AN ALGEBRAIC APPROACH TO COMPILER DESIGN

by

Augusto Sampaio

Technical Monograph PRG-110
ISBN 0 902928 87 2

October 1993

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD

Copyright © 1993 Augusto Sampaio

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

An Algebraic Approach to Compiler Design

Augusto Sampaio

Wolfson College

A thesis submitted for the degree of Doctor of Philosophy at the University of Oxford, Trinity Term, 1993.

Abstract

The general purpose of this thesis is to investigate the design of compilers for procedural languages, based on the algebraic laws which these languages satisfy. The particular strategy adopted is to reduce an arbitrary source program to a normal form which describes precisely the behaviour of the target machine. This is achieved by a series of algebraic transformations which are proved from the more basic laws. The correctness of the compiler follows from the correctness of each algebraic transformation.

The entire process is formalised within this single and uniform semantic framework of a procedural language and its algebraic laws. But in order to attain abstraction and more reasoning power, the language comprises specification features, in addition to the programming constructs to be implemented. These features are used to define a very general normal form, capable of representing an arbitrary target machine. The normal form reduction theorems can therefore be reused in the design of compilers which produce code for distinct target machines.

A central notion is an ordering relation on programs: $p \sqsubseteq q$ means that q is at least as good as p in the sense that it will meet every purpose and satisfy every specification satisfied by p. Furthermore, substitution of q for p in any context is an improvement (or at least will leave things unchanged when q is semantically equivalent to p). The compiling task is thereby simplified to finding a normal form that is not just the same but is allowed to be better than the original source code. Moreover, at all intermediate stages, we can postpone details of implementation until the most appropriate time to make the relevant design decisions.

Dijkstra's guarded command language is used to illustrate how a complete compiler can be developed, up to the level of instructions for a simple target machine. Each feature of the language is dealt with by a separate reduction theorem, which is the basis for the modularity of the approach. We then show how more elaborate features such as procedures and recursion can be handled in complete isolation from the simpler features of the language. Although the emphasis is on the compilation of control structures, we develop a scheme for compiling arrays. This is also an incremental extension in the sense that it has no effect on the features already implemented.

A large subset of the theory is mechanised as a collection of algebraic structures in the OBJ3 term rewriting system. The reduction theorems are used as rewrite rules to carry out compilation automatically. The overall structure of the original theory is preserved in the mechanisation. This is largely due to the powerful module system of OBJ3.

To

Claudia & Gabi

Acknowledgements

I wish to thank my supervisor, Prof. C.A.R Hoare, whose constant teaching, advice and encouragement have helped me greatly. He has originated the approach to compilation which is investigated here, and has suggested the project which gave rise to this thesis.

No less help I had from He Jifeng, who made himself always available and acted as my second supervisor. Most of what is reported in chapters 3 and 4 resulted from joint work with Prof. Hoare and He Jifeng.

My examiners, Prof. Mathai Joseph and Bernard Sufrin, contributed comments, suggestions and corrections which helped to improve this thesis.

The work of Ralph Back, Carroll Morgan, Joseph Morris and Greg Nelson has been a great source of inspiration.

Many thanks are due to the Declarative group led by Prof. Joseph Goguen. Apart from OBJ, I have learned much about algebra and theorem proving attending their group meetings. The discussions about OBJ with Adolfo Socorro, Paulo Borba, Grant Malcolm and Andrew Stevens were extremely helpful for the mechanisation reported in Chapter 6.

I thank David Naumann, Adolfo Socorro and Ignacio Trejos for pointing out errors and obscurities in an early draft of this thesis, and for many useful comments and suggestions. I am also grateful to Ignacio Trejos for constantly drawing my attention to relevant bibliography, and for the discussions about compilation and the refinement calculus.

My former Msc supervisor, Silvio Meira, introduced me to the field of formal methods. I thank him for the permanent encouragement, and for influencing me to come to Oxford.

I am grateful to my colleagues in the Attic for making it a pleasant working environment, especially to Nacho, for his company during the many nights we have spent there, and to Gavin, for his patience to answer so many questions about English.

I tbank people at the Wolfson College for their support and friendship, especially to the senior tutors (John Penney and Roger Hall) for help with the Day Nursery fees.

Some very good friends have been responsible for the best moments I have had in England. Special thanks go to Adolfo, Tetete, Nacho, George, Hermano, Max and Robin & Chris. I also thank Adolfo and Tetete for the baby-sitting.

I thank my parents, my parents-in-law, my brothers, my sisters-in-law and Daia for their continued support which can never be fully acknowledged. The weekly letters from my mother made me feel not so far away from home.

The most special thanks go to Claudia, for all the encouragement, dedication and patience, but above all for providing me with the most essential ingredient of life—love! My little daughter Gabi has constantly distracted me from work, demanding the attention of another full-time job—but one of enjoyment and happiness, which brought lots of meaning into my life.

Financial support for this work was provided by the Brazilian Research Council, CNPq.

Contents

1 Introduction 1

1.1 The Approach 3

1.2 Overview of Subsequent Chapters 6

2 Background 8

2.1 Partial Orders and Lattices 9

2.2 The Lattice of Predicates 10

2.3 The Lattice of Predicate Transformers 10

2.4 Properties of Predicate Transformers 11

2.5 Some Refinement Calculi 12

2.6 Data Refinement 13

2.7 Refinement and Compilation 16

3 The Reasoning Language 18

3.1 Concepts and Notation 19

3.2 Skip, Abort and Miracle 21

3.3 Sequential Composition 21

3.4 Demonic Nondeterminism 22

3.5 Angelic Nondeterminism 23

3.6 The Ordering Relation 23

3.7 Unbounded Nondeterminism 24

3.8 Recursion 26

3.9 Approximate Inverses 26

3.10 Simulation 28

3.11 Assumption and Assertion 30

3.12 Guarded Command 31
3.13 Guarded Command Set 32
3.14 Conditional 33
3.15 Assignment 34
3.16 Generalised Assignment 35
3.17 Iteration 37
3.18 Static Declaration 40
3.19 Dynamic Declaration 43
3.20 The Correctness of the Basic Laws 47

4 A Simple Compiler **49**

4.1 The Normal Form 50
4.2 Normal Form Reduction 51
4.3 The Target Machine 56
4.4 Simplification of Expressions 57
4.5 Control Elimination 60
4.6 Data Refinement 63
4.7 The Compilation Process 68

5 Procedures, Recursion and Parameters **71**

5.1 Notation 71
5.2 Procedures 73
5.3 Recursion 75
5.4 Parameterised Programs 78
5.5 Parameterised Procedures 82
5.6 Parameterised Recursion 83
5.7 Discussion 84

6 Machine Support **86**

6.1 OBJ3 87
6.2 Structure of the Specification 89
6.3 The Reasoning Language 91
6.3.1 An Example of a Proof 95

6.4 The Normal Form 97
6.5 A Compiler Prototype 98
6.5.1 Simplification of Expressions 98
6.5.2 Control Elimination 99
6.5.3 Data Refinement 100
6.5.4 Machine Instructions 101
6.5.5 Compiling with Theorems 102
6.6 Final Considerations 104
6.6.1 The Mechanisation 105
6.6.2 OBJ3 and 2OBJ 106
6.6.3 Other systems 108

7 Conclusions **110**
7.1 Related Work 112
7.2 Future Work 115
7.3 A Critical View 121

Bibliography **122**

A A Scheme for Compiling Arrays **128**

B Proof of Lemma 5.1 **131**

C Specification and Verification in OBJ3 **135**
C.1 The Reasoning Language 135
C.1.1 Proof of Theorem 3.17.6 139
C.1.2 Proof of Theorem 3.17.7 141
C.2 The Normal Form 144
C.2.1 Proof of Lemma 4.2 145
C.2.2 Proof of Lemma 4.3 146
C.2.3 Proof of Theorem 4.4 146
C.3 Simplification of Expressions 148
C.3.1 Proof of Theorem 4.3 148
C.3.2 Proof of Theorem 4.5 149

C.4 Control Elimination 151
C.5 Data Refinement 152
C.5.1 Proof of Theorem 4.10 152
C.5.2 Proof of Theorem 4.14 154
C.6 Machine Instructions 157

Chapter 1

Introduction

> We must not lose sight of the ultimate goal, that of the construction of programming language implementations which are known to be correct by virtue of the quality of the logical reasoning which has been devoted to them. Of course, such an implementation will still need to be comprehensively tested before delivery; but it will immediately pass all the tests, and then continue to work correctly for the benefit of programmers forever after.
>
> — C.A.R. Hoare

The development of reliable systems has been one of the main challenges to computing scientists. This is a consequence of the complexity of the development process, which usually comprises many stages, from the original capture of requirements to the hardware in which programs will run.

Many theories, methods, techniques and tools have been developed to deal with *adjacent* stages. For example, the derivation of programs from specifications; the translation of programs into machine code (compilation); and sometimes even a gate-level implementation of the hardware. Nevertheless, the development of a mathematical model to ensure global consistency of the process in general has attracted very little attention; most certainly because of the intricacy of the task.

Perhaps the most significant effort in this direction is the work of a group at Computational Logic, Inc. [7]. They suggest an operational approach to the development and mechanical verification of systems. The approach has been applied to many system components, including a compiler, a link-assembler and a gate-level design of a microprocessor. The approach is independent of any particular component and deals with the fundamental aspect of integration of components to form a verified *stack*.

This work inspired a European effort which gave rise to the Esprit project Provably Correct System (ProCoS) [9, 12]. This project also aims to cover the entire development process. The emphasis is on a constructive approach to correctness, using provably correct transformations between all the phases. It differs mostly from the previously cited work in

that an abstract (rather than operational) *universal* model is being developed to ensure consistency across all the interfaces between the development phases. Furthermore, a more ambitious scope is attempted, including explicit parallelism and time constraints throughout the development. The reusability of designs and proofs is also an objective of the project.

Our work is in the context of ProCoS. We are concerned with the compilation phase: the translation of programs to machine code. But the emphasis is on an algebraic approach to compilation, rather than in the translation between a particular pair of languages. Although neither the source language nor the target machine we use as examples coincide with the ones of the ProCoS project, it is hoped that the overall strategy suggested here will be useful to design the compiler required in the project, and many others.

A large number of approaches have been suggested to tackle the problem of compiler correctness. They differ both in the semantic style adopted to define source and target languages (operational, denotational, algebraic, axiomatic, attribute grammars, ...) and in the meaning of correctness associated with the translation process. The first attempt was undertaken by McCarthy and Painter [52]; they used operational semantics to prove the correctness of a compiler for a simple expression language. The algebraic approach originates with the work of Burstall and Landin [14]. Both works have been of great impact and many researchers have built upon them. A brief summary of some of the main approaches to compiler correctness is included in the final chapter.

Here we further develop the approach introduced in [45], where compilation is identified with the reduction of programs (written in a procedural language) to a normal form which describes precisely the behaviour of the target executing mechanism. The reduction process entails a series of semantic-preserving transformations; these are proved from more basic laws (axioms) which give an algebraic semantics to the language.

A central notion is an ordering relation on programs: $p \sqsubseteq q$ means that q is at least as good as p in the sense that it will meet every purpose and satisfy every specification satisfied by p. Furthermore, substitution of q for p in any context is an improvement (or at least will leave things unchanged when q is semantically equivalent to p). The compiling task is thereby simplified to finding a normal form that is not just the same but is allowed to be better than the original source code. Moreover, at all intermediate stages, we can postpone details of implementation until the most appropriate time to make the relevant design decisions.

Another essential feature of the approach is the embedding of the programming language into a more general space of specifications. This includes constructions to model features such as assumptions and assertions, and even the less familiar concept of a *miracle*, standing for an unsatisfiable specification. The specification space forms a complete distributive lattice[1] which allows us to define approximate inverses (Galois connections) for programming operators. The purpose of all of this is to achieve a very abstract normal form definition (which can model an arbitrary executing mechanism) and simple and elegant proofs. We hope to prove this claim.

[1] A brief overview of the lattice theory relevant to us is given in the next chapter.

All the calculations necessary to assert the correctness of the compilation process are carried out within this single framework of a specification language whose semantics is given by algebraic laws. No additional mathematical theory of source or target language is developed or used in the process. The relatively simple reasoning framework, its abstraction and modularity are the main features of the approach.

We select a small programming language (including iteration) to illustrate how a complete compiler[2] can he developed, up to the level of instructions for an abstract machine. Each feature of the language is dealt with by a separate reduction theorem, which is tbe basis for the modularity of the approach. We then show how more elaborate features such as procedures and recursion can be handled, in complete isolation from the simpler features of the language.

A large subset of the theory is mechanised as a collection of algebraic structures in the OBJ3 [31] term rewriting system. The reduction theorems are used as rewrite rules to carry out compilation automatically. The overall structure of the original theory is preserved in the mechanisation. This is largely due to the powerful module system of OBJ3.

1.1 The Approach

In this section we give an overview of our approach to compilation based on a simple example. We also identify the scope of the thesis in terms of the source language that we deal with, the additional specification features and the target machine used to illustrate an application of the approach.

The source programming language contains the following constructions:

skip	do nothing
$x := e$	assignment
$p;\ q$	sequential composition
$p \sqcap q$	nondeterminism (demonic)
$p \lhd b \rhd q$	conditional: if b then p else q
$b * p$	iteration: while b do p
dec $x \bullet p$	(static) declaration of variable x for use in program p
proc $X \mathrel{\widehat=} p \bullet q$	procedure X with hody p and scope q
$\mu\, X \bullet p$	recursive program X with body p

We avoid defining a syntax for expressions; we use **uop** and **bop** to stand for arbitrary unary and binary operators, respectively. Initially, we deal with a simplified version of the

[2] We address only the code generation phase of a compiler. Parsing and semantic analysis are not dealt with. Optimisation is hriefly considered as a topic for future research.

language, not including procedures or recursion. These are treated later, together with the issue of parameterisation.

The programming language is embedded in a specification space including:

$\top$	miracle
$\bot$	abort
$p \sqcup q$	nondeterminism (angelic)
$b_\bot$	assertion: if b then skip else $\bot$
$b^\top$	assumption: if b then skip else $\top$
$b \rightarrow p$	guarded command: if b then p else $\top$
$x :\in b$	generalised assignment: assign a value to x which makes b true; if not possible, $x :\in b$ behaves like $\top$
var x	declaration of variable x with undetermined (dynamic) scope
end x	end the previous (dynamic) scope of x introduced by a var x

Furthermore, while the source language allows only single assignments, the specification language allows multiple assignments of the form

$$x_1, \ldots, x_n := e_1, \ldots, e_n$$

Although some of the above constructs are not strictly necessary, as they can be defined in terms of others, each one represents a helpful concept both for specification and for reasoning.

There may seem to be unnecessary redundancy concerning the notation for variable declarations. dec is the usual construct available in most programming languages for introducing local variables with a lexical (or static) scope semantics. For reasoning purposes which will be explained later, it is useful to have independent constructs to introduce a variable and to end its scope. Operationally, one can think of var x as pushing the current value of x into an implicit stack, and assigning to x an arbitrary value; end x pops the stack and assigns the popped value to x. If the stack was empty, this value is arbitrary.

The semantics of the specification (reasoning) language is given by algebraic laws in the form of equations and inequations. The latter uses the refinement relation discussed in the previous section. As an example, a few algebraic laws are given below. They describe the fact that $\sqsubseteq$ is a lattice ordering. For all programs p, q and r we have:

$p \sqsubseteq \top$	(miracle is the top of the lattice)
$\bot \sqsubseteq p$	(abort is the bottom)
$(r \sqsubseteq p \wedge r \sqsubseteq q) \equiv r \sqsubseteq (p \sqcap q)$	($\sqcap$ is the greatest lower bound)
$(p \sqsubseteq r) \wedge (q \sqsubseteq r) \equiv (p \sqcup q) \sqsubseteq r$	($\sqcup$ is the least upper bound)

We define a simple target machine to illustrate the design of a compiler. It consists of four components:

P	a sequential register (program counter)
A	a general purpose register
M	a store for variables (RAM)
m	a store for instructions (ROM)

The instructions can be designed in the usual way, as assignments tbat update the machine state; for example,

$$\mathsf{load}(n) \stackrel{def}{=} \mathsf{A}, \mathsf{P} := \mathsf{M}[n], \mathsf{P}+1,$$
$$\mathsf{store}(n) \stackrel{def}{=} \mathsf{M}, \mathsf{P} := (\mathsf{M} \oplus \{n \mapsto \mathsf{A}\}), \mathsf{P}+1$$

where we use map overriding ($\oplus$) to update M at position n with the value of A. The more conventional notation is $\mathsf{M}[n] := \mathsf{A}$.

The normal form describing the behaviour of this machine (executing a stored program) is an iterated execution of instructions taken from the store m at location P:

$$\mathsf{dec}\,\mathsf{P}, \mathsf{A} \bullet \mathsf{P} := s;\ (s \leq \mathsf{P} < f) * \mathsf{m}[\mathsf{P}];\ (\mathsf{P} = f)_\perp$$

where s is the intended start address and f the finish address of the code to be executed. The obligation to start at the right instruction is expressed by the initial assignment $\mathsf{P} := s$, and the obligation to terminate at the right place (and not by a wild jump) is expressed by the final assertion $(P = f)_\perp$.

The design of a compiler is nothing but a constructive proof that every program, however deeply structured, can be improved by some program in this normal form. The process is split into three main phases: concerns of control elimination are separated from those of expression decomposition and data representation. In order to illustrate these phases, consider the compilation of an assignment of the form

$$x := y$$

where hoth x and y are variables. The simplification of expressions must produce assignments which will eventually give rise to one of the patterns used to define the machine instructions. Recalling that A represents the general purpose register of our simple machine, this assignment is transformed into

$$\mathsf{dec}\,\mathsf{A} \bullet \mathsf{A} := y;\ x := \mathsf{A}$$

where the first assignment will become a load and the second one a store instruction. But this program still operates on abstract global variables with symbolic names, whereas the target program operates only on the concrete store M. We therefore define a data refinement Ψ which maps each program variable x onto the corresponding machine location, so the value of x is held as $\mathsf{M}[\Psi x]$. Here Ψ is the compiler's symbol table, an injection which maps each variable onto the distinct location allocated to hold its value. Therefore

we need a data refinement phase to justify the substitution of $\mathsf{M}[\Psi x]$ for x throughout the program. When this data refinement is performed on our simple example program it becomes

$$\mathbf{dec}\,\mathsf{A} \bullet \mathsf{A} := \mathsf{M}[\Psi y];\ \mathsf{M} := \mathsf{M} \oplus \{\Psi x \mapsto \mathsf{A}\}$$

The remaining task of the compiler is that of control refinement, reducing the nested control structure of the source program to a single flat iteration, like that of the target program. This is done by introducing a control state variable to schedule the selection and sequencing of actions. In the case of our simple target machine, a single program pointer P indicates the location in memory of the next instruction. The above then becomes

$$\begin{array}{l}\mathbf{dec}\,\mathsf{P},\mathsf{A} \bullet \mathsf{P} := s;\\ \quad (s \leq \mathsf{P} < s+2) * \left(\begin{array}{ll} & (\mathsf{P} = s) \to \mathsf{A},\mathsf{P} := \mathsf{M}[\Psi y], \mathsf{P}+1 \\ \square & (\mathsf{P} = s+1) \to \mathsf{M},\mathsf{P} := (\mathsf{M} \oplus \{\Psi x \mapsto \mathsf{A}\}), \mathsf{P}+1 \end{array}\right);\\ \quad (\mathsf{P} = s+2)_\perp\end{array}$$

where we use $\square$ as syntactic sugar for $\sqcap$ when the choice is deterministic. The initial assignment $\mathsf{P} := s$ ensures that the assignment with guard $(\mathsf{P} = s)$ will be executed first; as this increments P, the assignment with guard $(\mathsf{P} = s + 1)$ is executed next. This also increments P, falsifying the condition of the iteration and satisfying the final assertion.

