 \text{ fi}) \\ \square GCS_1 \square PC = f_0 \rightarrow (PC := f) \square GCS_1 \end{array} \right)$$

□

RHS

$\sqsubseteq$ {*Lemma A.1.6*} **<Conditional elimination>**
if $b \rightarrow w : [s_1, R, f]$
 $\square \neg(b) \rightarrow w : [s_2, R, f]$
fi

$\sqsubseteq$ {*Lemma A.1.4*} **<Sequential composition>** and
{*Lemma A.1.5*} **<Guarded command introduction>**
if $b \rightarrow w : [s_1, b_1 \rightarrow p, f_1]; w : [f_1, PC = f_1 \rightarrow PC := f, f];$
 $\square \neg(b) \rightarrow w : [s_2, b_2 \rightarrow q, f]$
fi

$\sqsubseteq$ {*Rule 4.4.1*} **<skip>** and {*Law 2.2.46*} (**;- skip unit**)
LHS

◇

Proof of Rule 4.4.4 (Object Creation):

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

□

RHS

$=$ {*Definition 4.4.1*} **(Abbreviation for the PC frame)** and
{*Definition 4.4.2*} **(Abbreviations introduced in the Control Elimination phase)**
var $PC : Int \bullet$
 $PC := s;$
while $PC \geq s \wedge PC < s + 2 \rightarrow$
if $PC = s \rightarrow S.s := \langle \mathbf{new} \ C \rangle \wedge S.s; PC := PC + 2$ **fi**
end; $\{PC = s + 2\}$
end

$=$ {*Law 2.2.61*} (**;- void assertion**) and {*Law 2.2.71*} (**while unfold**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
if $PC = s \rightarrow S.s := \langle \mathbf{new} \ C \rangle \wedge S.s; PC := PC + 2$ **fi**
while $PC \geq s \wedge PC < s + 2 \rightarrow$
if $PC = s \rightarrow S.s := \langle \mathbf{new} \ C \rangle \wedge S.s; PC := PC + 2$ **fi**
end; $\{PC = s + 2\};$
end

$\sqsubseteq$ $\{Law\ 2.2.37\}$ (**if selection**) and $\{Law\ 2.2.61\}$ (**; := void assertion**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $S.s := \langle \mathbf{new}\ C \rangle \wedge S.s; PC := PC + 2; [PC = s + 2];$
while $PC \geq s \wedge PC < s + 2 \rightarrow$
 $\quad \mathbf{if}\ PC = s \rightarrow S.s := \langle \mathbf{new}\ C \rangle \wedge S.s; PC := PC + 2\ \mathbf{fi}$
 $\mathbf{end}; \{PC = s + 2\}$
end

$=$ $\{Law\ 2.2.70\}$ (**while elimination**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $S.s := \langle \mathbf{new}\ C \rangle \wedge S.s; PC := PC + 2; [PC = s + 2]; \{PC = s + 2\}$
end

$=$ $\{Law\ 2.2.62\}$ (**; [b]; {b} simulation**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
 $S.s := \langle \mathbf{new}\ C \rangle \wedge S.s; PC := PC + 2$
end

$=$ $\{Law\ 2.2.11\}$ (**var- := final value**) and $\{Law\ 2.2.9\}$ (**var-; left dist**)
var $PC : Int \bullet$
 $PC := s; [PC = s];$
end;
 $S.s := \langle \mathbf{new}\ C \rangle \wedge S.s$

$=$ $\{Law\ 2.2.61\}$ (**; := void assertion**) and $\{Law\ 2.2.11\}$ (**var- := final value**)
var $PC : Int \bullet \mathbf{end};$
 $S.s := \langle \mathbf{new}\ C \rangle \wedge S.s$

$\sqsubseteq$ $\{Law\ 2.2.6\}$ (**var elim**)
 $S.s := \langle \mathbf{new}\ C \rangle \wedge S.s$

$=$ $\{Definition\ 4.2.1\}$ (**Abbreviations — operand stack**)
 $Push(S, \mathbf{new}\ C)$

◇

Proof of Rule 4.4.5 (Type Test):

4.4.5 If $Is(S, C)$ is not followed by an assignment to PC , then

$$Is(S, C) \quad \sqsubseteq \quad w : [s, (PC = s \rightarrow IsOf(C), s + 2)]$$

□

Similar to Rule 4.4.2

Proof of Rule 4.4.6 (Set an attribute):

If $Putfd(S, C.t)$ is not followed by an assignment to PC , then

$$Putfd(S, C.t) \quad \sqsubseteq \quad w : [s, (PC = s \rightarrow Putfield(C.t), s + 2)]$$

□

Similar to Rule 4.4.2

Proof of Rule 4.4.7 (Get an attribute):

If $Getfd(S, C.t)$ is not followed by an assignment to PC , then

$$Getfd(S, C.t) \quad \sqsubseteq \quad w : [s, (PC = s \rightarrow Getfield(C.t), s + 2)]$$

□

Similar to Rule 4.4.2

Proof of Lemma 4.4.1 (Commands in normal form):

If w and PC are not free in p then

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

□

RHS
 $= \{Definition\ 4.4.1\} \text{ (Abbreviation for the } PC \text{ frame)}, \{Law\ 2.2.61\} (; := \text{void assertion}) ,$
 $\text{and } \{Law\ 2.2.71\} \text{ (while unfold)}$
var $v : T; \ PC : Int \bullet$
 $\quad v := v_0;$
 $\quad PC := s; \ [PC = s];$
 $\quad \text{if } PC = s \rightarrow p; \ PC := f \text{ fi}$
 $\quad \text{while } PC \geq s \wedge PC < f \rightarrow$
 $\quad \quad \text{if } PC = s \rightarrow p; \ PC := f \text{ fi}$
 $\quad \text{end; } \{PC = f\};$
end

$\sqsubseteq \{Law\ 2.2.37\} \text{ (if selection)} \text{ and } \{Law\ 2.2.61\} (; := \text{void assertion})$
var $v : T; \ PC : Int \bullet$
 $\quad v := v_0;$
 $\quad PC := s; \ [PC = s];$
 $\quad p; \ PC := f; \ [PC = f];$
 $\quad \text{while } PC \geq s \wedge PC < f \rightarrow$
 $\quad \quad \text{if } PC = s \rightarrow p; \ PC := f \text{ fi}$
 $\quad \text{end; } \{PC = f\};$
end

$= \{Law\ 2.2.70\} \text{ (while elimination)}$
var $v : T; \ PC : Int \bullet$
 $\quad v := v_0;$
 $\quad PC := s; \ [PC = s];$
 $\quad p; \ PC := f; \ [PC = f];$
 $\quad \{PC = f\};$
end

$\sqsubseteq \{Law\ 2.2.62\} (; [b]; \{b\} \text{ simulation}) , \{Law\ 2.2.61\} (; := \text{void assertion}) , \text{ and}$
 $\{Law\ 2.2.11\} \text{ (var- := final value)}$
var $v : T; \ PC : Int \bullet$
 $\quad v := v_0; \ PC := s; \ p;$
end

$\sqsubseteq \{Law\ 2.2.9\} \text{ (var-; left dist)} , \{Law\ 2.2.11\} \text{ (var- := final value)} , \text{ and}$
 $\{Law\ 2.2.6\} \text{ (var elim)}$
 LHS

◇

Proof of Rule 4.4.16 (Nested normal form) :

If w does not occur free in the left-hand side of the following inequation and r_i is not changed by $v : [s_0, b_0 \rightarrow p, f_0]$ then

$$v : [s, \left(\begin{array}{l} b_1 \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; PC := r_1) \\ \square \dots \\ \square b_n \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; PC := r_n) \\ \square b \rightarrow q \end{array} \right), f] \sqsubseteq v, w : [s, T, f]$$

where

$$T = \left(\begin{array}{l} b_1 \rightarrow (w := r_1; PC := K_{b_0}) \\ \square \dots \\ \square b_n \rightarrow (w := r_n; PC := K_{b_0}) \\ \square f_o \rightarrow PC := w \\ \square b_o \rightarrow p \square b \rightarrow q \end{array} \right)$$

□

$$\begin{aligned} (1) \quad & w := r_i; F := \mathbf{new} \text{ Stack}; S := \mathbf{new} \text{ Stack}; PC := s; \\ & \mathbf{while} \ d \rightarrow \mathbf{if} \ T \mathbf{fi} \mathbf{end} \\ = \quad & \{Law \ 2.2.76\} \ (\mathbf{while} \ \mathbf{sequence}) \ \text{and} \\ & \{Law \ 2.2.73\} \ (\mathbf{while} - \mathbf{if} \ \mathbf{guard} \ \mathbf{elim}) \\ & w := r_i; F := \mathbf{new} \text{ Stack}; S := \mathbf{new} \text{ Stack}; PC := s; \\ & \mathbf{while} \ b_0 \rightarrow \mathbf{if} \ p \mathbf{fi} \mathbf{end}; \mathbf{while} \ d \rightarrow \mathbf{if} \ T \mathbf{fi} \mathbf{end} \\ \sqsubseteq \quad & \{Law \ 2.2.62\} \ \langle \{b\} \sqsubseteq \mathbf{skip} \rangle \ \text{and} \ \{Law \ 2.2.72\} \ (\mathbf{while} - \mathbf{if} \ \mathbf{unfold}) \\ & w := r_i; F := \mathbf{new} \text{ Stack}; S := \mathbf{new} \text{ Stack}; PC := s; \mathbf{while} \ b_0 \rightarrow \mathbf{if} \ p \mathbf{fi} \mathbf{end}; \\ & \{PC = f_o\}; PC := w; \mathbf{while} \ d \rightarrow \mathbf{if} \ T \mathbf{fi} \mathbf{end} \\ = \quad & \{Definition \ 4.4.4\} \ \text{and} \ \{Law \ 2.2.29\} \ (\mathbf{var} \ \mathbf{introduction}) \\ & w := r_i; v : [s_0, b_0 \rightarrow p, f_0]; PC := w; \mathbf{while} \ d \rightarrow \mathbf{if} \ T \mathbf{fi} \mathbf{end} \\ = \quad & \{Law \ 2.2.55\} \ (\mathbf{;} := \mathbf{substitution}) \\ & v : [s_0, b_0 \rightarrow p, f_0]; w := r_i; PC := w; \mathbf{while} \ d \rightarrow \mathbf{if} \ T \mathbf{fi} \mathbf{end} \\ = \quad & \{Law \ 2.2.54\} \ (\mathbf{;} := \mathbf{substitution}) \\ (2) \quad & v : [s_0, b_0 \rightarrow p, f_0]; w := r_i; PC := r_i; \mathbf{while} \ d \rightarrow \mathbf{if} \ T \mathbf{fi} \mathbf{end} \end{aligned}$$

RHS

$\sqsubseteq \{Law\ 2.2.74\}$ (**while – if replace guarded command**) and (1) $\sqsubseteq$ (2)

$$v, w : [s, \left(\begin{array}{l} b_1 \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \ w := r_1; \ PC := r_1) \\ \square \quad \dots \\ \square \quad b_n \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \ w := r_n; \ PC := r_n) \\ \square \quad f_o \rightarrow PC := w \\ \square \quad b \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.40\}$ (**if - Guarded command introduction**)

$$v, w : [s, \left(\begin{array}{l} b_1 \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \ w := r_1; \ PC := r_n) \\ \square \quad \dots \\ \square \quad b_n \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \ w := r_n; \ PC := r_n) \\ \square \quad b \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.10\}$ (**var-; right dist**) and $\{Law\ 2.2.4\}$ (**var association**)

$$v : [s, \left(\begin{array}{l} b_1 \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \ \mathbf{var} \ w : int \bullet w := r_1 \mathbf{end}; \ PC := r_1) \\ \square \quad \dots \\ \square \quad b_n \rightarrow (v : [s_0, b_0 \rightarrow p, f_0]; \ \mathbf{var} \ w : int \bullet w := r_n \mathbf{end}; \ PC := r_n) \\ \square \quad b \rightarrow q \end{array} \right), f]$$