Note that the guarded assignments correspond precisely to the patterns used to define the load and store instructions, respectively. The above expresses the fact that these instructions must be loaded into the memory m at positions s and $s + 1$, completing the overall process.

The reduction theorems which justify the entire process are all provably correct from the basic algebraic laws of the language. Furthermore, some of the proofs are verified using OBJ3 and the reduction theorems are used as rewrite rules to do the compilation automatically.

1.2 Overview of Subsequent Chapters

Chapter 2 describes in some detail the view of specifications as monotonic predicate transformers, in the sense advocated by Dijkstra. We review the mathematical concepts of partial orders and lattices, and show that the language introduced above forms a complete distributive lattice of monotonic predicate transformers. We give examples of refinement calculi based on these ideas and address the problem of data refinement. Finally, we link our approach to compilation to the more general task of deriving programs from specifications.

In Chapter 3 we give meaning to the reasoning language in terms of equations and inequations which we call algebraic laws. The final section of this chapter discusses how the laws can be proved by linking the algebraic semantics to a given mathematical model; for the purpose of illustration we use weakest preconditions.

A complete compiler for the simplified source language (that is, not including procedures or recursion) is given in Chapter 4. The first two sections describe the normal form as a model of an arbitrary executing mechanism. The reduction theorems associated with this form are largely independent of a particular way of representing control state. We show that the use of a program pointer is one possible instantiation. The design of the compiler is split into three phases, as illustrated in Section 1.1.

In Chapter 5 we deal with procedures and recursion, and address the issue of parameterisation. We show how each of these can be eliminated through reduction to normal form; but we leave open the choice of a target machine to implement them. Each feature is treated by a separate theorem, in complete independence from the constructions of the simpler language. This illustrates the modularity of the approach.

Chapter 6 is concerned with the mechanisation of the approach. The purpose is to show how this can be achieved using the OBJ3 term rewriting system. There are three main activities involved: The formalisation (specification) of concepts such as the reasoning language, its algebraic laws, the normal form, the target machine, and so on, as a collection of *theories* in OBJ3; the verification of the related theorems; and the use of the reduction theorems as a compiler prototype. The final section of this chapter includes a critical view of the mechanisation, and considers how other systems can be used to perform the same (or a similar) task.

In the final chapter we summarise our work and discuss related and future work. The very final section contains a brief analysis of the overall work.

Apart from the main chapters, there are three appendices. Although the emphasis of our work is on the compilation of control structures, Appendix A describes a scheme for compiling arrays. Appendix B contains the proof of a lemma used to prove the reduction theorem for recursion. Appendix C contains more details about the mechanisation, including complete proofs of some of the main theorems.

Chapter 2

Background

> The beauty of lattice theory derives in part from the extreme simplicity of its basic concepts: (partial) ordering, least upper and greatest lower bounds.
>
> — G. Birkhoff

The purpose of this chapter is to briefly describe a theoretical basis for the kind of refinement algebra we will be using. Based on the work of Morris [59] and Back and von Wright [5] we show that a specification language (as introduced in the previous chapter) forms a complete distributive lattice where, following Dijkstra [21], specifications are viewed as monotonic predicate transformers. We give examples of refinement calculi based on these ideas and address the problem of data refinement. Finally, we link our approach to compilation to the more general task of deriving programs from specifications.

The first section reviews the concepts of partial orders and complete distributive lattices, and a simple boolean lattice is presented as an example. The next section describes the predicate lattice as functions from (program) states to booleans; this is constructed by pointwise extension from the boolean lattice. Further pointwise extension is used to construct the lattice of predicate transformers described in Section 2.3; these are functions from predicates to predicates. In Section 2.4 we review some properties of predicate transformers (known as *healthiness conditions*) and explain that some of them are dropped as a consequence of adding non-implementable features to the language. Some refinement calculi based on the predicate transformer model are considered in Section 2.5, and some approaches to data refinement in Section 2.6. The final section relates all these to our work.

2.1 Partial Orders and Lattices

A *partial order* is a pair $(S, \sqsubseteq)$ where S is a set and $\sqsubseteq$ is a binary relation (the *partial ordering*) on S satisfying the following axioms, for all $x, y, z \in S$:

$x \sqsubseteq x$	reflexivity
$(x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow (x \sqsubseteq z)$	transitivity
$(x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow (x = y)$	antisymmetry

$(S, \sqsubseteq)$ is called a *total order* if, in addition to the above, each pair of elements in S are comparable: $(x \sqsubseteq y) \vee (y \sqsubseteq x)$. Following usual practice we will abbreviate $(S, \sqsubseteq)$ to S; the context will make it clear if we are regarding S as a set or as a partial order.

Given a subset T of S, we say that $x \in S$ is an *upper bound* for T if $y \sqsubseteq x$ for all $y \in T$; x is the *least upper bound* of T if it is both an upper bound for T and whenever y is another upper bound for T then $x \sqsubseteq y$. Similarly, x is a *lower bound* for T if $x \sqsubseteq y$ for all $y \in T$; x is the *greatest lower bound* of T if it is both a lower bound for T and whenever y is another lower bound for T then $y \sqsubseteq x$. An element $\bot$ is a *least element* or *bottom* of S if $\bot \sqsubseteq x$ for all $x \in S$; $\top$ is a *greatest element* or *top* of S if $x \sqsubseteq \top$ for all $x \in S$.

Given sets S and T with T partially ordered by $\sqsubseteq_T$, the set $S \to T$ of functions from S to T is partially ordered by $\sqsubseteq$ defined by

$$f \sqsubseteq g \stackrel{def}{=} f(x) \sqsubseteq_T g(x) \quad \text{for all } x \in S$$

Additionally, if S is partially ordered by $\sqsubseteq_S$, then $f : S \to T$ is said to be *monotonic* if

$$x \sqsubseteq_S y \;\Rightarrow\; f(x) \sqsubseteq_T f(y) \quad \text{for all } x, y \in S$$

We denote by $[S \to T]$ the set of monotonic functions from S to T. If S is *discrete* (that is, $x \sqsubseteq_S y$ holds if and only if $x = y$) then $S \to T$ and $[S \to T]$ are identical.

A *complete lattice* is a partially ordered set containing arbitrary greatest lower bounds (*meets*) and least upper bounds (*joins*). A consequence is that every complete lattice has a bottom and a top element. Two additional well-known properties of complete lattices are given below; more details about lattice theory can be found, for example, in [8].

- Any finite totally ordered set is a complete lattice.
- If S is a partially ordered set and T is a complete lattice, then $[S \to T]$ is a complete lattice.

A lattice is said to be *distributive* if the least upper bound operator distributes through the greatest lower bound operator, and vice versa.

A very simple example of a complete distributive lattice is the boolean set $\{true, false\}$ when ordered by the implication relation. The least upper bound $\vee$ and the greatest lower bound $\wedge$ have their usual interpretations as disjunction and conjunction, respectively. The bottom element is *false* and the top element is *true*. This is actually a complete *boolean* lattice, since it has a *complement* (negation); but we will not use this property. In the next section we will refer to this lattice as *Bool*.

2.2 The Lattice of Predicates

Programs usually operate on a state space formed from a set of variables. We use *State* to stand for the set of all possible states partially ordered by the equality relation; therefore, it is a discrete partial order. In practice we need a way to describe particular sets of states; for example, to specify the set of initial states of a program, as well as the set of its final states. This can be described by boolean-valued functions (or predicates) on the state space.

As *State* is a partial order and *Bool* is a complete lattice, $[State \to Bool]$ is also a complete lattice. Furthermore, as *State* is discrete, $[State \to Bool]$ and $State \to Bool$ are identical. We will refer to it as the *Predicate* lattice. The least upper bound $a \vee b$ is the disjunction of the predicates a and b, and the greatest lower bound $a \wedge b$ is their conjunction. The bottom element **false** describes the empty set of states and the top element **true** describes the set of all possible states. These operations are defined by

$$
\begin{aligned}
(a \vee b) &\stackrel{def}{=} \lambda\, x \bullet a(x) \vee b(x) \\
(a \wedge b) &\stackrel{def}{=} \lambda\, x \bullet a(x) \wedge b(x) \\
\mathsf{true} &\stackrel{def}{=} \lambda\, x \bullet \mathit{true} \\
\mathsf{false} &\stackrel{def}{=} \lambda\, x \bullet \mathit{false}
\end{aligned}
$$

where the bounded variable x ranges over *State*. The lattice ordering is the implication on predicates, which is defined by pointwise extension in the usual way:

$$a \Rightarrow b \stackrel{def}{=} \forall x \bullet a(x) \Rightarrow b(x)$$

2.3 The Lattice of Predicate Transformers

The lattice presented next provides a theoretical basis for Dijkstra's view of programs as predicate transformers (functions from predicates to predicates) [21]. The usual notation

$$wp(p, a) = c$$

means that if program p is executed in an initial state satisfying its ***weakest precondition*** c, it will eventually terminate in a state satisfying the postcondition a. Furthermore, as the name suggests, the weakest precondition c describes the largest possible set of initial states which ensures that execution of p will terminate in a state satisfying a.

The predicate transformer lattice (*PredTran*) is the set of all monotonic functions from one predicate lattice to another: $[Predicate \to Predicate]$. The result of applying program (predicate transformer) p to predicate a, denoted $p(a)$, is equivalent to Dijkstra's $wp(p, a)$. The ordering on predicate transformers is defined by pointwise extension from the ordering on predicates

$$p \sqsubseteq q \stackrel{def}{=} \forall a \bullet p(a) \Rightarrow q(a)$$

where the bounded variable a ranges over the set of predicates.

Clearly, *PredTran* is a complete lattice and, therefore, it contains arbitrary least upper bounds and greatest lower bounds: $\sqcap$ is interpreted as demonic nondeterminism and $\sqcup$ as angelic nondeterminism. The bottom element is *abort* (denoted by $\bot$), the predicate transformer that does not establish any postcondition. The top element is *miracle* (denoted by $\top$); it establishes every postcondition. These are defined in the obvious way:

$$
\begin{aligned}
(p \sqcup q) &\stackrel{def}{=} \lambda\, a \bullet p(a) \vee q(a) \\
(p \sqcap q) &\stackrel{def}{=} \lambda\, a \bullet p(a) \wedge q(a) \\
\top &\stackrel{def}{=} \lambda\, a \bullet \mathsf{true} \\
\bot &\stackrel{def}{=} \lambda\, a \bullet \mathsf{false}
\end{aligned}
$$

The usual program constructs can be defined as predicate transformers. As an example we define skip (the identity predicate transformer) and sequential composition:

$$
\begin{aligned}
\mathsf{skip} &\stackrel{def}{=} \lambda\, a \bullet a \\
p;\ q &\stackrel{def}{=} \lambda\, a \bullet p(q(a))
\end{aligned}
$$

2.4 Properties of Predicate Transformers

Dijkstra [21] has suggested five healthiness conditions that every construct of a programming language must satisfy. They are defined below (we assume implicit universal quantification over a and b standing for predicates, and over p standing for programs)

1. $p(\mathsf{false}) = \mathsf{false}$	law of the excluded miracle
2. If $a \Rightarrow b$ then $p(a) \Rightarrow p(b)$	monotonicity
3. $p(a) \wedge p(b) = p(a \wedge b)$	conjunctivity
4. $p(a) \vee p(b) = p(a \vee b)$	disjunctivity
5. $p(\exists\, i : i \geq 0 : a_i) = \exists\, i : i \geq 0 : p(a_i)$ for all sequences of predicates $a_0, a_1, \ldots$ such that $a_i \Rightarrow a_{i+1}$ for all $i \geq 0$	continuity

The fourth property is satisfied only by *deterministic* programs; for nondeterministic programs, the equality has to be replaced by an implication. The last property is equivalent to requiring that nondeterminism be bounded [23].

The complete lattice *PredTran* includes predicate transformers useful for specification purposes; they are not implementable in general. Of the above properties, only monotonicity is satisfied by all the predicate transformers in *PredTran*: $\top$ trivially breaks the law of the excluded miracle; the fact that greatest lower bounds over arbitrary sets are allowed implies that the assumption of bounded nondeterminism (and therefore continuity) is not satisfied; and angelic nondeterminism violates the property of conjunctivity.

Of course, the healthiness conditions are still of fundamental importance; they are the criteria for distinguishing the implementable from the non-implementable in a general space of specifications.

2.5 Some Refinement Calculi

Back [3, 5], Morris [59] and Morgan [55, 56] have developed refinement calculi based on weakest preconditions. These calculi have the common purpose of formalising the well established *stepwise refinement* method for the systematic construction of programs from high-level specifications [76, 20].

As originally proposed, the stepwise refinement method is partly informal. Although specifications and programs are formal objects, the intermediate terms of a given derivation do not have a formal status. The essence of all the refinement calculi cited above is to extend a given procedural language (in particular, Dijkstra's guarded command language) with additional features for specification. For example, let $[a, c]$ be a specification construct used to describe a program that when executed in a state satisfying a, terminates in a state satisfying c. This can be viewed as a predicate transformer in just the same way as the other operators of the language. Its definition is given by

$$[a, c] \stackrel{def}{=} \lambda b \bullet a \wedge (c \Rightarrow b)$$

The extended language is thus a specification language and programs appear as a subclass of specifications. Programming is then viewed as constructing a seqnence of specifications; the initial specification is in a high-level of abstraction (not usually implementable) and the final specification is an executable program. The derivation process is to gradually transform specifications into programs. The intermediate steps of the derivation will normally contain a mixture of specification and program constructs; but these are formal objects too, since specifications and programs are embedded in the same semantic framework.

Derivation requires the notion of a refinement relation between specifications. All the cited calculi use the same definition of the refinement relation which is precisely the ordering on the lattice of predicate transformers described above. Two mathematical properties of this ordering are of fundamental importance to model stepwise refinement. Monotonicity of the language operators with respect to this ordering is necessary to allow a given specification to be replaced by a refinement of it in an arbitrary context; this can only refine the overall context. The other required property is transitivity: as the derivation process will normally entail a large number of steps, it is necessary to ensure that the final product (that is, the program) satisfies the original specification.

Rules for introducing programming constructs from given specifications are the additional tools required in the process. For example, the following rules illustrate the introduction of **skip** and sequential composition:

$$[a, c] \sqsubseteq \mathbf{skip} \text{ if } a \Rightarrow c$$

$$[a, c] \sqsubseteq [a, b]; [b, c]$$

There are also rules for manipulating specifications; for example, weakening the precondition of a given specification or strengthening its postcondition (or botb) lead to a specification which refines the original one:

$$[a_1, c_1] \sqsubseteq [a_2, c_2] \text{ if } a_1 \Rightarrow a_2 \wedge c_2 \Rightarrow c_1$$

Morris [59] was the first to give a lattice theoretic basis for the refinement calculus. He extended Dijkstra's guarded command language with predicate pairs (as illustrated above) and general recursion. Although he observed that the framework contains arbitrary least upper bounds and greatest lower bounds, these have not been incorporated into his specification language.

Back and von Wright [5] further explored the lattice approach and suggested a more powerful (infinitary) language which is complete in the sense that it can express every monotonic predicate transformer. The only constructors in their language are the lattice operators $\sqcap$ and $\sqcup$, together with functional composition (modelling sequential composition). From a very simple command language including these constructors, they define ordinary program constructs such as assignment, conditional and recursion. Inverses of programs are defined and used to formalise the notion of data refinement. This is further discussed in the next section.

Morgan's calculus [56] is perhaps the most appealing to practising programmers. His language includes procedures (possibly recursive and parameterised) and even modules. He defines a large number of refinement laws and illustrates their application in a wide range of derivations. The *specification statement*

$$x : [a, c]$$

is another distinctive feature of his work. Although it is similar to the notation used above, it includes the notion of a *frame*: x is a list of variables whose values may change. The frame reduces the number of possible refinements from a given specification, and is a way of making the designer's intention clearer. Furthermore, the above construct is very general and can be specialised for many useful purposes. For example, our *generalised assignment* command can be regarded as a special case of it:

$$x :\in a \;=\; x : [\mathsf{true}, a]$$

where the purpose is to establish a without changing any variables other than x. Another example is an *assumption* of a, meaning that a must be established without changing any variable:

$$a^{\top} \;=\; : [\mathsf{true}, a]$$

2.6 Data Refinement

In the previous section we discussed how an abstract specification is transformed into a program by progressively introducing control structures; this is known as *algorithmic* or *control* refinement. But this is only part of the process to obtain an implementation. Specifications are usually stated in terms of mathematical data types like sets and relations, and these are not normally available in procedural programming languages. Therefore the complementary step to *control* refinement is the transformation of the abstract types into concrete types such as arrays and records which can be efficiently implemented. This task is known as *data refinement*.

The idea of data refinement was first introduced by Hoare [41]. The basis of his approach is the use of an *abstraction function* to determine the abstract state that a given concrete state represents; in addition, the set of concrete states may be constrained by an *invariant* relation. Since then many approaches have been suggested which build on these ideas. The more recent approaches use a single relation to capture both the abstraction function and the invariant, thus relaxing the assumption that the abstract state is functionally dependent on the concrete state.

In connection with the refinement calculi considered in the previous section, two very similar approaches have been suggested by Morris [61] and Morgan and Gardiner [57]. In both cases, data refinement is characterised as a special case of algorithmic refinement between *blocks*. A block of the form

$$\mathsf{dec}\, x : Tx \bullet p$$

is used to represent the abstract program p operating on the variables x with type[1] Tx. Similarly,

$$\mathsf{dec}\, x' : Tx' \bullet p'$$

represents the concrete program p' which operates on the variables x' with type Tx'. The data refinement is captured by the inequation

$$(\mathsf{dec}\, x : Tx \bullet p) \sqsubseteq (\mathsf{dec}\, x' : Tx' \bullet p')$$

The general aim is to construct the concrete block by replacing the abstract local variables with the concrete ones, in such a way that the overall effect of the abstract block is preserved. In particular, p' is constructed with the same structure as p in the sense that each command in p' is the translation of a corresponding command in p, according to a uniform rule.

An essential ingredient to this strategy is an abstract invariant I which links the abstract variables x to the concrete variables x'. This is called the ***coupling invariant***. A new relation between programs is defined to express that program p' (operating on variables x') is a data refinement of program p (operating on variables x) under coupling invariant I. This is written $p \leq_{I,x,x'} p'$ and is formally defined by (considering programs as predicate transformers)

$$p \leq_{I,x,x'} p' \stackrel{def}{=} (\exists\, x : I \wedge p(a)) \Rightarrow p'(\exists\, x : I \wedge a) \quad \text{for all } a \text{ not containing } x'$$

Broadly, the antecedent requires that the initial values of the concrete variables couple to some set of abstract values for which the abstract program will succeed in establishing postcondition a; the consequent requires that the concrete program yields new concrete values that also couple to an acceptable abstract state.

[1] Recall that our language is untyped; types are considered here only for the purpose of the present discussion.