$\sqsubseteq \{Law\ 2.2.6\}$ (**var elim**) and $\{Law\ 2.2.11\}$ (**var- := final value**)

LHS

◇

A.5 Data Refinement — Proofs

Proof Theorem 4.5.2 ($(\hat{\Psi}, \hat{\Psi}^{-1})$ simulation):

$$\hat{\Psi}_w; \hat{\Psi}_w^{-1} \sqsubseteq \text{skip} \sqsubseteq \hat{\Psi}_w^{-1}; \hat{\Psi}_w$$

□

$$\begin{aligned} & \hat{\Psi}_{o,i}; \hat{\Psi}_{o,i}^{-1} \\ \sqsubseteq & \{ \text{Definitions 4.5.1 and 4.5.2} \} \text{ and } \{ \text{Law 2.2.25} \} \\ & \text{dvar } o : C; \ i : \text{Int}; \\ & \quad \text{if } M[\Psi_o] = \text{null} \rightarrow o := \text{null} \\ & \quad \quad \square \neg(M[\Psi_o] = \text{null}) \rightarrow o := \text{new } C; \ o.\text{encode}(M[\Psi_o]) \\ & \quad \text{fi}; \\ & \quad i := (\text{DataInt } M[\Psi_i]).\text{Info} \\ & \quad \text{if } o = \text{null} \rightarrow M[\Psi_o] := \text{null} \\ & \quad \quad \square \neg(o = \text{null}) \rightarrow o.\text{decode}(M[\Psi_o]) \\ & \quad \text{fi}; \\ & \quad M[\Psi_i] := (\text{new } \text{DataInt}; \ \text{Info} : i); \\ & \text{dend } o, i \\ \\ \sqsubseteq & \{ \text{Law 2.2.55} \} (; := \text{commutation}) \text{ and} \\ & \{ \text{Law 2.2.50} \} (; \text{ if left dist}) \\ & \text{dvar } o : C; \ i : \text{Int}; \\ & \quad \text{if } M[\Psi_o] = \text{null} \rightarrow o := \text{null}; \\ & \quad \quad \text{if } o = \text{null} \rightarrow M[\Psi_o] := \text{null} \\ & \quad \quad \quad \square \neg(o = \text{null}) \rightarrow o.\text{decode}(M[\Psi_o]) \\ & \quad \quad \text{fi}; \\ & \quad \quad \square \neg(M[\Psi_o] = \text{null}) \rightarrow o := \text{new } C; \ o.\text{encode}(M[\Psi_o]) \\ & \quad \quad \text{if } o = \text{null} \rightarrow M[\Psi_o] := \text{null} \\ & \quad \quad \quad \square \neg(o = \text{null}) \rightarrow o.\text{decode}(M[\Psi_o]) \\ & \quad \quad \text{fi}; \\ & \quad \text{fi}; \\ & \quad i := (\text{DataInt } M[\Psi_i]).\text{Info} \\ & \quad M[\Psi_i] := (\text{new } \text{DataInt}; \ \text{Info} : i); \\ & \text{dend } o, i \end{aligned}$$

$\sqsubseteq$ $\{Law\ 2.2.54\}$ ($;$:= **substitution**) and
 $\{Law\ 2.2.60\}$ ($;$:= **void assumption**)
dvar $o : C; i : Int;$
if $M[\Psi_o] = \text{null} \rightarrow o := \text{null}; \{o = \text{null}\};$
 $\text{if } o = \text{null} \rightarrow M[\Psi_o] := \text{null}$
 $\square \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o])$
fi;
 $\square \neg(M[\Psi_o] = \text{null}) \rightarrow o := \text{new } C; o.encode(M[\Psi_o]);$
 $\{ \neg(o = \text{null}) \}$
if $o = \text{null} \rightarrow M[\Psi_o] := \text{null}$
 $\square \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o])$
fi;
fi;
 $i := (DataInt\ M[\Psi_i]).Info$
 $M[\Psi_i] := (\text{new } DataInt; Info : i);$
dend o, i

$\sqsubseteq$ $\{Law\ 2.2.1\}$ ($;$:= **skip**),
 $\{Law\ 2.2.23\}$ (**dend-** := **final value**),
 $\{Law\ 2.2.37\}$ (**if selection**), and
 $\{Law\ 2.2.60\}$ ($;$:= **void assumption**)
dvar $o : C; i : Int;$
if $M[\Psi_o] = \text{null} \rightarrow o := \text{null}; M[\Psi_o] := \text{null}$
 $\square \neg(M[\Psi_o] = \text{null}) \rightarrow$
 $o := \text{new } C; o.encode(M[\Psi_o]); o.decode(M[\Psi_o])$
fi;
dend o, i

$\sqsubseteq$
 $\{Law\ 2.2.55\}$ ($;$:= **commutation**),
 $\{Law\ 2.2.1\}$ ($;$:= **skip**), and
 $\{Lemma\ 4.5.1\}$ (*Encode; Decode*)
dvar $o : C; i : Int;$
if $M[\Psi_o] = \text{null} \rightarrow o := \text{null}$
 $\square \neg(M[\Psi_o] = \text{null}) \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
fi;
dend o, i

$= \{Law\ 2.2.22\} \text{ (dvar; dend - [] dist)}$
if $M[\Psi_o] = \text{null} \rightarrow$
 dvar $o : C; \ i : Int;$
 $o := \text{null}$
 dend o, i
 $[] \neg(M[\Psi_o] = \text{null}) \rightarrow$
 dvar $o : C; \ i : Int;$
 $o := \text{new } C; \ o.encode(M[\Psi_o])$
 dend o, i
fi;

$\sqsubseteq \{Law\ 2.2.24\} \text{ (dend- := method call unit), and}$
 $\{Law\ 2.2.23\} \text{ (dend- := final value)}$
if $M[\Psi_o] = \text{null} \rightarrow$
 dvar $o : C; \ i : Int;$
 dend o, i
 $[] \neg(M[\Psi_o] = \text{null}) \rightarrow$
 dvar $o : C; \ i : Int;$
 dend o, i
fi;

$= \{Law\ 2.2.25\} \text{ (dend; dvar simulation)}$
if $M[\Psi_o] = \text{null} \rightarrow \text{skip}$
 $[] \neg(M[\Psi_o] = \text{null}) \rightarrow \text{skip}$
fi;

$= \{Law\ 2.2.34\} \text{ (if void guards)}$
skip

$\sqsubseteq \{Law\ 2.2.34\} \text{ (if void guards)}$
if $o = \text{null} \rightarrow \text{skip}$
 $[] \neg(o = \text{null}) \rightarrow \text{skip}$
fi;

$=$ $\{Law\ 2.2.25\}$ (**dend**; **dvar simulation**)
if $o = \mathbf{null}$ $\rightarrow$
 dvar $M : Seq\ Data$;
 dend M
 $\Box \neg(o = \mathbf{null}) \rightarrow$
 dvar $M : Seq\ Data$;
 dend M
fi;

 $\sqsubseteq$ $\{Law\ 2.2.24\}$ (**dend-** := **method call unit**), *and*
 $\{Law\ 2.2.23\}$ (**dend-** := **final value**)
if $o = \mathbf{null}$ $\rightarrow$
 dvar $M : Seq\ Data$;
 $M[\Psi_o] := \mathbf{null}$;
 dend M
 $\Box \neg(o = \mathbf{null}) \rightarrow$
 dvar $M : Seq\ Data$;
 $o.decode(M[\Psi_o])$;
 dend M
fi;

 $=$ $\{Law\ 2.2.60\}$ (**;** := **void assumption**),
 $\{Law\ 2.2.53\}$ (**;** := **combination**), *and*
 $\{Lemma\ 4.5.1\}$ (*Encode*; *Decode*)
if $o = \mathbf{null}$ $\rightarrow$
 dvar $M : Seq\ Data$;
 $M[\Psi_o] := \mathbf{null}$; $\{M[\Psi_o] = \mathbf{null}\}$;
 dend M
 $\Box \neg(o = \mathbf{null}) \rightarrow$
 dvar $M : Seq\ Data$;
 $o.decode(M[\Psi_o])$; $\{M[\Psi_o] \neq \mathbf{null}\}$;
 $o := \mathbf{new}\ C$; $o.encode(M[\Psi_o])$;
 dend M
fi;

$\sqsubseteq$ $\{Law\ 2.2.52\}$ (**; if selection**),
 $\{Law\ 2.2.54\}$ (**; := substitution**), *and*
 $\{Law\ 2.2.40\}$ (**if - Guarded command introduction**)
if $o = \text{null}$ $\rightarrow$
 dvar $M : Seq\ Data$;
 $M[\Psi_o] := \text{null}; \{M[\Psi_o] = \text{null}\};$
 if $(M[\Psi_o] = \text{null}) \rightarrow o := \text{null}$
 $\Box(M[\Psi_o] \neq \text{null}) \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
 dend M
 $\Box \neg(o = \text{null}) \rightarrow$
 dvar $M : Seq\ Data$;
 $o.decode(M[\Psi_o]); \{M[\Psi_o] \neq \text{null}\};$
 if $(M[\Psi_o] = \text{null}) \rightarrow o := \text{null}$
 $\Box(M[\Psi_o] \neq \text{null}) \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
 dend M
fi;

 $=$ $\{Law\ 2.2.22\}$ (**dvar; dend - $\Box$ dist**) *and*
 $\{Law\ 2.2.60\}$ (**; := void assumption**)
dvar $M : Seq\ Data$;
 if $o = \text{null} \rightarrow M[\Psi_o] := \text{null};$
 if $(M[\Psi_o] = \text{null}) \rightarrow o := \text{null}$
 $\Box(M[\Psi_o] \neq \text{null}) \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
 $\Box \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o]);$
 if $(M[\Psi_o] = \text{null}) \rightarrow o := \text{null}$
 $\Box(M[\Psi_o] \neq \text{null}) \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
 fi;
dend M

= {*Law* 2.2.50} (; **if left dist**)
dvar $M : Seq\ Data$;
 if $o = \text{null}$ $\rightarrow M[\Psi_o] := \text{null}$
 $\Box \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o]);$
 fi;
 if $(M[\Psi_o]) = \text{null}$ $\rightarrow o := \text{null}$
 $\Box (M[\Psi_o]) \neq \text{null} \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
dend M

= {*Law* 2.2.23} (**dend-** := **final value**)
dvar $M : Seq\ Data$;
 if $o = \text{null}$ $\rightarrow M[\Psi_o] := \text{null}$
 $\Box \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o]);$
 fi;
 if $(M[\Psi_o]) = \text{null}$ $\rightarrow o := \text{null}$
 $\Box (M[\Psi_o]) \neq \text{null} \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
 $M[\Psi_i] := (\text{new } DataInt; Info : i);$
dend M