This definition is chosen for two main reasons. The first is that it guarantees the characterisation of data refinement given above, that is

$$\text{If } (p \leq_{I,x,x'} p') \text{ then } (\mathsf{dec}\, x : Tx \bullet p) \sqsubseteq (\mathsf{dec}\, x' : Tx' \bullet p')$$

The second reason is that it distributes through the program constructors, thus allowing data refinement to be carried out piecewise. For example, the distribution through sequential composition is given by

$$\text{If } (p \leq_{I,x,x'} p') \text{ and } (q \leq_{I,x,x'} q') \text{ then } (p;\ q) \leq_{I,x,x'} (p';\ q')$$

Back and von Wright [5] suggest an approach which avoids the need to define a data refinement relation. They use the algorithmic refinement relation not only to characterise data refinement, but also to carry out the calculations. The basic idea is to introduce an *encoding program*, say ψ, which computes abstract states from concrete states and a *decoding program* program, say ϕ, which computes concrete states from abstract states. Then, for a given abstract program p, the task is to find a concrete program p' such that

$$\psi;\ p;\ \phi \sqsubseteq p'$$

With the aid of specification features, it is possible to give very high-level definitions for ψ and ϕ. Using the same convention adopted above that x stands for the abstract variables, x' for the concrete variables and I for the coupling invariant, ψ is defined by

$$\psi \stackrel{def}{=} \mathsf{var}\, x;\ x :\stackrel{\perp}{\in} I;\ \mathsf{end}\, x'$$

It first introduces the abstract variables x and assigns them values such that the invariant is satisfied, and then removes the concrete variables from the data space. The use of $\perp$ as an annotation in the above generalised assignment command means that it aborts if I cannot be established. Similarly we have the definition of ϕ

$$\phi \stackrel{def}{=} \mathsf{var}\, x';\ x' :\stackrel{\top}{\in} I;\ \mathsf{end}\, x$$

which introduces the concrete variables x' and assigns them values such that the invariant is satisfied, and then removes the abstract variables from the data space. But in this case the generalised assignment command results in a miracle if I cannot be established. (The above two kinds of generalised assignment commands were introduced only for the purpose of the present discussion. Recall from the previous chapter that our language includes only the latter kind, and henceforth we will use the previous notation $x :\in b$.)

Note that having separate commands to introduce and end the scope of a variable is an essential feature to define the encoding and decoding programs: the first introduces x and ends the scope of x'; the second introduces x' and ends the scope of x.

In this approach, data refinement can also be performed piecewise, by proving distributivity properties such as

$$\psi;\ (p;\ q);\ \phi \sqsubseteq (\psi;\ p;\ \phi);\ (\psi;\ q;\ \phi)$$

which illustrates that both algorithmic and data refinement can be carried out within the framework of one common refinement relation.

2.7 Refinement and Compilation

As explained in the previous chapter, we regard compilation as a task of program refinement. In this sense, we can establish some connections between our view of compiler design and the more general task of deriving programs from specifications (henceforth we will refer to the latter simply as "derivation"). In both cases, a programming language is extended with specification features, so that a uniform framework is built and the interface between programs and specifications (when expressed by distinct formalisms) is avoided. In particular, our language is roughly the same as the one defined by Back and von Wright [5], except that we deal with procedures and parameterisation. The first three sections of this chapter briefly explained how the language can be embedded in a complete distributive lattice of predicate transformers.

In a derivation, the idea is to start with an arbitrary specification and end with a program formed solely from constructs which can be executed by computer. In our case, the initial object is an arbitrary source program and the final product is its normal form. But the tools used to achieve the goals in both cases are of identical nature: transformations leading to refinement in the sense already discussed.

Derivation entails two main tasks: control and data refinement. We also split the design of the compiler into these two main phases. However, while in a derivation control refinement is concerned with progressively introducing control structure in the specification, we do the reverse process; we reduce the nested control structure of a source program to the single flat iteration of the normal form program.

Regarding data refinement, the general idea is the same both in a derivation process and in designing a compiler: to replace abstract data types with concrete representations. In particular, we use the idea of encoding and decoding programs. As discussed in the previous section, this avoids the need to define a separate relation to carry out data refinement. In our case, an encoding program retrieves the abstract space of the source program from the concrete state representing the store of the machine. Conversely, a decoding program maps the abstract space to the concrete machine state. In the next chapter, the pair formed by an encoding and the respective decoding program is formally defined as a *simulation*. It satisfies the distributivity properties illustrated above, allowing data refinement to be carried out piecewise.

But, as should be expected, there are some differences between designing a compiler in this way and the more general task of deriving programs from specifications. For example, we are not interested in capturing requirements in general, and therefore our language includes no construct to serve this purpose. The closest to a specification statement we have in our language is the generalised assignment command. Our use of it is to abstract from the way control state is encoded in a particular target machine.

Another difference is that we are mostly concerned with program transformation. We need a wide range of laws relating the operators of the language. In particular, we follow the approach suggested by Hoare and others [47] where the semantics of a language is characterised by a set of equations and inequations (laws) relating the language operators.

The same approach has been used to define an algebraic semantics for occam [68]. In our case, the set of laws must be complete in that it should allow us to reduce an arbitrary program to normal form. The framework we use is better characterised as a *refinement algebra* (rather than as a calculus).

Chapter 3

The Reasoning Language

> If you are faced by a difficulty or a controversy in science, an ounce of algebra is worth a ton of verbal argument.
>
> — J.B.S. Haldane

Here we give meaning to our specification (reasoning) language in terms of equations and inequations (laws) relating the operators of the language. Following Hoare and others [47, 68], we present the laws as self-evident axioms, normally preceded by an informal (operational) justification. Moreover, it is not our aim to describe a complete set of laws in the logical sense; although they are complete in that they will allow us to reduce an arbitrary source program to a normal form.

It is possible to select a small subset of our language and define the additional operators in terms of the more basic ones. This is shown in [45], where the only constructors are sequential composition, $\sqcup$ and $\sqcap$. The additional operators are defined in terms of these and the primitive commands. The laws of derived operators can then be proved from their definition and the laws of the basic operators.

This is not our concern here; our emphasis is on the algebraic laws which will be used in the process of designing a compiler. However, we do illustrate how a few operators can be defined from others. In particular, iteration is defined as a special case of recursion and all the laws about iteration are proved. They deserve such special attention because of their central role in the proofs of the normal form reduction theorems.

We will not normally distinguish between programs and specifications. We will refer to both of them as "programs". Another remark is that programs have both a syntactic and a semantic existence. On one hand we perform syntactic operations on them, such as substitution. On the other hand, the algebraic laws relating language operators express semantic properties. Strictly, we should distinguish between these two natures of programs. But it is not convenient to do so and it will be clear from the context which view we are taking.

The first section gives notational conventions and introduces the concepts of substitution and free and bound identifiers. Each of the subsequent sections describes the laws of

one or more language operators. The concepts of a refinement relation, approximate inverse (Galois connection) and simulation will be introduced when the need arises. The final section describes alternative ways in which the laws of the basic operators could be verified. As an example, we take the view of programs as predicate transformers (as discussed in the previous chapter) and illustrate how the laws can be proved.

3.1 Concepts and Notation

Name conventions

It is helpful to define some conventions as regards the names used to denote program terms:

X, Y, Z	program identifiers
p, q, r	programs
x, y, z	lists of variables
a, b, c	boolean expressions
e, f, g	lists of expressions

We also use subscripts in addition to the above conventions. For example, $b_0, b_1, \ldots$ stand for boolean expressions (also referred to as conditions). We use comma for list concatenation: x, y stands for the concatenation of lists x and y. Further conventions are explained when necessary.

Precedence rules

In order to reduce the number of brackets around program terms, we define the following precedence rules. Operators with the same precedence appear on the same line. As usual, we will assume that brackets bind tighter than any operator.

uop	unary operators	**binds tightest**
bop	binary operators	
,	list concatenation	
$:\in$ and $:=$	(generalised) assignment	
$\rightarrow$	guarded command	
$*$	iteration	
;	sequential composition	
$\sqcup$ and $\sqcap$	nondeterminism (angelic and demonic)	
$\lhd\ \rhd$	conditional	
μ	recursion	
dec	block with local declarations	**binds loosest**

Procedures are dealt with in Chapter 5 and are assumed to have the same precedence as μ. We will normally add some brackets (even if unnecessary) to aid readability.

Free and bound identifiers

An occurrence of a variable x in a program p is *free* if it is not in the scope of any static declaration of x in p, and ***bound*** otherwise. For example, x is bound in $\textbf{dec}\, x \bullet x := y$, but free in $x := y$. Notice that the commands for dynamic declaration are not ***binders*** for variables. For example, x is free in $\textbf{var}\, x$ as well as in $\textbf{end}\, x$. A list of variables is free in p if each variable in the list is free in p.

In the case of program identifiers, we say that an occurrence of X is free in a program p if it is not in the scope of any recursive program (with name X) defined in p, and bound otherwise.

Substitution

For variables x and y,

$$p[x \leftarrow y]$$

denotes the result of substituting y for every free occurrence of x in p. It is possible for x to be in the scope of (static) declarations of variables with the same name as y. In this case, a systematic renaming of local variables of p occurs in order to avoid variable *capture*. This is usually referred to as *safe* substitution.

If x and y are (equal-length) lists of variables, the substitution is positional. In this case, no variable may appear more than once in the list x.

Similarly,

$$f[x \leftarrow e]$$

denotes the substitution of the list of expressions e for the (equal-length) list of variables x in the list of expressions f.

We also allow the substitution of programs for program identifiers:

$$p[X \leftarrow q]$$

This avoids capture of any free identifiers of q by renaming local declarations in p, as discussed above. For conciseness, we will sometimes avoid writing substitutions of the last kind by making (free) occurrences of X explicit, as in $F(X)$. Then, the substitution of q for X in this case is written $F(q)$. In any case we assume that no capture of free identifiers occur.

Laws, definitions, lemmas, theorems and proofs

Each of the laws described in the following sections is given a number and a name suggestive of its use. The number is prefixed with the corresponding section number, in order

to ease further references. The name normally mentions the operators related by the law. For example,

⟨; −skip unit⟩

is the name associated with the law which says that skip is the unit of sequential composition. Every reference to a law comprises both its name and its number.

Some of the laws could be alternatively described as lemmas or theorems, as they are proved from more basic ones. However, we prefer to regard all the equations (and inequations) relating the language operators as laws. Each of the definitions, lemmas and theorems is also given a number and a name for further references.

Our proofs are confined to (in)equational reasoning. We use the terms *LHS* and *RHS* to refer to the left- and the right-hand sides of an (in)equation. The proof strategy is to start with one of the sides and try to reach the other side by a series of algebraic transformations. Each step is annotated with one or more references to laws, definitions, lemmas or theorems.

3.2 Skip, Abort and Miracle

The skip command has no effect and always terminates successfully.

The *abort* command, denoted by $\bot$, is the most unpredictable of all programs. It may fail to terminate or it may terminate with any result whatsoever. Thus $\bot$ represents the behaviour of a broken machine, or a program that has run wild.

The *miracle* command, denoted by $\top$, is the other extreme: it can be used to serve any purpose. But it is *infeasible* in that it cannot be implemented; otherwise we would not need to write programs—$\top$ would do anything for us.

The laws governing these primitive commands are included in the remaining sections. This is because each of these laws normally expresses a unit or zero property of one language operator.

3.3 Sequential Composition

The program $p;\ q$ denotes the usual sequential composition of programs p and q. If the execution of p terminates successfully then the execution of q follows that of p.

Since the execution of skip always terminates and leaves everything unchanged, to precede or follow a program p by skip does not change the effect of p. In other words, skip is both the left and the right unit of sequential composition.

Law 3.3.1 $(\mathsf{skip};\ p) \;=\; p \;=\; (p;\ \mathsf{skip})$ ⟨; −skip unit⟩

To specify the execution of a program p after the termination of $\bot$ cannot redeem the situation, because $\bot$ cannot be relied on to terminate. More precisely, $\bot$ is a left zero of sequential composition.

Law 3.3.2 $\bot;\ p\ =\ \bot$ $\langle ; -\bot$ left zero$\rangle$

To precede a program p by $\top$ results in a miracle; $\top$ is a left zero of sequential composition.

Law 3.3.3 $\top;\ p\ =\ \top$ $\langle ; -\top$ left zero$\rangle$

Sequential composition is associative.

Law 3.3.4 $(p;\ q);\ r\ =\ p;\ (q;\ r)$ $\langle ;$ assoc$\rangle$

3.4 Demonic Nondeterminism

The program $p \sqcap q$ denotes the demonic choice of programs p and q: either p or q is selected, the choice being totally arbitrary.

The abort command already allows completely arbitrary behaviour, so an offer of further choice makes no difference to it.

Law 3.4.1 $p \sqcap \bot\ =\ \bot$ $\langle \sqcap - \bot$ zero$\rangle$

On the other hand, the miracle command offers no choice at all.

Law 3.4.2 $p \sqcap \top\ =\ p$ $\langle \sqcap - \top$ unit$\rangle$

When the two alternatives are the same program, the choice becomes vacuous—$\sqcap$ is idempotent.

Law 3.4.3 $p \sqcap p\ =\ p$ $\langle \sqcap$ idemp$\rangle$

The order in which a choice is offered is immaterial—$\sqcap$ is symmetric.

Law 3.4.4 $p \sqcap q\ =\ q \sqcap p$ $\langle \sqcap$ sym$\rangle$

Demonic choice is associative.

Law 3.4.5 $(p \sqcap q) \sqcap r\ =\ p \sqcap (q \sqcap r)$ $\langle \sqcap$ assoc$\rangle$

3.5 Angelic Nondeterminism

The angelic choice of two programs p and q is denoted by $p \sqcup q$. Informally, it is a program that may act like p or q, whichever is more suitable in a given context.

As we have mentioned before, $\bot$ is totally unpredicatable and therefore the least suitable program for all purposes.

Law 3.5.1 $\bot \sqcup p \;=\; p$ $\qquad$ $(\sqcup - \bot \text{ unit})$

On the other extreme, $\top$ suits any situation.

Law 3.5.2 $\top \sqcup p \;=\; \top$ $\qquad$ $(\sqcup - \top \text{ zero})$

Like $\sqcap$, angelic choice is idempotent, symmetric and associative.

Law 3.5.3 $p \sqcup p \;=\; p$ $\qquad$ $(\sqcup \text{ idemp})$

Law 3.5.4 $p \sqcup q \;=\; q \sqcup p$ $\qquad$ $(\sqcup \text{ sym})$

Law 3.5.5 $(p \sqcup q) \sqcup r \;=\; p \sqcup (q \sqcup r)$ $\qquad$ $(\sqcup \text{ assoc})$

3.6 The Ordering Relation

Here we define the ordering relation $\sqsubseteq$ on programs: $p \sqsubseteq q$ holds whenever the program q is at least as deterministic as p or, alternatively, whenever q offers only a subset of the choices offered by p. In this case, q is at least as predictable as p. This coincides with the meaning we adopt for *refinement*. Thus $p \sqsubseteq q$ can be read as "p is refined by q" or "p is worse than q".

We define $\sqsubseteq$ in terms of $\sqcap$. Informally, if the demonic choice of p and q always yields p, one can be sure that p is worse than q in all situations.

Definition 3.1 (The ordering relation)

$$p \sqsubseteq q \;\stackrel{def}{=}\; (p \sqcap q) \;=\; p$$

■

In the final section we prove that this ordering coincides with the ordering on the lattice of predicate transformers described in the previous chapter.

Alternatively, the ordering relation could have been defined in terms of $\sqcup$.

Law 3.6.1 $p \sqsubseteq q \;\equiv\; (p \sqcup q) \;=\; q$ $\qquad$ $(\sqsubseteq - \sqcup)$

From Definition 3.1 and the laws of $\sqcap$, we conclude that $\sqsubseteq$ is a partial ordering on programs:

Law 3.6.2 $p \sqsubseteq p$ $\langle\sqsubseteq$ reflexivity$\rangle$

Law 3.6.3 $(p \sqsubseteq q) \wedge (q \sqsubseteq p) \Rightarrow (p = q)$ $\langle\sqsubseteq$ antisymmetry$\rangle$

Law 3.6.4 $(p \sqsubseteq q) \wedge (q \sqsubseteq r) \Rightarrow (p \sqsubseteq r)$ $\langle\sqsubseteq$ transitivity$\rangle$

Moreover $\sqsubseteq$ is a lattice ordering. The bottom and top elements are $\perp$ and $\top$, respectively; the *meet* (*greatest lower bound*) and *join* (*least upper bound*) operators are $\sqcap$ and $\sqcup$, in this order. These are also consequences of the definition of $\sqsubseteq$ and the laws of $\sqcap$ and $\sqcup$.

Law 3.6.5 $\perp \sqsubseteq p$ $\langle\sqsubseteq-\perp$ bottom$\rangle$

Law 3.6.6 $p \sqsubseteq \top$ $\langle\sqsubseteq-\top$ top$\rangle$

Law 3.6.7 $(r \sqsubseteq p \wedge r \sqsubseteq q) \equiv r \sqsubseteq (p \sqcap q)$ $\langle\sqsubseteq-\sqcap$ glb$\rangle$

Law 3.6.8 $(p \sqsubseteq r) \wedge (q \sqsubseteq r) \equiv (p \sqcup q) \sqsubseteq r$ $\langle\sqsubseteq-\sqcup$ lub$\rangle$

In order to be able to use the algebraic laws to transform subcomponents of compound programs, it is crucial that $p \sqsubseteq q$ imply $F(p) \sqsubseteq F(q)$, for all *contexts* F (functions from programs to programs). This is equivalent to saying that F (and consequently, all the operators of our language) must be *monotonic* with respect to $\sqsubseteq$. For example:

Law 3.6.9 If $p \sqsubseteq q$ then
(1) $(p \sqcap r) \sqsubseteq (q \sqcap r)$ $\langle\sqcap$ monotonic$\rangle$
(2) $(r;\ p) \sqsubseteq (r;\ q)$ and $(p;\ r) \sqsubseteq (q;\ r)$ $\langle;$ monotonic$\rangle$

We will not state monotonicity laws explicitly for the remaining operators of our language.

3.7 Unbounded Nondeterminism

Here we generalise the operators $\sqcap$ and $\sqcup$ to take an arbitrary set of programs, say $\mathcal{P}$, as argument. $\sqcup\mathcal{P}$ denotes the least upper bound of $\mathcal{P}$; it is defined by

Definition 3.2 (Least upper bound)

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

■

which states that p refines the least upper bound of the set $\mathcal{P}$ if and only if, for all X in $\mathcal{P}$, p refines X. The greatest lower bound of $\mathcal{P}$, denoted by $\sqcap\mathcal{P}$, is defined in a similar way.

Definition 3.3 (Greatest lower bound)

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

■

Let $\mathcal{U}$ be the set of all programs, and $\emptyset$ be the empty set. Then we have:

$$\begin{aligned} \sqcup\emptyset &= \bot = \sqcap\mathcal{U} \\ \sqcap\emptyset &= \top = \sqcup\mathcal{U} \end{aligned}$$

From the above we can easily show that sequential composition does not distribute rightward through the least upper bound or the greatest lower bound in general, since we have:

$$\begin{aligned} \bot;\ \sqcap\emptyset &= \bot \neq \sqcap\emptyset \\ \top;\ \sqcup\emptyset &= \top \neq \sqcup\emptyset \end{aligned}$$

The rightward distribution of sequential composition through these operators is used below to define Dijkstra's healthiness conditions. However, the leftward distribution is valid in general, and can be verified by considering programs as predicate transformers. In the following, the notation $\{X : b : F(X)\}$ should be read as: the set of elements $F(X)$ for all X in the range specified by b.

Law 3.7.1

(1) $\sqcup\mathcal{P};\ p = \sqcup\{X : X \in \mathcal{P} : (X;\ p)\}$ ⟨;−⊔ left dist⟩

(2) $\sqcap\mathcal{P};\ p = \sqcap\{X : X \in \mathcal{P} : (X;\ p)\}$ ⟨;−⊓ left dist⟩

It is also possible to verify that the lattice of programs (considered as predicate transformers) is distributive.