$\sqsubseteq$ {*Law* 2.2.1} (**:= skip**)
dvar $M : Seq\ Data$;
 if $o = \text{null}$ $\rightarrow M[\Psi_o] := \text{null}$
 $\Box \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o]);$
 fi;
 if $(M[\Psi_o]) = \text{null}$ $\rightarrow o := \text{null}$
 $\Box (M[\Psi_o]) \neq \text{null} \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$
 fi
 $M[\Psi_i] := (\text{new } DataInt; Info : i);$
 $i := i$
dend M


```

= {Law 2.2.54} (; := substitution)
  dvar  $M : Seq\ Data$ ;
    if  $o = \text{null}$   $\rightarrow M[\Psi_o] := \text{null}$ 
       $\Box \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o]);$ 
    fi;
    if  $(M[\Psi_o]) = \text{null}$   $\rightarrow o := \text{null}$ 
       $\Box (M[\Psi_o]) \neq \text{null} \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$ 
    fi
     $M[\Psi_i] := (\text{new } DataInt; Info : i);$ 
     $i := (DataInt\ M[\Psi_i]).Info;$ 
  dend  $M$ 

```

```

 $\sqsubseteq$  {Law 2.2.55} (; := commutation) and
{Law 2.2.25} (dend; dvar simulation)
  dvar  $M : Seq\ Data$ ;
    if  $o = \text{null}$   $\rightarrow M[\Psi_o] := \text{null}$ 
       $\Box \neg(o = \text{null}) \rightarrow o.decode(M[\Psi_o]);$ 
    fi;
     $M[\Psi_i] := (\text{new } DataInt; Info : i);$ 
  dend  $o, i$ ;
  dvar  $o : C; i : Int\bullet$ 
    if  $(M[\Psi_o]) = \text{null}$   $\rightarrow o := \text{null}$ 
       $\Box (M[\Psi_o]) \neq \text{null} \rightarrow o := \text{new } C; o.encode(M[\Psi_o])$ 
    fi
     $i := (DataInt\ M[\Psi_i]).Info;$ 
  dend  $M$ 

```

◇

Proof of Rule 4.5.1 (skip - Data Refinement)

$$\hat{\Psi}_w(\text{skip}) \sqsubseteq \text{skip}$$

□

Follows directly from the fact that $\hat{\Psi}$ is a simulation function.

Proof of Rule 4.5.4 (Sequential Composition - Data Refinement):

$$\hat{\Psi}_w(p; q) \sqsubseteq \hat{\Psi}_w(p); \hat{\Psi}_w(q)$$

□

Follows directly from the fact that $\hat{\Psi}$ is a simulation function.

Proof of Rule 4.5.5 (Conditional - Data Refinement):

$$\begin{aligned} & \hat{\Psi}_w \text{ (if } x \rightarrow p \square \neg(x) \rightarrow q \text{ fi)} \\ \sqsubseteq & \text{ if } \hat{\Psi}_w(x) \rightarrow \hat{\Psi}_w(p) \square \hat{\Psi}_w(\neg(x)) \rightarrow \hat{\Psi}_w(q) \text{ fi} \end{aligned}$$

where b is a boolean expression. □

$$\begin{aligned} & LHS \\ = & \{Definition\ 4.5.3\} \text{ (Simulation Function) and} \\ & \{Definition\ 4.5.1\} \text{ (Encoding block } \hat{\Psi}_{o,i}) \\ & \mathbf{dvar}\ o : C; \ i : Int; \\ & \quad \text{if } M[\Psi_o] = \mathbf{null} \rightarrow (o := \mathbf{null}) \\ & \quad \square \neg(M[\Psi_o] = \mathbf{null}) \rightarrow (o := \mathbf{new}\ C; \ o.encode(M[\Psi_o])) \\ & \quad \mathbf{fi} \\ & \quad i := (DataInt\ M[\Psi_i]).Info; \\ & \mathbf{dend}\ M; \\ & \text{if } (x) \rightarrow (p) \\ & \quad \square \neg(x) \rightarrow q \\ & \mathbf{fi} \\ & \hat{\Psi}_w^{-1} \\ = & \{Law\ 2.2.22\} (\mathbf{dvar}; \mathbf{dend} - \square \mathbf{dist}) \\ & \mathbf{dvar}\ o : C; \ i : Int; \\ & \quad \text{if } M[\Psi_o] = \mathbf{null} \rightarrow (o := \mathbf{null}) \\ & \quad \square \neg(M[\Psi_o] = \mathbf{null}) \rightarrow (o := \mathbf{new}\ C; \ o.encode(M[\Psi_o])) \\ & \quad \mathbf{fi} \\ & \quad i := (DataInt\ M[\Psi_i]).Info; \\ & \text{if } (x) \rightarrow (\mathbf{dend}\ M; p) \\ & \quad \square \neg(x) \rightarrow (\mathbf{dend}\ M; q) \\ & \mathbf{fi} \\ & \hat{\Psi}_w^{-1} \end{aligned}$$

$$\begin{aligned}
&= \{Law\ 2.2.56\} \text{ (; := right dist) and} \\
&\quad \{Definition\ 4.5.4\} \text{ (Simulation as substitution)} \\
&\quad \mathbf{dvar}\ o : C; \ i : Int; \\
&\quad \quad \mathbf{if}\ M[\Psi_o] = \mathbf{null} \rightarrow (o := \mathbf{null}) \\
&\quad \quad \square \neg(M[\Psi_o] = \mathbf{null}) \rightarrow (o := \mathbf{new}\ C; \ o.encode(M[\Psi_o])) \\
&\quad \mathbf{fi} \\
&\quad \mathbf{if}\ \hat{\Psi}_w(x) \rightarrow (i := (DataInt\ M[\Psi_i]).Info; \ \mathbf{dend}\ M; \ p) \\
&\quad \quad \square \hat{\Psi}_w(\neg(x)) \rightarrow (i := (DataInt\ M[\Psi_i]).Info; \ \mathbf{dend}\ M; \ q) \\
&\quad \mathbf{fi} \\
&\quad \hat{\Psi}_w^{-1} \\
&= \{Law\ 2.2.51\} \text{ (; if right dist) and} \\
&\quad \{Law\ 2.2.22\} \text{ (dvar; dend - [] dist)} \\
&\quad \mathbf{if}\ \hat{\Psi}_w(x) \rightarrow (\\
&\quad \quad \mathbf{dvar}\ o : C; \ i : Int; \\
&\quad \quad \quad \mathbf{if}\ M[\Psi_o] = \mathbf{null} \rightarrow (o := \mathbf{null}) \\
&\quad \quad \quad \square \neg(M[\Psi_o] = \mathbf{null}) \rightarrow (o := \mathbf{new}\ C; \ o.encode(M[\Psi_o])) \\
&\quad \quad \quad \mathbf{fi} \\
&\quad \quad \quad i := (DataInt\ M[\Psi_i]).Info; \\
&\quad \quad \mathbf{dend}\ M; \ p) \\
&\quad \square \hat{\Psi}_w(\neg(x)) \rightarrow (\\
&\quad \quad \mathbf{dvar}\ o : C; \ i : Int; \\
&\quad \quad \quad \mathbf{if}\ M[\Psi_x] = \mathbf{null} \rightarrow (o := \mathbf{null}) \\
&\quad \quad \quad \square \neg(M[\Psi_o] = \mathbf{null}) \rightarrow (o := \mathbf{new}\ C; \ o.encode(M[\Psi_o])) \\
&\quad \quad \quad \mathbf{fi} \\
&\quad \quad \quad i := (DataInt\ M[\Psi_i]).Info; \\
&\quad \quad \mathbf{dend}\ M; \ q) \\
&\quad \mathbf{fi} \\
&\quad \hat{\Psi}_w^{-1} \\
&= \{Law\ 2.2.50\} \text{ (; if left dist)} \\
&\quad \mathbf{if}\ \hat{\Psi}_w(x) \rightarrow (\hat{\Psi}_w(p)) \\
&\quad \quad \square \hat{\Psi}_w(\neg(x))(\hat{\Psi}_w(q)) \\
&\quad \mathbf{fi}
\end{aligned}$$

◇

Proof of Rule 4.5.6 (Load variable - Data Refinement):

$$\hat{\Psi}_w(Load_{CE}(x)) \sqsubseteq Load(\Psi_x)$$

□

$$\begin{aligned}
& \hat{\Psi} (Load(x)) \\
= & \{Definition\ 4.5.3\} (SimulationFunction) \\
& \hat{\Psi}_w; \ Load_{CE}(x); \ \hat{\Psi}_w^{-1} \\
\\
= & \{Definition\ 4.2.1\} (\textbf{Abbreviations} \text{ --- operand stack}) \\
& \{Definition\ 4.4.2\} \textbf{Abbreviations introduced in the Control Elimination phase} \\
& \hat{\Psi}_w; \ S.s := \langle x \rangle \wedge S.s; \ PC := PC + 2; \ \hat{\Psi}_w^{-1} \\
\\
= & \{Lemma\ 4.5.3\} (\textbf{Assignment - } \hat{\Psi}_w \text{ - Data Refinement}) \\
& S.s := \langle \hat{\Psi}_w(x) \rangle \wedge S.s; \ PC := PC + 2 \\
\\
= & \{Definition\ 4.5.4\} (\textbf{Simulation as substitution}) \\
& S := \langle M[\Psi_x] \rangle \wedge S; \ PC := PC + 2 \\
\\
= & \{Definition\ 3.4.3\} (\textbf{Pushing a value onto the operand stack}) \\
& Load(\Psi_x)
\end{aligned}$$

◇

Proof of Rule 4.5.7 (Store variable - Data Refinement):

$$\hat{\Psi}_w (Store_{CE}(x)) \sqsubseteq Store(\Psi_x)$$

□

Similar to Rule 4.5.6.

Proof of Rule 4.5.8 (Load Constant - Data Refinement):

$$\hat{\Psi}_w (Load_{CE}(a)) \sqsubseteq Ldc(\Phi_a)$$

□

Similar to Rule 4.5.6.

Proof of Rule 4.5.9 (Object Creation - Data Refinement):

$$\hat{\Psi}_w (Load_{CE}(\textbf{new } C)) \sqsubseteq new(\Phi_C)$$

□

$$\begin{aligned}
& \hat{\Psi} (Load(x)) \\
= & \{Definition\ 4.5.3\} (SimulationFunction) \\
& \hat{\Psi}_w; \ Load_{CE}(\mathbf{new}\ C); \ \hat{\Psi}_w^{-1} \\
\\
= & \{Definition\ 4.2.1\} (\mathbf{Abbreviations} \text{ --- operand stack}) \\
& \{Definition\ 4.4.2\} \mathbf{Abbreviations introduced in the Control Elimination phase} \\
& \hat{\Psi}_w; \ S.s := \langle \mathbf{new}\ C \rangle \wedge S.s; \ PC := PC + 2; \ \hat{\Psi}_w^{-1} \\
\\
= & \{Lemma\ 4.5.3\} (\mathbf{Assignment - } \hat{\Psi}_w \text{ - Data Refinement}) \\
& S.s := \langle \hat{\Psi}_w (\mathbf{new}\ C) \rangle \wedge S.s; \ PC := PC + 2 \\
\\
= & \{Definition\ 4.5.4\} (\mathbf{Simulation as substitution}) \\
& S := \langle CP[\Phi_C] \rangle \wedge S; \ PC := PC + 2 \\
\\
= & \{Definition\ 3.4.6\} (\mathbf{Object creation and manipulation}) \\
& new(\Phi_C)
\end{aligned}$$

◇

Bibliography

- [1] Ecma Standardization - Original Submission. Web pages at <http://msdn.microsoft.com/net/ecma/OctoberSubmission.asp>, November 2002.
- [2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 1985.
- [4] R. Back. *On the Correctness of Refinement in Program Development*. PhD thesis, Departament of Computer Science, University of Helsinki, 1978.
- [5] R. Back. *Correct Preserving Program Refinements: Proof Theory and Application*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.
- [6] R. Back. On Correct Refinement of Programs. *Journal of Computer and Systems Science*, 23(1):51–63, 1981.
- [7] R. Back. Procedural Abstraction in the Refinement Calculus. Technical Report Ser. A No. 55, Department of Computer Science, Abo - Finland, 1987.
- [8] R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.
- [9] R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Text in Computer Science. Springer-Verlag, 1998.
- [10] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 1961.
- [11] P. Borba and A. Sampaio. Basic Laws of ROOL. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil, 2000. <http://www.cin.ufpe.br/~lmf/coop/papers>.