Law 3.7.2

(1) $(\sqcup\mathcal{P}) \sqcap p = \sqcup\{X : X \in \mathcal{P} : (X \sqcap p)\}$ ⟨⊓−⊔ dist⟩

(2) $(\sqcap\mathcal{P}) \sqcup p = \sqcap\{X : X \in \mathcal{P} : (X \sqcup p)\}$ ⟨⊔−⊓ dist⟩

As discussed in the previous chapter, among all the predicate transformers Dijkstra singles out the implementable ones by certain healthiness conditions. Here we show that these conditions can be formulated as equations relating operators of our language.

1. $p;\ \bot = \bot$ — p is non-miraculous
2. $p;\ \sqcap\mathcal{P} = \sqcap\{X : X \in \mathcal{P} : (p;\ X)\}$ — p is conjunctive
   for all (non-empty) sets of programs $\mathcal{P}$
3. $p;\ \sqcup\mathcal{P} = \sqcup\{X : X \in \mathcal{P} : (p;\ X)\}$ — p is disjunctive
   for all (non-empty) sets of programs $\mathcal{P}$
4. $p;\ \sqcup\{i : i \geq 0 : q_i\} = \sqcup\{i : i \geq 0 : p;\ q_i\}$ — p is continuous
   provided $q_i \sqsubseteq q_{i+1}$ for all $i \geq 0$

We say that a program p is *universally* conjunctive if the second equation above bolds for all sets of programs $\mathcal{P}$ (possibly empty). Similarly, if the third equation holds for all $\mathcal{P}$, we say that p is universally disjunctive.

3.8 Recursion

Let X stand for the name of the recursive program we wish to construct, and let $F(X)$ define the intended behaviour of tbe program, for a given context F. If F is defined solely in terms of the notations introduced already, it follows by structural induction that F is monotonic:

$$p \sqsubseteq q \Rightarrow F(p) \sqsubseteq F(q)$$

Actually, this will remain true for the commands which will be introduced later, since they are all monotonic. The following two properties, due to Knaster-Tarski [72], say that $\mu X \bullet F(X)$ is a solution of the equation $X = F(X)$; furthermore, it is the least solution.

Law 3.8.1 $\mu X \bullet F(X) \; = \; F(\mu X \bullet F(X))$ $\langle\mu$ fixed point$\rangle$

Law 3.8.2 $F(Y) \sqsubseteq Y \; \Rightarrow \; \mu X \bullet F(X) \sqsubseteq Y$ $\langle\mu$ least fixed point$\rangle$

3.9 Approximate Inverses

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

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

Then G is the inverse of F, and vice-versa. Therefore $G(F(X)) = X = F(G(X))$, for all X. It is well-known, however, that a function has an inverse if and only if it is bijective. As the set of bijective functions is relatively small tbis makes the notion of inverse rather limited. The standard approach is to generalise the notion of inverse functions as follows.

Definition 3.4 (Approximate inverses)
Let F and F^{-1} be functions on programs such that, for all X and Y

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

Then we call F the *weakest inverse* of F^{-1}, and F^{-1} the *strongest inverse* of F. The pair (F, F^{-1}) is called a *Galois connection*. ■

Weakest inverses have been used in [47, 46] for top-down design of programs. In particular, the left and right weakest inverses of sequential composition are defined together with a

calculus of program development. Broadly, the aim is to decompose a task (specification) r into two subtasks p and q, such that

$$r \sqsubseteq p;\ q$$

The method described in [46] allows one to calculate the *weakest* specification that must be satisfied by one of the components p or q when the other one is known. For example, one can calculate the weakest specification of p from q and r. It is denoted by $q\backslash r$ and satisfies $r \sqsubseteq (q\backslash r);\ q$. This is called the *weakest prespecification*. Dually, r/p is the weakest specification of component q satisfying $r \sqsubseteq p;\ (r/p)$. It is named the *weakest postspecification*.

Strongest inverses of language constructs are less commonly used. This is perhaps a consequence of the fact that they exist only for operators which are *universally disjunctive* (see theorem below). Gardiner and Pandya [25] have suggested a method to reason about recursion based on the notion of strongest inverses, which they call *weak-op-inverses*. Below we consider some of the properties of strongest inverses that are proved in the cited paper. A similar treatment is given in [45].

Before presenting the properties of strongest inverses we review two basic definitions. F is *universally conjunctive* if for all $\mathcal{P}$

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

Similarly, F is *universally disjunctive* if for all $\mathcal{P}$

$$F(\sqcup \mathcal{P}) \ = \ \sqcup\{X : X \in \mathcal{P} : F(X)\}$$

Theorem 3.1 (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

$$F^{-1}(Y) \ \stackrel{def}{=} \ \sqcup\{X : F(X) \sqsubseteq Y : X\}$$

(5) F^{-1} is universally conjunctive if it exists. ■

The following lemma shows that sequential composition has a strongest inverse in its first argument. As noted in [25] this allows a concise proof (given later in this chapter) of an important property about composition of iteration commands.

Lemma 3.1 (Strongest inverse of ;)
Let $F(X) \stackrel{def}{=} (X;\ p)$. Then it has a strongest inverse which we denote by $F^{-1}(X) \stackrel{def}{=} X \stackrel{\sqcup}{;} p$. Furthermore, for all X

$$(X \stackrel{\sqcup}{;} p);\ p \sqsubseteq p$$

Proof: From Law (; $-\sqcup$ left dist)(3.7.1) it follows that F is disjunctive. Consequently, from Theorem 3.1(4), it has a strongest inverse. The inequation follows from Theorem 3.1(3). ∎

3.10 Simulation

In the previous section we discussed the inverse of functions on programs. Here we consider the inverse of programs themselves. An inverse of a program S is a program T that satisfies

$$S;\ T\ =\ \textsf{skip}\ =\ T;\ S$$

That means that running S followed by T or T followed by S is the same as not running any program at all, since **skip** has no effect whatsoever.

Inversion of programs has been previously discussed by Dijkstra [22] and Gries [35]. A more formal approach to program inversion is given in [16], which defines proof rules for inverting programs written in Dijkstra's language. A common feature of these works is the use of the notion of *exact* inverse given above. As mentioned for functions, this notion of inverse is rather limited. Following a similar idea to that of the previous section, we adopt a weaker definition of program inversion.

Definition 3.5 (Simulation)
Let S and S^{-1} be programs such that

$$(S;\ S^{-1})\ \sqsubseteq\ \textsf{skip}\ \sqsubseteq\ (S^{-1};\ S)$$

Then the pair (S,S^{-1}) is called a *simulation*, S^{-1} is the *strongest inverse* of S, whereas S is the *weakest inverse* of S^{-1}. ∎

A very simple example of a simulation is the pair $(\bot,\ \top)$ since

$$(\bot;\ \top)\ =\ \bot\ \sqsubseteq\ \textsf{skip}\ \sqsubseteq\ \top\ =\ (\top;\ \bot)$$

Simulations are useful for calculation in general. When carrying out program transformation, it is not rare to reach situations where a program followed by its inverse (that is, $S;\ S^{-1}$ or $S^{-1};\ S$) appears as a subterm of the program being transformed. Thus, from the definition of simulation, it is possible to eliminate subterms of the above form by replacing them with **skip** (of course, this is only valid for inequational reasoning). This will be illustrated in many of the proofs in the next two chapters where we give further examples of simulations.

But the most valuable use that has been made of the concept of simulation is for data refinement. This was discussed in some detail in the previous chapter where we introduced the concepts of *encoding* and *decoding* programs which form a simulation pair. The distributivity properties of simulations given below are particularly useful to prove the

correctness of the change of data representation phase of the compilation process, where the abstract space of the source program is replaced by the concrete state of the target machine. The appropriate encoding and decoding programs will be defined when the need arises.

A detailed discussion on simulations can be found in [5] (where it is called *inverse commands*) and in [45]. Here we present some of the properties of simulation. As should be expected, these are similar to the ones given in the previous section.

Theorem 3.2 (Simulation)
Let S be a program. The following properties hold:
(1) S^{-1} is unique if it exists.
(2) S^{-1} exists if and only if S is universally disjunctive.
(3) S^{-1} is universally conjunctive if it exists. ■

We define the following abbreviations.

Definition 3.6 (Simulation functions)
Let (S, S^{-1}) be a simulation. We use S and S^{-1} themselves as functions defined by

$$S(X) \stackrel{def}{=} S;\ X;\ S^{-1}$$
$$S^{-1}(X) \stackrel{def}{=} S^{-1};\ X;\ S$$

■

The next theorem shows that the concepts of simulation and approximate inverse are closely related.

Theorem 3.3 (Lift of simulation)
Let S and S^{-1} be simulation functions as defined above. Then S^{-1} is the strongest inverse of S. Furthermore, from Theorem 3.1 (Strongest inverses) we have

$$S(S^{-1}(X)) \sqsubseteq X \sqsubseteq S^{-1}(S(X))$$

■

The following theorem shows how simulation functions distribute through all the language operators introduced so far, with a possible improvement in the distributed result.

Theorem 3.4 (Distributivity of simulation functions)
(1) $S(\bot) = \bot$
(2) $S(\top) \sqsubseteq \top$
(3) $S(\mathsf{skip}) \sqsubseteq \mathsf{skip}$
(4) $S(X;\ Y) \sqsubseteq S(X);\ S(Y)$
(5) $S(\sqcap \mathcal{P}) \sqsubseteq \sqcap \{X : X :\in \mathcal{P} : S(X)\}$
(6) $S(\sqcup \mathcal{P}) = \sqcup \{X : X :\in \mathcal{P} : S(X)\}$
(7) $S(\mu\, X \bullet F(X)) \sqsubseteq \mu\, X \bullet S(F(S^{-1}(X)))$ ■

3.11 Assumption and Assertion

The assumption of a condition b, designated as $b^{\top}$, can be regarded as a miraculous test: it leaves the state unchanged (behaving like skip) if b is true; otherwise it behaves like $\top$. The assertion of b, $b_{\perp}$, also behaves like skip when b is true; otherwise it fails, behaving like $\perp$.

The intended purpose of assumptions and assertions is to give *preconditions* and *postconditions*, respectively, the status of programs. For example,

$$a^{\top};\ p;\ b_{\perp}$$

is used to express the fact that the assumption of a is an obligation placed on the environment of the program p. If the environment fails to provide a state satisfying a, $a^{\top}$ behaves like a miracle; this saves the programmer from dealing with states not satisfying a, since no program can implement $\top$. On the other hand, an assertion is an obligation placed on the program itself. If p fails to make b true on its completion, it ends up behaving like abort.

The first three laws formally state that the assumption and the assertion of a true condition are equivalent to skip, that the assumption of a false condition leads to miracle and that the assertion of a false condition leads to abortion.

Law 3.11.1 $\mathsf{true}^{\top}\ =\ \mathsf{true}_{\perp}\ =\ \mathsf{skip}$ $\langle b^{\top}, b_{\perp}$ true cond$\rangle$

Law 3.11.2 $\mathsf{false}^{\top}\ =\ \top$ $\langle b^{\top}$ false cond$\rangle$

Law 3.11.3 $\mathsf{false}_{\perp}\ =\ \perp$ $\langle b_{\perp}$ false cond$\rangle$

Two consecutive assumptions can be combined, giving rise to an assumption of the conjunction of the original conditions; this obviously means that if any of the conditions is not satisfied, the result will be miraculous. An analogous law holds for assertions.

Law 3.11.4 $(a^{\top};\ b^{\top})\ =\ (a \wedge b)^{\top}\ =\ (a^{\top} \sqcup b^{\top})$ $\langle b^{\top}$ conjunction$\rangle$

Law 3.11.5 $(a_{\perp};\ b_{\perp})\ =\ (a \wedge b)_{\perp}\ =\ (a_{\perp} \sqcap b_{\perp})$ $\langle b_{\perp}$ conjunction$\rangle$

The assumption of the disjunction of two conditions will behave like a miracle if and only if none of the conditions are satisfied. There is a similar law for assertions.

Law 3.11.6 $(a \vee b)^{\top}\ =\ (a^{\top} \sqcap b^{\top})$ $\langle b^{\top}$ disjunction$\rangle$

Law 3.11.7 $(a \vee b)_{\perp}\ =\ (a_{\perp} \sqcup b_{\perp})$ $\langle b_{\perp}$ disjunction$\rangle$

It does not matter if a choice is made before or after an assumption (or an assertion) is executed.

Law 3.11.8 $b^{\top};\ (p \sqcap q) \ = \ (b^{\top};\ p) \sqcap (b^{\top};\ q)$ $\quad$ ($b^{\top}$ − $\sqcap$ dist)

Law 3.11.9 $b_{\perp};\ (p \sqcap q) \ = \ (b_{\perp};\ p) \sqcap (b_{\perp};\ q)$ $\quad$ ($b_{\perp}$ − $\sqcap$ dist)

The next law states that $(b_{\perp},\ b^{\top})$ is a simulation.

Law 3.11.10 $(b_{\perp};\ b^{\top}) \ = \ b_{\perp} \ \sqsubseteq \ \mathsf{skip} \ \sqsubseteq \ b^{\top} \ = \ (b^{\top};\ b_{\perp})$ $\quad$ ($b_{\perp} - b^{\top}$ simulation)

An assumption commutes with an arbitrary program p in the following sense. (A similar law holds for assertions, but we do not need it here.)

Law 3.11.11 If the free variables of b are not assigned by p
$(p;\ b^{\top}) \ \sqsubseteq \ (b^{\top};\ p)$ $\quad$ ($b^{\top};\ p$ commute)

The inequality occurs when b is **false** and p is $\perp$, in which case the left-hand side reduces to $\perp$ whereas the right-hand side reduces to $\top$.

3.12 Guarded Command

The standard notation $b \to p$ stands for a guarded command. If the guard b is true, the whole command behaves like p; otherwise it behaves like $\top$. This suggests that a guard has the same effect as an assumption of the given condition, which allows us to define a guarded command as follows.

Definition 3.7 (Guarded command)

$$b \to p \ \stackrel{def}{=} \ b^{\top};\ p$$

■

The laws of guarded commands can therefore be proved from the above definition and the laws of sequential composition and assumptions.

Law 3.12.1 $(\mathsf{true} \to p) \ = \ p$ $\quad$ ($\to$ true guard)

Law 3.12.2 $(\mathsf{false} \to p) \ = \ \top$ $\quad$ ($\to$ false guard)

Guards can be unnested by taking their conjunction.

Law 3.12.3 $a \to (b \to p) \ = \ (a \wedge b) \to p$ $\quad$ ($\to$ guard conjunction)

Guards distribute over $\sqcap$.

Law 3.12.4 $b \to (p \sqcap q) \;=\; (b \to p) \sqcap (b \to q)$ ⟨guard − ⊓ dist⟩

The demonic choice of guarded commands can be written as a single guarded command by taking the disjunction of their guards. This is easily derived from the last two laws.

Law 3.12.5 $(a \to p \;\sqcap\; b \to q) \;=\; (a \vee b) \to (a \to p \;\sqcap\; b \to q)$
⟨→ guard disjunction1⟩

Proof:

$$
\begin{array}{ll}
 & RHS \\
= & \{\langle \text{guard} - \sqcap \text{ dist}\rangle(3.12.4)\} \\
 & (a \vee b) \to (a \to p) \;\sqcap\; (a \vee b) \to (b \to q) \\
= & \{\langle \to \text{ guard conjunction}\rangle(3.12.3)\} \\
 & LHS
\end{array}
$$

■

When p and q above are the same program, we have:

Law 3.12.6 $(a \to p) \sqcap (b \to p) \;=\; (a \vee b) \to p$ ⟨→ guard disjunction2⟩

Sequential composition distributes leftward through guarded commands.

Law 3.12.7 $(b \to p);\; q \;=\; b \to (p;\; q)$ ⟨; − → left dist⟩

3.13 Guarded Command Set

Our main use of guarded commands is to model the possible actions of a deterministic executing mechanism. The fact that the mechanism can perform one of n actions, according to its current state, can be modelled by a program fragment of the form

$$b_1 \to action_1 \;\sqcap \ldots \sqcap\; b_n \to action_n$$

provided $b_1, \ldots, b_n$ are pairwise disjoint. Instead of mentioning this disjointness condition explicitly, we will write the above as

$$b_1 \to action_1 \;\Box \ldots \Box\; b_n \to action_n$$

Strictly, $\Box$ is not a new operator of our language. It is just syntactic sugar to improve conciseness and readability. Any theorem that uses $\Box$ can be readily restated in terms of $\sqcap$ with the associated disjointness conditions. As an example we have the following law.

If one of the guards of a guarded command set holds initially, the associated command will always be selected for execution.

Law 3.13.1 $a \rightarrow (a \rightarrow p \; \Box \; b \rightarrow q) \; = \; a \rightarrow p$ $\langle \Box$ elim$\rangle$
Proof: The proof relies on the (implicit) assumption that a and b are disjoint (or $a \wedge b =$ false).

$$\begin{array}{rl} & a \rightarrow (a \rightarrow p \; \sqcap \; b \rightarrow q) \\ = & \{\langle \text{guard} - \sqcap \text{ dist}\rangle(3.12.4)\} \\ & a \rightarrow (a \rightarrow p) \; \sqcap \; a \rightarrow (b \rightarrow q) \\ = & \{\langle \rightarrow \text{ guard conjunction}\rangle(3.12.3)\} \\ & a \rightarrow p \; \sqcap \; \mathsf{false} \rightarrow q \\ = & \{\langle \rightarrow \text{ false guard}\rangle(3.12.2) \text{ and } \langle \sqcap - \top \text{ unit}\rangle(3.4.2)\} \\ & a \rightarrow p \end{array}$$

■

Other laws and theorems involving $\Box$ will be described as the need arises.

3.14 Conditional

A conditional command has the general syntax $p \lhd b \rhd q$ which is a concise form of the more usual notation

$\mathsf{if}\; b \;\mathsf{then}\; p \;\mathsf{else}\; q$

It can also be defined in terms of more basic operators.

Definition 3.8 (Conditional)

$$(p \lhd b \rhd q) \stackrel{def}{=} (b \rightarrow p \; \Box \; \neg b \rightarrow q)$$

■

The most basic property of a conditional is that its left branch is executed if the condition holds initially; otherwise its right branch is executed.

Law 3.14.1 $(a \wedge b)^\top; \; (p \lhd b \vee c \rhd q) \; = \; (a \wedge b)^\top; \; p$ $\langle \lhd \; \rhd$ true cond$\rangle$

Law 3.14.2 $(a \wedge \neg b)^\top; \; (p \lhd b \wedge c \rhd q) \; = \; (a \wedge \neg b)^\top; \; q$ $\langle \lhd \; \rhd$ false cond$\rangle$

The left branch of a conditional can always be preceded by an assumption of the condition. Similarly, to precede the right branch by an assumption of the negation of the condition has no effect.

Law 3.14.3 $(b^\top; \; p \lhd b \rhd q) \; = \; (p \lhd b \rhd q) \; = \; (p \lhd b \rhd \neg b^\top; \; q)$ $\langle \lhd \; \rhd$ void $b^\top\rangle$

If the two branches are the same program, the conditional can be eliminated.

Law 3.14.4 $p \lhd b \rhd p \;=\; p$ $\quad$ $(\lhd\ \rhd$ idemp$)$

Guard distributes through the conditional.

Law 3.14.5 $a \rightarrow (p \lhd b \rhd q) \;=\; (a \rightarrow p) \lhd b \rhd (a \rightarrow q)$ $\quad$ $(\text{guard} - \lhd\ \rhd$ dist$)$

Sequential composition distributes leftward through the conditional.