- [12] P. Borba and A. Sampaio. Basic Laws of ROOL: an Object-Oriented Language. *Revista de Informática Teórica e Aplicada*, 7(I):49–68, 2000.
- [13] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, January 2004.
- [14] P. Borba, A. Sampaio, and M. Cornélio. A Refinement Algebra for Object-Oriented Programming. In *To Appear in the Proceedings of ECOOP 2003*, 2003.
- [15] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *MFCS'98.*, number 1450, pages 17–35. Springer LNCS, 1998.
- [16] K. B. Bruce and B. C. Pierce, editors. *Foundations of Object-Oriented Languages*. (Informal Workshop Record of FOOL 5), January 1998.
- [17] R. Burstall and J. Goguen. The Semantics of Clear, a Specification Language. In *Advanced Course on Abstract Software Specifications*, number 86 in LNCS, pages 292–332, Copenhagen, 1980. Springer-Verlag.
- [18] R. Burstall and P. Landin. Programs and their Proofs: an Algebraic Approach. *Machine Intelligence*, (7):17–43, 1969.
- [19] A. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for Refinement of Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 26(08):713–728, 2000.
- [20] A. Cavalcanti, A. Sampaio, and J. Woodcock. An Inconsistency in Procedures, Parameters and Substitution in the Refinement Calculus. *Science of Computer Programming*, 33(1):87–96, 1999.
- [21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 2001.
- [22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.
- [23] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a Formal Meta-Tool. In *World Congress on Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

- [24] M. Cornélio. *Applying Object-Oriented Refactoring and Patterns as Formal Refinements*. PhD thesis, Universidade Federal de Pernambuco, 2004.
- [25] M. Cornélio, A. Cavalcanti, and A. Sampaio. Refactoring by Transformation. In *Proceedings of REFINE'2002, Electronic Notes in Theoretical Computer Science*, 2002.
- [26] M. Cornélio, A. Cavalcanti, A. Sampaio, and P. Borba. Proving the Basic Laws of ROOL in a Weakest Precondition Semantics. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil, 2000. <http://www.cin.ufpe.br/~lmf/coop/papers>.
- [27] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
- [28] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press, London, 1972.
- [29] O. J. Dahl and C. A. R. Hoare. Hierarchical Program Structures. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 175–220. Academic Press, London, 1972.
- [30] B. A. Davey and H. A. Priestley. *Introduction to Lattice and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.
- [31] E. W. Dijkstra. *Notes on Structured Programming*. Academic Press, Structured Programming edition, 1972.
- [32] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Engewood Cliffs, 1976.
- [33] S. Drossopoulou, F. Damiani, M. Dzani-Ciancaglini, and P. Giannini. More Dynamic Object Reclassification: Fickle II. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, March 2002.
- [34] 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. Lecture Notes in Computer Science, Springer-Verlag.

- [35] 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. Lecture Notes in Computer Science, Springer-Verlag.
- [36] A. Duran, A. Sampaio, and A. Cavalcanti. Formal Bytecode Generation for ROOL Virtual Machine. In *IV WMF— Workshop on Formal Methods*. PUC—Rio de Janeiro/Brazil, October 2001.
- [37] C. A. R. Hoare et al. Laws of Programming. *Communications of the ACM*, 30(8):672–686, August 1987.
- [38] K. Fisher and J. C. Mitchell. The Development of Type Systems for Object-Oriented Languages. *Theory and Practice of Object Systems*, 1(3), 1995.
- [39] R. W. Floyd. Assigning Meaning to Programs. In *Symposium on Applied Mathematics*, volume 1, pages 19–32. American Mathematical Society, 1967.
- [40] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [42] P. H. B. Gardiner and P. K. Pandya. Reasoning Algebraically about Recursion. *Sci. Comput. Program.*, 18(3):271–280, 1992.
- [43] J. Goguen, J. A. Thatcher, and E. G. Wagner. An Initial Algebra Approach to the Specification Correctness, and Implementation of Abstract Data Types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 49–189, Englewood Cliffs, NJ, 1978. Prentice-Hall.
- [44] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
- [45] J. Hartmanis and R. E. Stearns. Pair Algebras and their Application to Automata Theory. *Information and Control*, 4(7):465–507, 1964.
- [46] I. J. Hayes. Separating Timing and Calculation in Real-Time Refinement. In J. Gundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop and Formal Methods Pacific ’98*. Discrete Mathematics and Theoretical Computer Science, Springer-Verlag, 1998.

- [47] I. J. Hayes and M. Utting. Deadlines are Termination. In D. Gries and W.-P de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET'98)*, pages 186–204. Chapman and Hall, 1998.
- [48] J. He. Hybrid Parallel Programming and Implementation of Synchronised Communication. In *MFCS*, pages 537–546, 1993.
- [49] E. C. R. Hehner. *The Logic of Programming*. International Series in Computer Science. Prentice-Hall, 1984.
- [50] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of ACM*, 12(10), October 1969.
- [51] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, (1):271–281, 1972.
- [52] C. A. R. Hoare and J. He. The Weakest Prespecification. *Information Processing Letters*, (24):127–132, January 1987.
- [53] C. A. R. Hoare, J. He, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.
- [54] H. Lam and T. Thai. *.NET Framework Essentials*. O'Reilly & Associates, 2001.
- [55] J. Lambek. The Mathematics of Sentence Structure. *The American Mathematical Monthly*, (65):154–170, 1958.
- [56] K. Lano. *Formal Object-Oriented Development*. Springer, 1995.
- [57] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. Prentice Hall, 1994.
- [58] G. T. Leavens and W. E. Weihl. Specification and Verification of Object-Oriented Programs using Supertype Abstraction. *Acta Informatica*, (32), 1995.
- [59] K. Lermer and C. Fidge. A Formal Model of Real-Time Program Compilation. *Lecture Notes in Computer Science*, 1601:192–210, 1999.
- [60] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
- [61] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6), 1994.
- [62] B. H. Liskov and S. N. Zilles. Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering*, 1(1):7–19, 1975.

- [63] T. Maibaum, M. Sadler, and P. Veloso. *Logical Specication and Implementation, Foundations of Software Technology and Theoretical Computer Science*. LNCS 181. Springer-Verlag, 1984.
- [64] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In *Symposium on Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.
- [65] A Melton, D A Schmidt, and G E Strecker. Galois Connections and Computer Science Applications. In *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming*, pages 299–312. Springer-Verlag New York, Inc., 1986.
- [66] J. Meseguer. Software Specification and Verification in Rewriting Logic. NATO Advanced Study Institute, Marktoberdorf, Germany, 2003.
- [67] A. Mikhajlova and E. Sererinski. Class Refinement and Interface Refinement in Object-Oriented Programs. In *Proceedings of FME’97: Industrial Benefit of Formal Methods*, Springer-Verlag, 1997.
- [68] C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.
- [69] F. Morris. Advice on Structuring Compilers and Prove them Correct. In *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 144–152. ACM, 1973.
- [70] L. M. Morris. The Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
- [71] P. D. Mosses. *SIS: A Compiler Generator System using Denotational Semantics*. Reference manual, University of Aarhus, 1978.
- [72] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS*. Springer-Verlag, Heidelberg, Germany, 1997.
- [73] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized Types for Java. In ACM Press, editor, *POPL*, pages 132–45, 1997.
- [74] P. Naur. Proof of Algorithms by General Snapshots. *BIT*, 6:310–316, 1966.
- [75] G. Nelson. A Generalization of Dijkstra’s Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

- [76] T. Nipkow and D. Oheimb. Java-light is type-safe — Definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, 1998. ACM Press.
- [77] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. pages 146–159. POPL, ACM Press, 1997.
- [78] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [79] O. Ore. Galois Connections. *Transactions of the American Mathematical Society*, (55):493–513, 1944.
- [80] J. Palsberg. An Automatically Generated and Provably Correct Compiler for a Subset of Ada. In *Proceedings of the fourth IEEE International Conference on Computer Languages*, San Francisco, CA, 1992.
- [81] L. C. Paulson. *A Compiler Generator for Semantic Grammars*. PhD thesis, Stanford University, 1981.
- [82] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2001.
- [83] W. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
- [84] G. J. Popek and et. al. Notes on the Design of EUCLID. *SIGPLAN Notice*, 3(12):11–18, 1977.
- [85] M. R. Raskovsky. Denotational Semantics as a Specification of Code Generators. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 230–244. ACM Press, 1982.
- [86] A. W. Roscoe and C. A. R. Hoare. The Laws of Occam Programming. *Theoretical Computer Science*, 60(2):177–229, 1988.
- [87] A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.
- [88] J. Schmid. Executing ASM Specifications with AsmGofer. Web pages at <http://www.tydo.de/AsmGofer>, 1999.
- [89] M. Serrano. Wide Classes. *Lecture Notes in Computer Science*, 1628:391–415, 1999.

- [90] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.
- [91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.
- [92] A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. *Pacific Journal of Mathematics*, 5, 1955.
- [93] 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.
- [94] M. Wand. A Semantic Prototyping System. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 213–221. ACM Press, 1984.
- [95] G. Watson. Compilation by Refinement for a Practical Assembly Language. In Jin Song Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering*, volume 2885 of *LNCS*, pages 286–305, Singapore, November 2003. 5th International Conference on Formal Engineering Methods, ICFEM 2003, Springer-Verlag.
- [96] L. Wildman. A Formal Basis for a Program Compilation Proof Tool. In Lars Henrik Eriksson and Peter Alexander Lindsay, editors, *FME2002: Formal Methods – Getting IT Right*, Copenhagen, Denmark, July 2002. International Symposium of Formal Methods Europe, Springer.
- [97] M. Wirsing. Structured Algebraic Specifications: a Kernel Language. *Theoretical Computer Science*, 42:123–249, 1986.
- [98] M. Wirsing and M. Broy. A Modular Framework for Specification and Implementation. In *TAPSOFT’89*, number 351 in *LNCS*, 1989.
- [99] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, (14):221–227, 1971.

Tese de Doutorado apresentada por **Adolfo Almeida Duran** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**An Algebraic Approach to the Design of Compilers for Object-Oriented Languages**", elaborada sob a orientação do **Prof. Augusto César Alves Sampaio** e aprovada pela Banca Examinadora formada pelos professores:

Prof. **André Luis de Medeiros Santos**
Departamento de Sistemas de Computação - CIn / UFPE

Prof. **Paulo Henrique Monteiro Borba**
Departamento de Sistemas de Computação - CIn / UFPE

Prof. **Hermano Perrelli de Moura**
Departamento de Informação e Sistemas - CIn / UFPE

Prof. **Arnaldo Vieira Moura**
Instituto de Computação / UNICAMP

Prof. **Edward Herman Hausler**
Departamento de Informática / PUC-RJ

Visto e permitida a impressão.
Recife, 4 de março de 2005.

Prof. Jaelson Freire Brelaz de Castro

Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.