Law 3.14.6 $(p \lhd b \rhd q);\ r \;=\; (p;\ r \lhd b \rhd q;\ r)$ $\quad$ $(; - \lhd\ \rhd$ left dist$)$

The following two laws allow the elimination of nested conditionals in certain cases.

Law 3.14.7 $p \lhd b \rhd (p \lhd c \rhd q) \;=\; p \lhd b \vee c \rhd q$ $\quad$ $(\lhd\ \rhd$ cond disjunction$)$

Law 3.14.8 $(p \lhd b \rhd q) \lhd c \rhd q \;=\; p \lhd b \wedge c \rhd q$ $\quad$ $(\lhd\ \rhd$ cond conjunction$)$

We have considered assumptions and assertions as primitive commands and have defined guarded commands and the conditional in terms of them. The following equations show that an alternative could be to consider the conditional as a constructor and regard assumptions, assertions and guarded commands as special cases. These are not stated as laws because they are unnecessary in our proofs.

$$\begin{aligned} b_\perp &= \textsf{skip} \lhd b \rhd \perp \\ b^\top &= \textsf{skip} \lhd b \rhd \top \\ b \rightarrow p &= p \lhd b \rhd \top \end{aligned}$$

3.15 Assignment

The command $x := e$ stands for a multiple assignment where x is a list of distinct variables and e is an equal-length list of expressions. The components of e are evaluated and simultaneously assigned to the corresponding (same position) components of x. For example,

$$x, y := y, x$$

swaps the values of x and y. For simplicity, we assume that the evaluation of an expression always delivers a result, so the assignment will always terminate. Furthermore, the validity of most of the laws relies on the fact that expression evaluation does not change the value of any variable; that is, no *side-effect* is allowed.

Obviously, the assignment of the value of a variable to itself does not change anything.

Law 3.15.1 $(x := x) \;=\; \textsf{skip}$ $\quad$ $(:= \textsf{skip})$

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

Law 3.15.2 $(x, y := e, y) \;=\; (x := e)$ $\langle := \text{ identity}\rangle$

The list of variables and expressions may be subjected to the same permutation without changing the effect of the assignment.

Law 3.15.3 $(x, y, z := e, f, g) \;=\; (y, x, z := f, e, g)$ $\langle := \text{ sym}\rangle$

The sequential composition of two assignments to the same variables is easily combined to a single assignment.

Law 3.15.4 $(x := e;\; x := f) \;=\; (x := f[x \leftarrow e])$ $\langle := \text{ combination}\rangle$

Recall that $f[x \leftarrow e]$ denotes the substitution of e for every free occurrence of x in f.

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

Law 3.15.5 $(x = e) \rightarrow (y := f) \;=\; (x = e) \rightarrow (y := f[x \leftarrow e])$ $\langle := \text{ substitution}\rangle$

Assignment is universally conjunctive.

Law 3.15.6 $x := e;\; \sqcap \mathcal{P} \;=\; \sqcap \{X : X \in \mathcal{P} : (x := e;\; X)\}$ $\langle := - \sqcap \text{ right dist}\rangle$

Assignment distributes rightward through a conditional, replacing occurrences of the assigned variables in the condition by the corresponding expressions.

Law 3.15.7 $x := e;\; (p \lhd b \rhd q) \;=\; (x := e;\; p) \lhd b[x \leftarrow e] \rhd (x := e;\; q)$
$\langle := - \lhd \rhd \text{ right dist}\rangle$

Similarly, assignment commutes with an assertion in the following sense.

Law 3.15.8 $(x := e;\; b_\perp) \;=\; (b[x \leftarrow e])_\perp;\; x := e$ $\langle := -b_\perp \text{ commutation}\rangle$

3.16 Generalised Assignment

The notation $x :\in b$ stands for a generalised assignment command. Whenever possible, x is assigned an arbitrary value that makes the condition b hold; but if no such value exists, the assignment behaves like $\top$.

Law 3.16.1 $(x :\in \mathsf{false}) \;=\; \top$ $\langle :\in \text{ false cond}\rangle$

On the other hand, a true condition imposes no constraints on the final value of x. In this case, the generalised assignment is less deterministic than **skip**, because it might leave everything unchanged.

Law 3.16.2 $(x :\in \mathsf{true}) \sqsubseteq \mathsf{skip}$ $\langle :\in$ true cond$\rangle$

To follow a generalised assignment by an assumption of the same condition has no effect: if the assignment establishes the condition, the assumption behaves like **skip**; otherwise, the assignment itself (and consequently, its composition with the assumption) behaves like $\top$.

Law 3.16.3 $x :\in b;\ b^{\top} = x :\in b$ $\langle :\in$ void $b^{\top}\rangle$

A similar law holds for assertions.

Law 3.16.4 $x :\in b;\ b_{\perp} = x :\in b$ $\langle :\in$ void $b_{\perp}\rangle$

A generalised assignment is refined by an assumption of the same condition. The reason is that the final values of the variables of the assignment are arbitrary, whereas the assumption does not change the value of any variable. Actually, an assumption can be regarded as a generalised assignment to an empty list of variables.

Law 3.16.5 $(x :\in b) \sqsubseteq b^{\top}$ $\langle :\in$ refined by $b^{\top}\rangle$

Generalised assignment distributes rightward through the conditional, provided the following condition is observed.

Law 3.16.6 If x does not occur in b
$x :\in a;\ (p \lhd b \rhd q) = (x :\in a;\ p \lhd b \rhd x :\in a;\ q)$ $\langle :\in - \lhd\ \rhd$ right dist$\rangle$

In general, an assignment cannot be expressed in terms of a generalised assignment only. For example, there is no generalised assignment that corresponds to the assignment $x := x + 1$. The reason is that we have not introduced notation to allow the condition of a generalised assignment of the form $x :\in b$ to refer back to the initial value of x. But $x := e$ can always be written as a generalised assignment whenever the expression e does not mention x.

Law 3.16.7 If e does not mention x
$x :\in (x = e) = x := e$ $\langle :\in - :=$ conversion$\rangle$

If x and y are to be assigned arbitrary values (in sequence) to make a given condition hold, we can reduce the nondeterminism by ensuring that the same (arbitrary) value is assigned to both x and y.

Law 3.16.8 If b does not mention y
$(x :\in b;\ y :\in b[x \leftarrow y]) \sqsubseteq (x :\in b;\ y := x)$ $\langle :\in$ refined by $:=\rangle$

We can commute the order of execution of an assignment and an arbitrary program p, provided no interference occurs with the global variables.

Law 3.16.9 If no free variables of b nor x are assigned by p
$(p;\ x :\in b) \sqsubseteq (x :\in b;\ p)$ $\qquad\qquad\qquad\qquad$ $\langle x :\in b;\ p$ commute$\rangle$

The inequality occurs when p is $\bot$ and the assignment results in $\top$.

3.17 Iteration

We use $b*p$ to denote the iteration command. It is a concise form of the more conventional syntax

$$\textsf{while}\ b\ \textsf{do}\ p$$

Iteration can be defined as a special case of recursion.

Definition 3.9 (Iteration)

$$b * p \stackrel{def}{=} \mu X \bullet ((p;\ X) \lhd b \rhd \textsf{skip})$$

■

As iteration is a derived operator in our language, we are able to prove (rather than just postulate) some of its properties. This illustrates the modularity provided by the algebraic laws in developing more elaborate transformation strategies from the basic ones. These strategies are largely used in the next two chapters, substantially simplifying the proofs of normal form reduction.

If the condition b does not hold initially, the iteration $b * p$ behaves like skip; otherwise it behaves like p followed by the whole iteration.

Law 3.17.1 $(a \wedge \neg b)^\top;\ b * p = (a \wedge \neg b)^\top$ $\qquad\qquad\qquad\qquad$ $\langle *$ elim$\rangle$
Proof:

$$\begin{array}{ll} & LHS \\ = & \{\text{Definition 3.9(Iteration) and } \langle \mu \text{ fixed point}\rangle(3.8.1)\} \\ & (a \wedge \neg b)^\top;\ ((p;\ b * p) \lhd b \rhd \textsf{skip}) \\ = & \{\langle \lhd\ \rhd \text{ false cond}\rangle(3.14.2) \text{ and } \langle ;-\textsf{skip} \text{ unit}\rangle(3.3.1)\} \\ & RHS \end{array}$$

■

Law 3.17.2 $a^\top;\ (a \vee b) * p \ = \ a^\top;\ p;\ (a \vee b) * p$ (* unfold)
Proof:

$$
\begin{array}{ll}
 & LHS \\
= & \{\text{Definition 3.9(Iteration) and } (\mu \text{ fixed point})(3.8.1)\} \\
 & a^\top;\ ((p;\ (a \vee b) * p) \lhd (a \vee b) \rhd \mathsf{skip}) \\
= & \{(\lhd\ \rhd \text{ true cond})(3.14.1)\} \\
 & RHS
\end{array}
$$

■

A recurrent step in our proofs is to unfold an iteration and simplify the unfolded body when this is a guarded command set.

Law 3.17.3 Let $R = (a \rightarrow p \ \Box \ b \rightarrow q)$. Then
$a^\top;\ (a \vee b) * R \ = \ a^\top;\ p;\ (a \vee b) * R$ $(* - \Box$ unfold)
Proof: From (* unfold)(3.17.2) and ($\Box$ elim)(3.13.1). ■

A guarded command set within an iteration can be eliminated if the condition of the iteration allows only one of the guards to hold.

Law 3.17.4 Let $R = (a \rightarrow p \ \Box \ b \rightarrow q)$. Then $a * R \ = \ a * p$ $(* - \Box$ elim)
Proof:

$$
\begin{array}{ll}
 & LHS \\
= & \{\text{Definition 3.9(Iteration) and } (\lhd\ \rhd \text{ void } b^\top)(3.14.3)\} \\
 & \mu X \bullet ((a^\top;\ R;\ X) \lhd a \rhd \mathsf{skip}) \\
= & \{(\Box \text{ elim})(3.13.1)\} \\
 & \mu X \bullet ((a^\top;\ p;\ X) \lhd a \rhd \mathsf{skip}) \\
= & \{(\lhd\ \rhd \text{ void } b^\top)(3.14.3) \text{ and Definition 3.9(Iteration)}\} \\
 & RHS
\end{array}
$$

■

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

Law 3.17.5 Let $R = (a \rightarrow p \ \Box \ b \rightarrow q)$. If $r;\ (a \vee b) * R \ \sqsubseteq \ p;\ (a \vee b) * R$, then
$(a \vee b) * (a \rightarrow r \ \Box \ b \rightarrow q) \ \sqsubseteq \ (a \vee b) * R$ (* replace guarded command)
Proof:

$$
\begin{array}{ll}
 & RHS \\
= & \{\text{Definition 3.9(Iteration) and } (\mu \text{ fixed point})(3.8.1)\}
\end{array}
$$

$(R;\ RHS) \lhd a \vee b \rhd \mathbf{skip}$

$= \{\langle ; - \sqcap \text{ left dist}\rangle(3.7.1)\}$

$(a \rightarrow (p;\ RHS) \;\Box\; b \rightarrow (q;\ RHS)) \lhd a \vee b \rhd \mathbf{skip}$

$\sqsupseteq \{\text{Assumption}\}$

$(a \rightarrow (r;\ RHS) \;\Box\; b \rightarrow (q;\ RHS)) \lhd a \vee b \rhd \mathbf{skip}$

$= \{\langle ; - \sqcap \text{ left dist}\rangle(3.7.1)\}$

$((a \rightarrow r \;\Box\; b \rightarrow q);\ RHS) \lhd a \vee b \rhd \mathbf{skip}$

The final result follows from $\langle \mu$ least fixed point$\rangle$(3.8.2). ■

The following law establishes the connection between tail-recursion and iteration. Its proof illustrates the use of approximate inverses of programming constructs.

Law 3.17.6 $(b * p);\ q \;=\; \mu X \bullet ((p;\ X) \lhd b \rhd q)$ $\langle * - \mu$ tail recursion$\rangle$

Proof: $(LHS \sqsupseteq RHS)$:

$\{\text{Definition 3.9(Iteration) and } \langle \mu \text{ fixed point}\rangle(3.8.1)\}$

$LHS \;=\; ((p;\ b * p) \lhd b \rhd \mathbf{skip});\ q$

$\equiv \{\langle ; - \lhd\ \rhd \text{ left dist}\rangle(3.14.6) \text{ and } \langle ; -\mathbf{skip} \text{ unit}\rangle(3.3.1)\}$

$LHS \;=\; (p;\ LHS) \lhd b \rhd q$

$\Rightarrow \{\langle \mu \text{ least fixed point}\rangle(3.8.2)\}$

$LHS \sqsupseteq RHS$

$(RHS \sqsupseteq LHS)$:

$\{\langle \mu \text{ fixed point}\rangle(3.8.1)\}$

$RHS \;=\; (p;\ RHS) \lhd b \rhd q$

$\Rightarrow \{\text{From Lemma 3.1(Strongest inverse of ;) we have } (RHS \overset{\cup}{;} q);\ q \sqsubseteq RHS\}$

$RHS \sqsupseteq (p;\ (RHS \overset{\cup}{;} q);\ q) \lhd b \rhd q$

$\equiv \{\langle ; - \lhd\ \rhd \text{ left dist}\rangle(3.14.6) \text{ and } \langle ; -\mathbf{skip} \text{ unit}\rangle(3.3.1)\}$

$RHS \sqsupseteq ((p;\ (RHS \overset{\cup}{;} q)) \lhd b \rhd \mathbf{skip});\ q$

$\equiv \{\text{Definition 3.4(Approximate inverses)}\}$

$(RHS \overset{\cup}{;} q) \sqsupseteq (p;\ (RHS \overset{\cup}{;} q)) \lhd b \rhd \mathbf{skip}$

$\Rightarrow \{\langle \mu \text{ least fixed point}\rangle(3.8.1) \text{ and Definition 3.9(Iteration)}\}$

$(RHS \overset{\cup}{;} q) \sqsupseteq b * p$

$\equiv \{\text{Definition 3.4(Approximate inverses)}\}$

$RHS \sqsupseteq LHS$

■

The following law is surprisingly important, mainly in proving the correctness of the normal form reduction of sequential composition. Its proof without assuming continuity of the language operators is originally due to Gardiner and Pandya [25].

Law 3.17.7 $(b * p);\ (b \vee c) * p \ = \ (b \vee c) * p$ ($*$ sequence)

Proof: $(RHS \ \sqsupseteq \ LHS)$:

$$
\begin{array}{ll}
 & \{(\lhd\ \rhd \text{ idemp})(3.14.4)\} \\
 & RHS \ = \ RHS \lhd b \rhd RHS \\
\equiv & \{\text{Definition 3.9(Iteration) and } (\mu \text{ fixed point})(3.8.1)\} \\
 & RHS \ = \ ((p;\ RHS) \lhd b \vee c \rhd \mathsf{skip}) \lhd b \rhd ((b \vee c) * p) \\
\equiv & \{(\lhd\ \rhd \text{ void } b^{\top})(3.14.3) \text{ and } (* \text{ elim})(3.17.1)\} \\
 & RHS \ = \ ((p;\ RHS) \lhd b \vee c \rhd (b \vee c) * p) \lhd b \rhd ((b \vee c) * p) \\
\equiv & \{(\lhd\ \rhd \text{ cond conjunction})(3.14.8)\} \\
 & RHS \ = \ (p;\ RHS) \lhd b \rhd ((b \vee c) * p) \\
\Rightarrow & \{(\mu \text{ least fixed point})(3.8.2)\} \\
 & RHS \ \sqsupseteq \ \mu X \bullet (p;\ X) \lhd b \rhd ((b \vee c) * p) \\
\equiv & \{(* - \mu \text{ tail recursion})(3.17.6)\} \\
 & RHS \ \sqsupseteq \ (b * p);\ ((b \vee c) * p) \\
\equiv & RHS \ \sqsupseteq \ LHS
\end{array}
$$

$(LHS \sqsupseteq RHS)$:

$$
\begin{array}{ll}
 & \{\text{Definition 3.9(Iteration) and } (\mu \text{ fixed point})(3.8.1)\} \\
 & LHS \ = \ ((q;\ (b * p)) \lhd b \rhd \mathsf{skip});\ (b \vee c) * p \\
\equiv & \{(; - \lhd\ \rhd \text{ left dist})(3.14.6) \text{ and } (; -\mathsf{skip} \text{ unit})(3.3.1)\} \\
 & LHS \ = \ (p;\ LHS) \lhd b \rhd RHS \\
\equiv & \{(\mu \text{ fixed point})(3.8.1)\} \\
 & LHS \ = \ (p;\ LHS) \lhd b \rhd ((p;\ RHS) \lhd b \vee c \rhd \mathsf{skip}) \\
\Rightarrow & \{(\lhd\ \rhd \text{ cond disjunction})(3.14.7) \text{ and } LHS \sqsubseteq RHS\} \\
 & LHS \ \sqsupseteq \ (p;\ LHS) \lhd b \vee c \rhd \mathsf{skip} \\
\Rightarrow & \{\text{Definition 3.9(Iteration) and } (\mu \text{ least fixed point})(3.8.2)\} \\
 & LHS \ \sqsupseteq \ RHS
\end{array}
$$

■

3.18 Static Declaration

The notation $\mathsf{dec}\, x \bullet p$ declares the list of distinct variables x for use in the program p (the scope of the declaration). Local blocks of this form may appear anywhere a program is expected.

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

Law 3.18.1 If x and y have no variables in common

$\mathsf{dec}\, x \bullet (\mathsf{dec}\, y \bullet p) \ = \ \mathsf{dec}\, x, y \bullet p$ (dec assoc)

Nor does it matter in which order they are declared.

Law 3.18.2 $\mathbf{dec}\, x \bullet (\mathbf{dec}\, y \bullet p) \;=\; \mathbf{dec}\, y \bullet (\mathbf{dec}\, x \bullet p)$ (dec sym)

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

Law 3.18.3 If x is not free in p
$\mathbf{dec}\, x \bullet p \;=\; p$ (dec elim)

One can change the name of a bound variable, provided the new name is not used for a free variable. This law is normally stated as follows:

$$\mathbf{dec}\, x \bullet p \;=\; \mathbf{dec}\, y \bullet p[x \leftarrow y] \quad \text{provided } y \text{ is not free in } p$$

where the clashes of y with bound variables of p are dealt with by the renaming implicit in the substitution operator. However, this law justifies transformations which are not always valid (in the context of our language). For example, consider tbe recursive program

$$(1) \quad \mathbf{dec}\, x \bullet (\mu\ X \bullet F(\mathbf{dec}\, x \bullet X))$$

Because of static scope rules, any free occurrence of x in F is bound to the outer declaration of x. The inner declaration of x has no effect, since its scope does not include F, and therefore it can be eliminated:

$$(2) \quad \mathbf{dec}\, x \bullet (\mu\ X \bullet F(X))$$

However, renaming of bound variables as stated above allows (1) to be transformed into (assuming that y is not free in F)

$$\mathbf{dec}\, x \bullet (\mu\ X \bullet F(\mathbf{dec}\, y \bullet X[x \leftarrow y]))$$

which is clearly distinct from (2) if there are free occurrences of x in F. The inconsistency arises from the fact that the application of the law identified the occurrence of x bound to the inner declaration with the (possible) occurrences bound to the outer declaration; this violates the concept of static scoping. One way to avoid the problem is to add an extra condition to the law about renaming of local variables. This requires the following concept:

Definition 3.10 (Contiguous scope)
We say that a variable x has a ***contiguous*** scope in a program p if

- p contains no free program identifiers (standing for call commands) or
- if X is free in p, then x is not free in (the program defining) X.

■

The concept also applies when X is the name of a procedure, which will be dealt with in Chapter 5. The law then becomes:

Law 3.18.4 If y is not free in p and x has a contiguous scope in p, then
$\mathbf{dec}\, x \bullet p \;=\; \mathbf{dec}\, y \bullet p[x \leftarrow y]$ $\langle$**dec** rename$\rangle$

We can ensure that programs will always have contiguous scope (with respect to any local variable) by requiring that nested declarations always use distinct names for variables (which are also distinct from the names used for global variables). When applying the above law we will assume that the condition of contiguous scope is always satisfied.

The value of a declared variable is totally arbitrary. Therefore initialisation of a variable may reduce nondeterminism.

Law 3.18.5
(1) $\mathbf{dec}\, x \bullet p \;\sqsubseteq\; \mathbf{dec}\, x \bullet x := e;\ p$ $\langle$**dec**$-:=$ initial value$\rangle$
(2) $\mathbf{dec}\, x \bullet p \;\sqsubseteq\; \mathbf{dec}\, x \bullet x :\in b;\ p$ $\langle$**dec**$-:\in$ initial value$\rangle$

An assignment to a variable just before the end of its scope is irrelevant. But a generalised assignment cannot be completely ignored, since it may result in a miracle.

Law 3.18.6
(1) $\mathbf{dec}\, x \bullet p \;=\; \mathbf{dec}\, x \bullet p;\ x := e$ $\langle$**dec**$-:=$ final value$\rangle$
(2) $\mathbf{dec}\, x \bullet p \;\sqsubseteq\; \mathbf{dec}\, x \bullet p;\ x :\in b$ $\langle$**dec**$-:\in$ final value$\rangle$

The scope of a variable may be increased without effect, provided that it does not interfere with the other variables with the same name. Thus each of the programming constructs has a distribution law with declaration. For example, if one of the arguments of the sequential composition operator declares the variable x then the scope of the declaration can be extended with the other component, provided there is no capture of free variables.

Law 3.18.7 If x is not free in q
(1) $(\mathbf{dec}\, x \bullet p);\ q \;=\; \mathbf{dec}\, x \bullet p;\ q$ $\langle ;-$**dec** left dist$\rangle$
(2) $q;\ (\mathbf{dec}\, x \bullet p) \;=\; \mathbf{dec}\, x \bullet q;\ p$ $\langle ;-$**dec** right dist$\rangle$

When both arguments declare the same variable, the two declarations can be replaced with a single one.

Law 3.18.8 $(\mathbf{dec}\, x \bullet p);\ (\mathbf{dec}\, x \bullet q) \;\sqsubseteq\; \mathbf{dec}\, x \bullet p;\ q$ $\langle$**dec**$-;$ dist$\rangle$

But note that this may reduce nondeterminism. Consider the case where q is $y := x$. Then the final value of y on the left-hand side of the above inequation would be totally arbitrary. On the right-hand side, however, it may be the case that x was assigned a value in p; thus the final value of y would be that of x. In all cases, the right-hand side is at least as deterministic as the left-hand side.

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

Law 3.18.9 If x does not occur in a or b
$a \rightarrow (\mathsf{dec}\, x \bullet p) \,\square\, b \rightarrow (\mathsf{dec}\, x \bullet q) \;=\; \mathsf{dec}\, x \bullet a \rightarrow p \,\square\, b \rightarrow q$ $\qquad(\mathsf{dec} - \square \text{ dist})$

Law 3.18.10 If x does not occur in b
$(\mathsf{dec}\, x \bullet p) \lhd b \rhd (\mathsf{dec}\, x \bullet q) \;=\; \mathsf{dec}\, x \bullet p \lhd b \rhd q$ $\qquad(\mathsf{dec} - \lhd\ \rhd \text{ dist})$

Note that it is possible to deal with cases where x is only declared in one of the branches (and is not free in the other one) by using Law 3.18.3.

Declaration can also be moved outside an iteration, possibly reducing nondeterminism. As shown below, this law can be derived from more basic ones.

Law 3.18.11 If x does not occur in b
$b * (\mathsf{dec}\, x \bullet p) \;\sqsubseteq\; \mathsf{dec}\, x \bullet b * p$ $\qquad(\mathsf{dec} - * \text{ dist})$
Proof:

$$
\begin{array}{ll}
 & \{\text{Definition 3.9(Iteration) and } (\mu \text{ fixed point})(3.8.1)\} \\
 & RHS \;=\; \mathsf{dec}\, x \bullet (p;\ b * p) \lhd b \rhd \mathsf{skip} \\
\equiv & \{(\mathsf{dec} - \lhd\ \rhd \text{ dist})(3.18.10) \text{ and } (\mathsf{dec} \text{ elim})(3.18.3)\} \\
 & RHS \;=\; (\mathsf{dec}\, x \bullet p;\ b * p) \lhd b \rhd \mathsf{skip} \\
\Rightarrow & \{(\mathsf{dec}-;\ \text{dist})(3.18.8)\} \\
 & RHS \sqsupseteq ((\mathsf{dec}\, x \bullet p);\ RHS) \lhd b \rhd \mathsf{skip} \\
\Rightarrow & \{\text{Definition 3.9(Iteration) and } (\mu \text{ least fixed point})(3.8.2)\} \\
 & RHS \;\sqsupseteq\; LHS
\end{array}
$$

■

3.19 Dynamic Declaration

The command $\mathsf{var}\, x$ introduces a dynamic scope of x which extends up to

- the end of the static scope of x or
- the execution of the command $\mathsf{end}\, x$

whichever comes first.

An operational argument may help to clarify how the two kinds of declaration differ. The general idea is to associate an unbounded stack with each variable. One can think of a static declaration of x as introducing a new variable (which is assigned an arbitrary value)

witb its (implicit) unbounded stack which is initially empty. Rather than creating a new variable, the commands for dynamic declaration operate on this stack. The effect of $\mathsf{var}\, x$ is to push the current value of x onto the stack, assigning to x an arbitrary value; $\mathsf{end}\, x$ pops the stack and assigns the popped value to x. If the stack was empty, this value is arbitrary.

Recall from Section 2.6 that having separate commands to introduce and end the scope of a variable is an essential feature to define the encoding and decoding programs used in our approacb to data refinement: the encoding program introduces the abstract state and ends tbe scope of the concrete state, whereas the decoding program introduces the concrete state and ends the scope of the abstract state.

var and end obey laws similar to those of dec. Both var and end are associative in the sense described below.

Law 3.19.1 If x and y bave no variables in common

(1) $(\mathsf{var}\, x;\ \mathsf{var}\, y) \;=\; \mathsf{var}\, x, y$ (var assoc)

(2) $(\mathsf{end}\, x;\ \mathsf{end}\, y) \;=\; \mathsf{end}\, x, y$ ⟨end assoc⟩

The (dynamic) scope of a variable may be increased without effect, provided that this does not interfere with other free variables.

Law 3.19.2 If x is not free in p

(1) $p;\ \mathsf{var}\, x \;=\; \mathsf{var}\, x;\ p$ ⟨var change scope⟩

(2) $\mathsf{end}\, x;\ p \;=\; p;\ \mathsf{end}\, x$ ⟨end change scope⟩

Both var and end distribute rightward through the conditional, as long as no interference occurs with tbe condition.

Law 3.19.3 If b does not mention x

(1) $(\mathsf{var}\, x;\ p) \lhd b \rhd (\mathsf{var}\, x;\ q) \;=\; \mathsf{var}\, x;\ (p \lhd b \rhd q)$ ⟨var $- \lhd\ \rhd$ right dist⟩

(2) $(\mathsf{end}\, x;\ p) \lhd b \rhd (\mathsf{end}\, x;\ q) \;=\; \mathsf{end}\, x;\ (p \lhd b \rhd q)$ ⟨end $- \lhd\ \rhd$ right dist⟩

As explained above $\mathsf{var}\, x$ assigns an arbitrary value to x. The nondeterminism can be reduced by initialisation of x.

Law 3.19.4

(1) $\mathsf{var}\, x \;\sqsubseteq\; (\mathsf{var}\, x;\ x := e)$ ⟨var$-$:= initial value⟩

(2) $\mathsf{var}\, x \;\sqsubseteq\; (\mathsf{var}\, x;\ x :\in b)$ ⟨var$-$:∈ initial value⟩

An assignment to a variable just before the end of its scope is irrelevant. But a generalised assignment cannot be completely ignored, as it may result in a miracle.

Law 3.19.5

(1) $\mathsf{end}\, x \;=\; (x := e;\ \mathsf{end}\, x)$ ⟨end$-$:= final value⟩

(2) $\mathsf{end}\, x \;\sqsubseteq\; (x :\in b;\ \mathsf{end}\, x)$ ⟨end$-$:∈ final value⟩

The next two laws are essential for reasoning about data refinement. They are precisely the ones that assign the dynamic declaration semantics to **var** and **end**. The first law says that **end** x followed by **var** x leaves all variables but x unchanged; **var** x followed by **end** x has no effect (even on x). Therefore the pair (**end** x, **var** x) is a simulation.

Law 3.19.6 $(\mathsf{end}\ x;\ \mathsf{var}\ x) \sqsubseteq \mathsf{skip} = (\mathsf{var}\ x;\ \mathsf{end}\ x)$ (**end** − **var** simulation)

The second law postulates that the sequential composition of **end** x with **var** x has no effect whenever it is followed by an assignment to x that does not rely on the previous value of x.

Law 3.19.7 $(\mathsf{end}\ x;\ \mathsf{var}\ x;\ x :\in b) = x :\in b$ (**end** − **var** skip)

Observe that the syntax of static declaration (**dec**) promptly disallows the above two laws, since there is no separate construct to end the scope of a variable.

The following laws relate the two kinds of declaration. They formalise the intuitive meaning given in the beginning of this section.

If the first command in the scope of a static declaration of x is **var** x or **end** x, this command has no effect.

Law 3.19.8
(1) $\mathsf{dec}\ x \bullet \mathsf{var}\ x;\ p = \mathsf{dec}\ x \bullet p$ (**var** elim1)
(2) $\mathsf{dec}\ x \bullet \mathsf{end}\ x;\ p = \mathsf{dec}\ x \bullet p$ (**end** elim1)

First we give an operational justification of (1). Recall that a static declaration of x creates an implicit stack which is originally empty. Then the effect of **var** x on the left-hand side of (1) is to push the current value of x (which is arbitrary) onto this stack, and to assign an arbitrary value to x. But the value of x was already arbitrary; thus the assignment has no effect. Furthermore, the effect of pushing an arbitrary value onto the empty stack is also immaterial. The only command that may access this value is a subsequent **end** x which would assign an arbitrary value to x if the stack was empty anyway. The justification of (2) is simpler. As the stack associated with x is initially empty, the effect of **end** x is to assign an arbitrary value to x; but the value of x was already arbitrary.

As we said before, the dynamic scope of a variable x cannot extend further than its static scope. Therefore starting or ending a dynamic scope of x just before the end of its static scope is irrelevant.

Law 3.19.9
(1) $\mathsf{dec}\ x \bullet p;\ \mathsf{var}\ x = \mathsf{dec}\ x \bullet p$ (**var** elim2)
(2) $\mathsf{dec}\ x \bullet p;\ \mathsf{end}\ x = \mathsf{dec}\ x \bullet p$ (**end** elim2)

In some cases, there is no need to distinguish between a static and a dynamic scope of a given variable. To state this law we need two auxiliary concepts. One is that of a contiguous scope, as defined in the previous section (Definition 3.10). The other is defined below.

Definition 3.11 (Block-structure)
A program p is ***block-structured*** with respect to a variable x if each dynamic declaration of x in p ($\mathsf{var}\, x$) has a corresponding, statically determined, undeclaration ($\mathsf{end}\, x$). More precisely,

1. all programs that do not contain the commands $\mathsf{var}\, x$ or $\mathsf{end}\, x$ are block-structured with respect to x;
2. if p is block-structured with respect to x, so is $\mathsf{var}\, x;\ p;\ \mathsf{end}\, x$.

■

Then we have the following law.

Law 3.19.10 If p is block structured with respect to x, and x has a contiguous scope in p, then
$\mathsf{dec}\, x \bullet p\ =\ \mathsf{var}\, x;\ p;\ \mathsf{end}\, x$ (dec − (var, end) conversion)

From the above, we derive the following law about introduction of local declarations.

Law 3.19.11 If p is block structured with respect to x, and x has a contiguous scope in p, then
$x :\in b;\ p;\ x :\in c\ =\ (\mathsf{dec}\, x \bullet x :\in b;\ p);\ x :\in c$ (dec introduction)

Proof:

$$
\begin{array}{ll}
 & LHS \\
= & \{(\mathsf{end} - \mathsf{var}\ \mathsf{skip})(3.19.7)\} \\
 & \mathsf{end}\ x;\ \mathsf{var}\ x;\ x :\in b;\ p;\ \mathsf{end}\ x;\ \mathsf{var}\ x;\ x :\in c \\
= & \{(\mathsf{dec} - (\mathsf{var}, \mathsf{end})\ \mathrm{conversion})(3.19.10)\} \\
 & \mathsf{end}\ x;\ (\mathsf{dec}\, x \bullet x :\in b;\ p);\ \mathsf{var}\ x;\ x :\in c \\
= & \{(\mathsf{end}\ \mathrm{change\ scope})(3.19.2)\} \\
 & (\mathsf{dec}\, x \bullet x :\in b;\ p);\ \mathsf{end}\ x;\ \mathsf{var}\ x;\ x :\in c \\
= & \{(\mathsf{end} - \mathsf{var}\ \mathsf{skip})(3.19.7)\} \\
 & RHS
\end{array}
$$

■

The above two laws will be used in the next chapters to perform transformations on source programs. These are always block-structured, since var and end are not part of our source language. In the previous section we explained how to ensure that programs always have contiguous scope (with respect to any local variable). Therefore these laws will be applied assuming that these conditions are always satisfied.

3.20 The Correctness of the Basic Laws

In a purely algebraic view, the laws of a given language are an algebraic semantics for this language. There is no place for the task of verifying the validity of the more basic laws; they are axioms which express the relationship between the operators of the language. However, the method of postulating is questioned by those who follow a model-oriented approach, especially when the set of axioms is relatively large, as it is here. Postulating an inconsistent set of laws could be a disaster: it would allow one to prove invalid results, like the correctness of an inaccurate compiler.

The way to avoid this danger is to link the algebraic semantics of the language with a mathematical model in which the laws can be proved. For example, Hoare and He [46] provide a relational model for programs where the correctness of the laws could be established by appealing to the calculus of relations [71]. Another model that has gained widespread acceptance is the predicate transformer model of Dijskstra [21], which was briefly discussed in the previous chapter. In this case, the semantics of each language construct (that is, its *weakest precondition*) is given, and the laws are verified by appealing to the predicate calculus.

It is also possible to link the algebraic semantics of the language to a more concrete (operational) model. This allows to check for feasibility (implementability) of the language operators. But in our case this is not possible, as our language includes non-implementable operators.

Once the laws have been proved, in whatever model, they should serve as tools for carrying out program transformation. The mathematical definitions that allow their verification are normally more complex, and therefore not appealing to practical use. This separation of concerns is well described in [42], which explores the role of algebra and models to the construction of theories relevant to computing.

But even after the model has served its intended purpose, additional results of practical interest can be achieved. For example, from the experience in the application of basic algebraic laws of programming to solve a given task, one discovers more elaborate transformation strategies that allow more concise and elegant proofs. This was illustrated in the section on iteration, where all the laws were derived from more basic ones.

In the remainder of this section we use the predicate transformer model to illustrate how the basic laws of our language can be verified.

Predicate Transformers

We deal only with a few language operators. Their definitions as predicate transformers were given in the previous chapter, but are repeated here for convenience. In the following, a ranges over the set of predicates, p and q stand for arbitrary programs, and $\mathcal{P}$ for an

arbitrary set of programs.

$$\begin{array}{lcl}
\textsf{skip} & \stackrel{def}{=} & \lambda\, a \bullet a \\
\bot & \stackrel{def}{=} & \lambda\, a \bullet \textsf{false} \\
\top & \stackrel{def}{=} & \lambda\, a \bullet \textsf{true} \\
\sqcup \mathcal{P} & \stackrel{def}{=} & \lambda\, a \bullet (\exists\, X \in \mathcal{P} \bullet X(a)) \\
\sqcap \mathcal{P} & \stackrel{def}{=} & \lambda\, a \bullet (\forall\, X \in \mathcal{P} \bullet X(a)) \\
p;\ q & \stackrel{def}{=} & \lambda\, a \bullet p(q(a))
\end{array}$$

The definition of the remaining operators can be found, for example, in [5]. From the above definitions we can prove the laws of the corresponding operators. For example, from the definition of $\sqsubseteq$ we can derive its characterisation in terms of weakest preconditions:

$$\begin{array}{ll}
 & p \sqsubseteq q \\
\equiv & \{\text{Definition 3.1(The ordering Relation)}\} \\
 & (p \sqcap q) = p \\
\equiv & \{\text{Definition of } \sqcap\} \\
 & (\lambda\, a \bullet p(a) \wedge q(a)) = p \\
\equiv & \{\text{The axiom of extensionality}\} \\
 & \forall\, a \bullet (p(a) \wedge q(a) \Leftrightarrow p(a)) \\
\equiv & \{\text{Predicate calculus}\} \\
 & \forall\, a \bullet (p(a) \Rightarrow q(a))
\end{array}$$

which corresponds precisely to the definition of refinement adopted in all approaches to the refinement calculus based on weakest preconditions, as discussed in the previous chapter. As another example, we verify Law $(\sqcup - \sqcap$ dist)(3.7.2).

$$\begin{array}{ll}
 & (\sqcup \mathcal{P}) \sqcap p \\
= & \{\text{Definition of } \sqcap\} \\
 & \lambda\, a \bullet (\sqcup \mathcal{P})(a) \wedge p(a) \\
= & \{\text{Definition of } \sqcup\} \\
 & \lambda\, a \bullet (\exists\, X \in \mathcal{P} \bullet X(a)) \wedge p(a) \\
= & \{\text{Assuming that } X \text{ is not free in } p\} \\
 & \lambda\, a \bullet (\exists\, X \in \mathcal{P} \bullet (X(a) \wedge p(a))) \\
= & \{\text{Definition of } \sqcap\} \\
 & \lambda\, a \bullet (\exists\, X \in \mathcal{P} \bullet (X \sqcap p)(a)) \\
= & \{\text{Set theory}\} \\
 & \lambda\, a \bullet (\exists\, X \in \{X : X \in \mathcal{P} : (X \sqcap p)\} \bullet X(a)) \\
= & \{\text{Definition of } \sqcup\} \\
 & \sqcup\{X : X \in \mathcal{P} : (X \sqcap p)\}
\end{array}$$

Chapter 4

A Simple Compiler

> Setting up equations is like translating from one language into another.
>
> — G. Polya

In the first two sections of this chapter we describe the *normal form* as a model of an arbitrary executing mechanism. The normal form theorems of Section 4.2 are concerned with control elimination: the reduction of the nested control structure of the source program to a single flat iteration. These theorems are largely independent of a particular target machine.

In the subsequent sections, we design and prove the correctness of a compiler for a subset of our source language, not including procedures or recursion (which are dealt with in the next chapter). The constructions considered here are skip, assignment, sequential composition, demonic nondeterminism, conditional, iteration and local declarations.

As described earlier, we split the compilation process into three main phases: simplification of expressions, control elimination and data refinement (the conversion from the abstract space of the source program to the concrete state of the target machine). The control elimination phase in particular is directly achieved by instantiating the generic theorems of Section 4.2.

Every theorem directly relevant to the compilation process has the status of a *rule*. Each rule expresses a transformation which brings the source program closer to a normal form with the same structure as the target machine. The final section shows that, taken collectively, these rules can be used to carry out the compilation task.

It is important to emphasise the different roles played by the algebraic laws described in the previous chapter and these reduction rules: the laws express general properties of the language operators, whereas the rules serve the special purpose of transforming an arbitrary program to a normal form. The laws are necessary to prove the rules, and these (not the laws) are used to carry out compilation.

4.1 The Normal Form

A program of the form

$$\mathbf{dec}\, v \bullet v :\in a;\ b * p;\ c_\perp$$

can be interpreted as a very general model of a machine executing a stored program computer in the following way:

- The list of variables v represents the machine components (for example, registers). They are introduced as local variables since they have no counterpart at the source level; therefore their final values are irrelevant.
- a is an assumption about the initial state; if it is impossible to make a true by assigning to v, the machine behaves miraculously.
- p is the stored program; it is executed until the condition b becomes false. Usually, p will be a guarded command set of the form

  $$b_1 \rightarrow p_1 \,\square \ldots \square\, b_n \rightarrow p_n$$

  Whenever the machine is in state b_i the *action* (or *instruction*) p_i is executed. In this case, the condition b is given by

  $$b_1 \vee \ldots \vee b_n$$

- c is an assertion about the final state of the machine; if execution of the stored program does not assert c, the machine ends up behaving like abort.

Notice that, upon termination of the iteration $b * p$, b is false and we have

$$b * p;\ c_\perp \ =\ b * p;\ (\neg b)_\perp;\ c_\perp \ =\ b * p;\ (\neg b \wedge c)_\perp$$

Thus, there is no loss of generality in assuming that $c = (\neg b \wedge c)$, and consequently that $(b \wedge c) = \mathsf{false}$. The normal form theorems will rely on the assumption that b and c are disjoint.

A normal form program will be abbreviated as follows.

Definition 4.1 (Normal form)

$$v : [a,\ b \rightarrow p,\ c] \ \stackrel{def}{=}\ \mathbf{dec}\, v \bullet v :\in a;\ b * p;\ c_\perp, \quad \text{where } (b \wedge c) = \mathsf{false}.$$

■

For convenience, we will sometimes use the form

$$v : [a,\ (b_1 \rightarrow p_1 \,\square \ldots \square\, b_n \rightarrow p_n),\ c]$$

as an abbreviation of

$$v : [a,\ (b_1 \vee \ldots \vee b_n) \rightarrow (b_1 \rightarrow p_1 \,\square \ldots \square\, b_n \rightarrow p_n),\ c]$$

4.2 Normal Form Reduction

To reduce an arbitrary program to normal form, it is sufficient to show how each primitive command can be written in normal form, and how each operator of the language (when applied to operands in normal form) yields a result expressible in normal form. The following reductions involve no change of data representation. Therefore we can directly compare the source constructs with the associated normal form programs.

If the initial state coincides with the final state, the machine does not perform any action. In more concrete terms, the empty code is a possible implementation of **skip**.

Theorem 4.1 (Skip)

$$\textbf{skip} \sqsubseteq v : [a,\ b \to p,\ a]$$

Proof:

$$\begin{array}{ll} & RHS \\ = & \{(*\ \text{elim})(3.17.1),\ \text{remember } a \wedge b = \textbf{false}\} \\ & \textbf{dec}\ v \bullet v :\in a;\ a_\perp \\ = & \{(:\in\ \text{void } a_\perp)(3.16.4) \text{ and } (;-\textbf{skip} \text{ unit})(3.3.1)\} \\ & \textbf{dec}\ v \bullet v :\in a;\ \textbf{skip} \\ \sqsupseteq & \{(\textbf{dec}-\ :\in\ \text{initial value})(3.18.5) \text{ and } (\textbf{dec}\ \text{elim})(3.18.3)\} \\ & LHS \end{array}$$

■

The following lemma shows how a primitive command can be written in normal form. Actually, the lemma is valid for all programs p, but we will not make use of it for non-primitive constructs because we follow an innermost (*bottom-up*) reduction strategy.

Lemma 4.1 (Primitive commands)
If v is not free in p then

$$p \sqsubseteq v : [a, a \to (p;\ v :\in c), c]$$

Proof:

$$\begin{array}{ll} & RHS \\ = & \{(*\ \text{unfold})(3.17.2) \text{ and } (*\ \text{elim})(3.17.1)\} \\ & \textbf{dec}\ v \bullet v :\in a;\ p;\ v :\in c;\ c_\perp \\ \sqsupseteq & \{(\textbf{dec}-\ :\in\ \text{initial value})(3.18.5) \text{ and } (:\in\ \text{void } c_\perp)(3.16.4)\} \\ & \textbf{dec}\ v \bullet p;\ v :\in c \\ \sqsupseteq & \{(\textbf{dec}-\ :\in\ \text{final value})(3.18.6) \text{ and } (\textbf{dec}\ \text{elim})(3.18.3)\} \\ & LHS \end{array}$$

■

The following normal form representations of skip and assignment are instantiations of the above lemma. The one of skip is further simplified by the fact that it is an identity of sequential composition. The operational interpretation is that skip can be implemented by a jump.

Theorem 4.2 (Skip)

$$\textsf{skip} \sqsubseteq v : [a, (a \to v :\in c), c]$$

■

Theorem 4.3 (Assignment)

$$x := e \sqsubseteq v : [a, a \to (x := e;\; v :\in c), c]$$

■

The reduction of sequential composition assumes that both arguments are already in normal form, and that the final state of the left argument coincides with the initial state of the right argument. The components of the resulting normal form are the initial state of the left argument, the final state of the right argument and a guarded command set that combines the original guarded commands.

First we prove the reduction of sequential composition for the particular case where the guarded command set of the right argument includes that of the left argument.

Lemma 4.2 (Sequential composition)

$$v : [a, b_1 \to p, c_0];\; v : [c_0, \begin{pmatrix} & b_1 \to p \\ \Box & b_2 \to q \end{pmatrix}, c] \sqsubseteq v : [a, \begin{pmatrix} & b_1 \to p \\ \Box & b_2 \to q \end{pmatrix}, c]$$

Proof:
Let $R = (b_1 \to p \,\Box\, b_2 \to q)$.

$$
\begin{array}{ll}
 & LHS \\
\sqsubseteq & \{(\textbf{dec}-;\ \textrm{dist})(3.18.8)\} \\
 & \textbf{dec}\ v \bullet v :\in a;\ b_1 * p;\ c_{0\perp};\ v :\in c_0;\ (b_1 \vee b_2) * R;\ c_\perp \\
\sqsubseteq & \{(v :\in c_0 \textrm{ refined by } {c_0}^\top)(3.19.6) \textrm{ and } (c_{0\perp} - {c_0}^\top \textrm{ simulation})(3.11.10)\} \\
 & \textbf{dec}\ v \bullet v :\in a;\ b_1 * p;\ (b_1 \vee b_2) * R;\ c_\perp \\
= & \{(* - \Box \textrm{ elim})(3.17.4)\} \\
 & \textbf{dec}\ v \bullet v :\in a;\ b_1 * R;\ (b_1 \vee b_2) * R;\ c_\perp \\
= & \{(* \textrm{ sequence})(3.17.7)\} \\
 & RHS
\end{array}
$$

■

Now we show that the guarded command set of a normal form program can be reduced by eliminating arbitrary guarded commands. We obtain a program which is worse than the original one.

Lemma 4.3 (Eliminate guarded command)

$$v : [a, \begin{pmatrix} & b_1 \to p \\ \Box & b_2 \to q \end{pmatrix}, c] \sqsupseteq v : [a, b_1 \to p, c]$$

Proof:
Let $R = (b_1 \to p \;\Box\; b_2 \to q)$

$$\begin{array}{ll} & LHS \\ \sqsupseteq & \{\text{Lemma 4.2(Sequential composition)}\} \\ & v : [a, b_1 \to p, c];\; v : [c, R, c] \\ \sqsupseteq & \{\text{Theorem 4.1(Skip) and } \langle ;-\mathbf{skip}\text{ unit}\rangle(3.3.1)\} \\ & RHS \end{array}$$

■

The reduction of sequential composition is proved directly from the above two lemmas. These lemmas will be of more general utility.

Theorem 4.4 (Sequential composition)

$$v : [a, b_1 \to p, c_0];\; v : [c_0, b_2 \to q, c] \sqsubseteq v : [a, \begin{pmatrix} & b_1 \to p \\ \Box & b_2 \to q \end{pmatrix}, c]$$

Proof:

$$\begin{array}{ll} & RHS \\ \sqsupseteq & \{\text{Lemma 4.2(Sequential composition)}\} \\ & v : [a, b_1 \to p, c_0];\; v : [c_0, \begin{pmatrix} & b_1 \to p \\ \Box & b_2 \to q \end{pmatrix}, c] \\ \sqsupseteq & \{\text{Lemma 4.3(Eliminate guarded command)}\} \\ & LHS \end{array}$$

■

The following lemma shows how to eliminate a conditional command when its branches are normal form programs with identical components, except for the initial state. The first action to be executed in the resulting normal form program determines which of the original initial states should be activated.

Lemma 4.4 (Conditional)
If v is not free in b then

$$v : [a_1,\ R,\ c] \lhd b \rhd v : [a_2,\ R,\ c] \ \sqsubseteq\ v : [a,\ R,\ c]$$

where $R = \begin{pmatrix} & a \rightarrow (v :\in a_1 \lhd b \rhd v :\in a_2) \\ \Box & b_1 \rightarrow p \end{pmatrix}$

Proof:

$$\begin{array}{ll} & RHS \\ = & \{(* - \Box\ \text{unfold})(3.17.3)\} \\ & \mathbf{dec}\, v \bullet v :\in a;\ (v :\in a_1 \lhd b \rhd v :\in a_2);\ (a \vee b_1) * R;\ c_\perp \\ = & \{(; - \lhd\ \rhd\ \text{left dist})(3.14.6)\} \\ & \mathbf{dec}\, v \bullet v :\in a;\ ((v :\in a_1;\ (a \vee b_1) * R;\ c_\perp) \lhd b \rhd (v :\in a_2;\ (a \vee b_1) * R;\ c_\perp)) \\ = & \{(:\in - \lhd\ \rhd\ \text{right dist})(3.16.6)\ \text{and}\ (\mathbf{dec} - \lhd\ \rhd\ \text{dist})(3.18.10)\} \\ & (\mathbf{dec}\, v \bullet v :\in a;\ v :\in a_1;\ (a \vee b_1) * R;\ c_\perp) \lhd b \rhd \\ & (\mathbf{dec}\, v \bullet v :\in a;\ v :\in a_2;\ (a \vee b_1) * R;\ c_\perp) \\ \sqsupseteq & \{(\mathbf{dec} - :\in\ \text{initial value})(3.18.5)\} \\ & LHS \end{array}$$

■

The above lemma is useful for intermediate calculations. It is used in the proof of the normal form reduction of conditional and iteration commands.

Theorem 4.5 (Conditional)
If v does not occur in b then

$$v : [a_1,\ b_1 \rightarrow p,\ c_1] \lhd b \rhd v : [a_2,\ b_2 \rightarrow q,\ c] \ \sqsubseteq\ v : [a,\ R,\ c]$$

where $R = \begin{pmatrix} & a \rightarrow (v :\in a_1 \lhd b \rhd v :\in a_2) \\ \Box & b_1 \rightarrow p \ \Box\ c_1 \rightarrow v :\in c \\ \Box & b_2 \rightarrow q \end{pmatrix}$

Proof:

$$\begin{array}{ll} & RHS \\ \sqsupseteq & \{\text{Lemma 4.4(Conditional)}\} \\ & v : [a_1,\ R,\ c] \lhd b \rhd v : [a_2,\ R,\ c] \\ \sqsupseteq & \{\text{Lemmas 4.2(Sequential composition) and 4.3(Eliminate guarded command)}\} \\ & (v : [a_1,\ b_1 \rightarrow p,\ c_1];\ v : [c_1,\ c_1 \rightarrow v :\in c,\ c]) \lhd b \rhd v : [a_2,\ b_2 \rightarrow q,\ c] \\ \sqsupseteq & \{\text{Theorem 4.2(Skip) and } (; -\mathbf{skip}\ \text{unit})(3.3.1)\} \\ & LHS \end{array}$$

■

The next lemma establishes a simple fact: if the unique effect of the first guarded command to be executed is to make a certain expression true, we may substitute the expression for the initial state of the normal form program.

Lemma 4.5 (Void initial state)

$$v : [c_0, \begin{pmatrix} & c_0 \rightarrow v :\in a \\ \square & b \rightarrow p \end{pmatrix}, c] \sqsupseteq v : [a, \begin{pmatrix} & c_0 \rightarrow v :\in a \\ \square & b \rightarrow p \end{pmatrix}, c]$$

Proof:

$$\begin{array}{ll} & LHS \\ = & \{(* - \square \text{ unfold})(3.17.3)\} \\ & \mathbf{dec}\, v \bullet v :\in c_0;\ v :\in a;\ (c_0 \vee b) * \begin{pmatrix} & c_0 \rightarrow v :\in a \\ \square & b \rightarrow p \end{pmatrix};\ c_\perp \\ \sqsupseteq & \{(\mathbf{dec} - :\in \text{ initial value})(3.18.5)\} \\ & RHS \end{array}$$

■

In order to reduce an iteration command, we assume that its body is in normal form. Let a_0 and c_0 be the initial and final states of this normal form program. The normal form of the whole iteration behaves as follows. The first action to be executed is a conditional command which tests if the condition of the iteration holds, in which case a_0 is activated; otherwise, the program reaches its final state. When c_0 is activated, the guard of the first action is activated so that the conditional command is executed again.

Theorem 4.6 (Iteration)
If v does not occur in b then

$$b * v : [a_0,\ b_1 \rightarrow p,\ c_0] \sqsubseteq v : [a,\ R,\ c]$$

where $R = \begin{pmatrix} & a \rightarrow (v :\in a_0 \lhd b \rhd v :\in c) \\ \square & c_0 \rightarrow v :\in a \\ \square & b_1 \rightarrow p \end{pmatrix}$

Proof:

$$\begin{array}{ll} & RHS \\ \sqsupseteq & \{\text{Lemma 4.4(Conditional)}\} \\ & v : [a_0,\ R,\ c] \lhd b \rhd v : [c,\ R,\ c] \\ \sqsupseteq & \{\text{Lemma 4.2(Sequential composition) and Theorem 4.1(Skip)}\} \\ & v : [a_0,\ b_1 \rightarrow p,\ c_0];\ v : [c_0,\ R,\ c] \lhd b \rhd \mathbf{skip} \\ \sqsupseteq & \{\text{Lemma 4.5(Void initial state)}\} \\ & (v : [a_0,\ b_1 \rightarrow p,\ c_0];\ RHS) \lhd b \rhd \mathbf{skip} \end{array}$$

The final result follows from the above and (μ least fixed point)(3.8.2). ■

The nondeterministic choice of two programs can be implemented by either of them. We can actually eliminate the choice at the source level, and avoid compiling one of the components.

Theorem 4.7 (Nondeterminism)

$$(1)\ (p \sqcap q) \sqsubseteq p$$
$$(2)\ (p \sqcap q) \sqsubseteq q$$

Proof: From $(\sqsubseteq - \sqcap \text{ glb})(3.6.7)$. ■

4.3 The Target Machine

The compiler we design in the next three sections produces code for a simple target machine which consists of four components:

P	a sequential register (program counter)
A	a general purpose register
M	a store for variables (RAM)
m	a store for instructions (ROM)

The idea is to regard the machine components as program variables and design the instructions as assignments that update the machine state.

P and A will be represented by single variables. Although we do not deal with types explicitly, P will be assigned integer expressions, standing for locations in ROM. A will be treated as an ordinary source variable; it will play an important role in the decomposition of expressions, which is the subject of the next section. M will be modelled as a map from addresses of locations in RAM to expressions denoting the corresponding values, and m as a map from addresses of locations in ROM to instructions.

In order to model M and m, we need to extend our language to allow map variables. We use the following operators on maps:

$\{x \mapsto e\}$	Singleton map
$m_1 \cup m_2$	union
$m_1 \oplus m_2$	overriding
$m[x]$	application

Furthermore, we use the following abbreviations:

$$\{x_1, \ldots, x_n \mapsto e_1, \ldots, e_n\} \stackrel{def}{=} \{x_1 \mapsto e_1\} \cup \ldots \cup \{x_n \mapsto e_n\}$$
$$m[x_1, \ldots, x_n] \stackrel{def}{=} m[x_1], \ldots, m[x_n]$$

In the first case we assume that no variable appears more than once in the list $x_1, \ldots, x_n$.

One of the greatest advantages of algebra is abstraction. All the laws of our language are unaffected by this extension. In particular, the laws of assignment and declaration can be

readily used to manipulate map variables and expressions, and no additional laws turn out to be necessary. For example, we can combine the two assignments

$$m := m \oplus \{x \mapsto e\};\ m := m \oplus \{y \mapsto f\}$$

by using Law $\langle :=$ combination$\rangle$(3.15.4), resulting in

$$m := m \oplus \{x \mapsto e\} \oplus \{y \mapsto f\}$$

Similarly,

$$m := m \oplus \{x \mapsto e\};\ \mathsf{end}\, m$$

is equivalent to (from Law $\langle$end$- :=$ final value$\rangle$(3.19.5))

$$\mathsf{end}\, m$$

The instructions of our simple machine are defined below. We assume that n stands for an address in RAM and k for an address in ROM.

$$\begin{array}{lcl}
\mathsf{load}(n) & \stackrel{def}{=} & \mathsf{A}, \mathsf{P} := \mathsf{M}[n], \mathsf{P}+1, \\
\mathsf{store}(n) & \stackrel{def}{=} & \mathsf{M}, \mathsf{P} := (\mathsf{M} \oplus \{n \mapsto \mathsf{A}\}), \mathsf{P}+1 \\
\mathsf{bop\text{—}A}(n) & \stackrel{def}{=} & \mathsf{A}, \mathsf{P} := (\mathsf{A}\, \mathsf{bop}\, \mathsf{M}[n]), \mathsf{P}+1 \\
\mathsf{uop\text{—}A} & \stackrel{def}{=} & \mathsf{A}, \mathsf{P} := (\mathsf{uop}\, \mathsf{A}), \mathsf{P}+1 \\
\mathsf{jump}(k) & \stackrel{def}{=} & \mathsf{P} := k \\
\mathsf{cjump}(k) & \stackrel{def}{=} & \mathsf{P} := (\mathsf{P}+1 \lhd \mathsf{A} \rhd k)
\end{array}$$

where

$$x := (e_1 \lhd b \rhd e_2) \stackrel{def}{=} (x := e_1 \lhd b \rhd x := e_2)$$

and, as before, bop and uop stand for arbitrary binary and unary operators, respectively.

The normal form describing the behaviour of this machine is an iterated execution of instructions taken from the store m at location P:

$$\mathsf{dec}\, \mathsf{P}, \mathsf{A} \bullet \mathsf{P} := s;\ (s \leq \mathsf{P} < f) * \mathsf{m}[\mathsf{P}];\ (\mathsf{P} = f)_{\perp}$$

where s is the intended start address of code and f the finish address. The aim of the following sections is to show how an arbitrary source program can be reduced to this form.

4.4 Simplification of Expressions

One of the tasks involved in the translation process is the elimination of nested expressions. The outcome of this phase is a program where each assignment is *simple* (see definition below). Furthermore, all the local variables are expanded to widest scope, so that they can be implemented as global variables. Of course, this is valid only in the absence of recursion, and it requires that all local variables have distinct names, which must also be different from the names used for global variables.

Definition 4.2 (Simple assignment)
An assignment is simple if it has one of the forms

$$\mathsf{A} := x$$
$$x := \mathsf{A}$$
$$\mathsf{A} := \mathsf{A}\ \mathsf{bop}\ x$$
$$\mathsf{A} := \mathsf{uop}\ \mathsf{A}$$

where x is a source variable. ■

These patterns are closely related to the ones used to define the instructions. For example, the first one will eventually turn into a load instruction: the variable P will appear as a result of the control elimination phase, and x will be replaced by its memory location as a result of the change of data representation.

In the remainder of this section we will show how to simplify expressions by using the register variable A and new local variables. We assume that x is a single variable and e a single expression, rather than arbitrary lists.

The first rule transforms an assignment into a block that introduces A as a local variable.

Rule 4.1 (Introduce A)
If A does not occur in $x := e$

$$(x := e) \;=\; \mathsf{dec}\,\mathsf{A} \bullet \mathsf{A} := e;\; x := \mathsf{A}$$

Proof: From the laws to combine assignment and eliminate local variables. ■

Note that the assignment $x := \mathsf{A}$ is already simple. By transforming all the assignments in this way, we need only simplify expressions assigned to A. The next rule deals with unary operators.

Rule 4.2 (Unary operator)

$$(\mathsf{A} := \mathsf{uop}\, e) \;=\; (\mathsf{A} := e;\; \mathsf{A} := \mathsf{uop}\, A)$$

Proof: From the law to combine assignments. ■

Observe that the second assignment on the right-hand side of the above equation is simple. To deal with binary operators we need to introduce a fresh local variable t. It plays the role of a temporary variable that holds the value of a subexpression.

Rule 4.3 (Binary operator)
If neither A nor t occur in e or f

$$(\mathsf{A} := e\,\mathsf{bop}\,f) \;=\; \mathsf{dec}\,t \bullet \mathsf{A} := f;\; t := \mathsf{A};\; \mathsf{A} := e;\; \mathsf{A} := \mathsf{A}\,\mathsf{bop}\,t$$

Proof:

$$
\begin{array}{ll}
 & RHS \\
= & \{(:=\ \text{combination})(3.15.4) \text{ and } (:=\ \text{identity})(3.15.2)\} \\
 & \mathbf{dec}\, t \bullet \mathsf{A} := f;\ t := f;\ \mathsf{A} := e\ \mathbf{bop}\ t \\
= & \{(:=\ \text{combination})(3.15.4) \text{ and } (:=\ \text{identity})(3.15.2)\} \\
 & \mathbf{dec}\, t \bullet \mathsf{A} := f;\ \mathsf{A} := e\ \mathbf{bop}\, f;\ t := f \\
= & \{(:=\ \text{combination})(3.15.4),\ (\mathbf{dec}-:=\ \text{final value})(3.18.6) \text{ and} \\
 & \ (\mathbf{dec}\ \text{elim})(3.18.3)\} \\
 & LHS
\end{array}
$$

■

Again, the only expressions that may still need to be simplified (e and f) are both assigned to the variable A. An exhaustive application of the above rules will simplify arbitrarily nested expressions, turning every assignment into a simple one.

When the expression f above is a variable, it is unnecessary to create a temporary variable to hold its value. The following is an optimisation of the previous rule for this particular case.

Rule 4.4 (Binary operator — optimisation)
If A does not occur in e or x

$$(\mathsf{A} := e\ \mathbf{bop}\ x) \ =\ (\mathsf{A} := e;\ \mathsf{A} := \mathsf{A}\ \mathbf{bop}\ x)$$

Proof: From the law to combine assignments. ■

The boolean expressions appearing in iteration and conditional commands may also be arbitrarily nested, and therefore need to be simplified.

Rule 4.5 (Condition of iteration)
If neither A nor v occur in b

$$b * (\mathbf{dec}\, v, \mathsf{A} \bullet p) \ \sqsubseteq\ \mathbf{dec}\, v, \mathsf{A} \bullet \mathsf{A} := b;\ \mathsf{A} * (p;\ \mathsf{A} := b)$$

Proof:

$$
\begin{array}{ll}
 & RHS \\
= & \{(\mu\ \text{fixed point})(3.8.1)\} \\
 & \mathbf{dec}\, v, \mathsf{A} \bullet \mathsf{A} := b;\ ((p;\ \mathsf{A} := b;\ \mathsf{A} * (p;\ \mathsf{A} := b)) \lhd b \rhd \mathbf{skip}) \\
= & \{(:= - \lhd \rhd \text{ right dist})(3.15.7),\ (\mathbf{dec} - \lhd \rhd \text{ dist})(3.18.10) \text{ and} \\
 & \ (\mathbf{dec}\ \text{elim})(3.18.3)\} \\
 & (\mathbf{dec}\, v, \mathsf{A} \bullet \mathsf{A} := b;\ p;\ \mathsf{A} := b;\ \mathsf{A} * (p;\ \mathsf{A} := b)) \lhd b \rhd \mathbf{skip} \\
\sqsupseteq & \{(\mathbf{dec}-;\ \text{dist})(3.18.8) \text{ and } (\mathbf{dec}-:=\ \text{initial value})(3.18.5)\} \\
 & ((\mathbf{dec}\, v, \mathsf{A} \bullet p);\ RHS) \lhd b \rhd \mathbf{skip}
\end{array}
$$

The result follows from (μ least fixed point)(3.8.2). ■

The local variable A on the left-band side of the above inequation is a result of the simplification of assignments in p. Likewise, v may be an arbitrary list of temporary variables created to simplify boolean operators or originally introduced by the programmer. By moving these declarations out of the body of the iteration, we avoid nested declarations of the variable A. The expression b can now be simplified using the previous theorems.

In a similar way, we can simplify the boolean expressions of conditional statements.

Rule 4.6 (Condition of conditional)
If neither v nor A occur in b

$$(\mathsf{dec}\, v, \mathsf{A} \bullet p) \lhd b \rhd (\mathsf{dec}\, v, \mathsf{A} \bullet q) \sqsubseteq \mathsf{dec}\, v, \mathsf{A} \bullet \mathsf{A} := b;\ (p \lhd \mathsf{A} \rhd q)$$

Proof: Similar to the one above. ∎

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

Theorem 4.8 (Expression simplification)
For an arbitrary source program p, there is a program q such that

$$p \sqsubseteq \mathsf{dec}\, v, \mathsf{A} \bullet q$$

where q contains no local declarations, all assignments in q are simple and the only boolean condition in q is the variable A.
Proof: By structural induction using rules 4.1–4.6, together with the following laws:

- (;–dec dist)(3.18.7, 3.18.8)
  This is used to increase the scope of the local variables as much as possible.

- (dec assoc)(3.18.1) and (dec rename)(3.18.4)
  The former is used to eliminate nested declarations that may have been introduced by the programmer or resulted from the simplification of boolean operators; the latter is used to rename nested occurrences of variables which were declared with the same name, so that the nesting can be eliminated.

- (dec elim)(3.18.3)
  Rule 4.6 assumes that the local variables of the two branches of a conditional are the same. The above law can be used to introduce void declarations to ensure that this assumption will be satisfied.

∎

4.5 Control Elimination

Recall that the machine considered here is equipped with a register P which is used for scheduling the selection and sequencing of instructions. This can be simulated by regarding P as a variable in the following way:

- Selection is achieved by representing the stored program as a set of guarded commands, each one of the form

  $$(\mathsf{P} = k) \rightarrow q$$

  meaning that q (standing for some machine instruction) will be executed when P has value k.

- Sequencing is modelled by incrementing P

  $$\mathsf{P} := \mathsf{P} + 1$$

- A jump to an instruction at memory location k is achieved by the assignment

  $$\mathsf{P} := k$$

Clearly, the initial value of P must be the address of the location of the first instruction to be executed. These conventions are our basis to transform the nested control structure of the source program into a single flat iteration which models the execution of a stored program. The outcome of this section is a *simple* normal form program.

Definition 4.3 (Simple normal form)
We say that a normal form program is *simple* if it has the form

$$\mathsf{P} : [(\mathsf{P} = s), b \rightarrow p, (\mathsf{P} = f)]$$

where p is a set of guarded commands of the form

$$\square_{s \le k < f} (\mathsf{P} = k) \rightarrow x_k, \mathsf{P} := e_k, d_k$$

and b is the union of the guards $(\mathsf{P} = k)$, for all k such that $s \le k < f$. Furthermore, the assignment $x_k, \mathsf{P} := e_k, d_k$ follows one of the patterns used to define the machine instructions, except that the source variables may not yet have been replaced by their corresponding memory locations (this is addressed in the next section). ■

Reduction to this simple normal form can be achieved by instantiating the normal form theorems of Section 4.2, taking into account the particular encoding of control state of our simple machine. In the following we abbreviate

$$\mathsf{P} : [(\mathsf{P} = s), b \rightarrow p, (\mathsf{P} = f)]$$

to

$$\mathsf{P} : [s, b \rightarrow p, f]$$

The purpose of the first implementation of **skip** is to generate empty code: **false** $\rightarrow$ **skip** is equivalent to $\top$ which is the identity of $\square$.

Rule 4.7 (Skip)

$$\text{skip} \sqsubseteq \mathsf{P} : [s, \text{false} \rightarrow \text{skip}, s]$$

■

Rule 4.8 (Skip)

$$\text{skip} \sqsubseteq \mathsf{P} : [s, (\mathsf{P} = s \rightarrow \mathsf{P} := s+1), s+1]$$

■

Rule 4.9 (Assignment)

$$(x := e) \sqsubseteq \mathsf{P} : [s, (\mathsf{P} = s \rightarrow (x, \mathsf{P} := e, \mathsf{P}+1)), s+1]$$

■

Rule 4.10 (Sequential composition)

$$(\mathsf{P} : [s, b_1 \rightarrow p, f_0]);\ (\mathsf{P} : [f_0, b_2 \rightarrow q, f]) \sqsubseteq \mathsf{P} : [s, \begin{pmatrix} & b_1 \rightarrow p \\ \Box & b_2 \rightarrow q \end{pmatrix}, f]$$

■

Rule 4.11 (Conditional)

$$(\mathsf{P} : [s+1, b_1 \rightarrow p, f_0]) \lhd \mathsf{A} \rhd (\mathsf{P} : [f_0+1, b_2 \rightarrow q, f]) \sqsubseteq \mathsf{P} : [s, R, f]$$

where $R = \begin{pmatrix} & \mathsf{P} = s \rightarrow \mathsf{P} := (\mathsf{P}+1 \lhd \mathsf{A} \rhd f_0+1) \\ \Box & b_1 \rightarrow p \ \Box\ \mathsf{P} = f_0 \rightarrow \mathsf{P} := f \\ \Box & b_2 \rightarrow q \end{pmatrix}$

■

Rule 4.12 (Iteration)

$$\mathsf{A} * (\mathsf{P} : [s+1, b \rightarrow p, f_0]) \sqsubseteq \mathsf{P} : [s, \begin{pmatrix} \mathsf{P} = s \rightarrow \mathsf{P} := (\mathsf{P}+1 \lhd \mathsf{A} \rhd f_0+1) \\ \Box\ b \rightarrow p\ \Box\ \mathsf{P} = f_0 \rightarrow \mathsf{P} := s \end{pmatrix}, f_0+1]$$

■

It is worth observing that the above rules assume the allocation of contiguous addresses for the stored program. For example, the rule for sequential composition assumes that the finish address of the normal form program on the left coincides with the start address of the normal form program on the right.

Strictly, the above rules cannot be justified only from the reduction theorems of Section 4.2. Some additional (although trivial) transformations are required. As an example, we present the proof of Rule 4.9.

$$\begin{array}{ll} & \mathsf{P} : [s, (\mathsf{P} = s \rightarrow (x, \mathsf{P} := e, \mathsf{P}+1)), s+1] \\ = & \{(:= \text{ substitution})(3.15.5)\} \\ & \mathsf{P} : [s, (\mathsf{P} = s \rightarrow (x, \mathsf{P} := e, s+1)), s+1] \\ = & \{(:= \text{ combination})(3.15.4) \text{ and } (:= \text{ identity})(3.15.2)\} \\ & \mathsf{P} : [s, (\mathsf{P} = s \rightarrow (x := e;\ \mathsf{P} := s+1)), s+1] \\ = & \{(:\in - := \text{ conversion})(3.16.7)\} \\ & \mathsf{P} : [s, (\mathsf{P} = s \rightarrow (x := e;\ \mathsf{P} :\in (\mathsf{P} = s+1))), s+1] \\ \sqsupseteq & \{\text{Theorem 4.3(Assignment)}\} \\ & x := e \end{array}$$

The additional transformations required to prove the other theorems are similar. The next theorem summarises the outcome of this phase of compilation.

Theorem 4.9 (Control elimination)
Consider a program of the form

$$\mathsf{dec}\, v, \mathsf{A} \bullet q$$

where q contains no local declaration, all assignments in q are simple and the only boolean condition in q is the variable A. Then there is a *simple* normal form program such that

$$\mathsf{dec}\, v, \mathsf{A} \bullet q \ \sqsubseteq\ v, \mathsf{A}, \mathsf{P} : [(\mathsf{P} = s), b \rightarrow r, (\mathsf{P} = f)]$$

Proof: By structural induction using rules 4.7–4.12, we transform q into

$$\mathsf{P} : [(\mathsf{P} = s), b \rightarrow r, (\mathsf{P} = f)]$$

The final result follows from (dec assoc)(3.18.1). ■

4.6 Data Refinement

The only task that remains to be addressed is the replacement of the abstract space of the source program (formed from the source variables) by the concrete state of the target machine, represented by the store M. As mentioned in Section 3.10, the idea is to define a simulation function and use its distributivity properties to perform this data refinement in a systematic way.

Suppose that Ψ is a symbol table which maps each global variable of the source program to the address of the storage M allocated to hold its value, so $\mathsf{M}[\Psi x]$[1] is the location holding the value of x. Clearly it is necessary to insist that Ψ is a total injection. Assuming that w is a list formed from the global variables (where each variable in the domain of Ψ occurs exactly once in w) we define the following *encoding* program.

[1]To improve readability we abbreviate the function application $\Psi[x]$ to Ψx.

Definition 4.4 (Encoding Program)

$$\hat{\Psi}_w \stackrel{def}{=} \mathsf{var}\, w;\ w := \mathsf{M}[\Psi w];\ \mathsf{end}\, \mathsf{M}$$

■

which *retrieves* the abstract state from the concrete state by assigning to each source variable the value in the corresponding location. (Recall that we allow list application: for $w = x, \ldots, z$ the above assignment is equivalent to $x, \ldots, z := \mathsf{M}[\Psi x], \ldots, \mathsf{M}[\Psi z]$.)

The following *decoding* program maps the abstract state to the concrete machine state.

Definition 4.5 (Decoding program)

$$\hat{\Psi}_w^{-1} \stackrel{def}{=} \mathsf{var}\, \mathsf{M};\ \mathsf{M} := \mathsf{M} \oplus \{\Psi w \mapsto w\};\ \mathsf{end}\, w$$

■

Also recall that for $w = x, \ldots, z$ the above assignment corresponds to

$$\mathsf{M} := \mathsf{M} \oplus (\{\Psi x \mapsto x\} \cup \ldots \cup \{\Psi z \mapsto z\})$$

which updates the memory M at position Ψx with the value currently held by the variable x, and so on. The first theorem formalises the obvious relationship between $\hat{\Psi}_w$ and $\hat{\Psi}_w^{-1}$.

Theorem 4.10 ($(\hat{\Psi}_w, \hat{\Psi}_w^{-1})$ simulation)
The pair of programs $(\hat{\Psi}_w, \hat{\Psi}_w^{-1})$ is a simulation.
Proof:

$$\begin{array}{ll}
& \hat{\Psi}_w;\ \hat{\Psi}_w^{-1} \\
\sqsubseteq & \{\text{Definitions of } \hat{\Psi}_w, \hat{\Psi}_w^{-1} \text{ and } \langle \mathsf{end} - \mathsf{var} \text{ simulation}\rangle(3.19.6)\} \\
& \mathsf{var}\, w;\ w := \mathsf{M}[\Psi w];\ \mathsf{M} := \mathsf{M} \oplus \{\Psi w \mapsto w\};\ \mathsf{end}\, w \\
= & \{\langle := \text{ combination}\rangle(3.15.4) \text{ and } \langle := \text{ identity}\rangle(3.15.2)\} \\
& \mathsf{var}\, w;\ w, \mathsf{M} := \mathsf{M}[\Psi w], (\mathsf{M} \oplus \{\Psi w \mapsto \mathsf{M}[\Psi w]\});\ \mathsf{end}\, w \\
= & \{\text{Property of maps } (\mathsf{M} = \mathsf{M} \oplus \{\Psi w \mapsto \mathsf{M}[\Psi w]\}) \text{ and } \langle := \text{ identity}\rangle(3.15.2)\} \\
& \mathsf{var}\, w;\ w := \mathsf{M}[\Psi w];\ \mathsf{end}\, w \\
= & \{\langle \mathsf{end} - \mathsf{var} \text{ simulation}\rangle(3.19.6) \text{ and } \langle \mathsf{end} - := \text{ final value}(3.19.5)\rangle\} \\
& \mathsf{skip} \\
= & \{\langle \mathsf{end} - \mathsf{var} \text{ simulation}\rangle(3.19.6) \text{ and } \langle \mathsf{end} - := \text{ final value}(3.19.5)\rangle\} \\
& \mathsf{var}\, \mathsf{M};\ \mathsf{M} := \mathsf{M} \oplus \{\Psi w \mapsto w\};\ \mathsf{end}\, \mathsf{M} \\
= & \{\langle := \text{ combination}\rangle(3.15.4) \text{ and } \langle := \text{ identity}\rangle(3.15.2)\} \\
& \mathsf{var}\, \mathsf{M};\ \mathsf{M} := \mathsf{M} \oplus \{\Psi w \mapsto w\};\ w := \mathsf{M}[\Psi w];\ \mathsf{end}\, \mathsf{M} \\
= & \{\langle \mathsf{end} - \mathsf{var} \text{ skip}\rangle(3.19.7)\} \\
& (\mathsf{var}\, \mathsf{M};\ \mathsf{M} := \mathsf{M} \oplus \{\Psi w \mapsto w\};\ \mathsf{end}\, w);\ (\mathsf{var}\, w;\ w := \mathsf{M}[\Psi w];\ \mathsf{end}\, \mathsf{M}) \\
= & \{\text{Definitions of } \hat{\Psi}_w, \hat{\Psi}_w^{-1}\} \\
& \hat{\Psi}_w^{-1};\ \hat{\Psi}_w
\end{array}$$

■

Recall from Section 3.10 that we use the first component of a simulation as a function. For example, for a program p we have

$$\hat{\Psi}_w(p) \;=\; \hat{\Psi}_w;\; p;\; \hat{\Psi}_w^{-1}$$

Here we generalise this particular simulation function to take an expression as argument. The effect of applying $\hat{\Psi}_w$ to e is to replace free occurrences of w in e with the corresponding machine locations $\mathsf{M}[\Psi w]$.

Definition 4.6 (Simulation as substitution)

$$\hat{\Psi}_w(e) \;\stackrel{def}{=}\; e[w \leftarrow \mathsf{M}[\Psi w]]$$

■

In order to carry out the change of data representation in a systematic way, we need to prove the following distributivity properties.

Rule 4.13 (Piecewise data refinement)
(1) $\hat{\Psi}_w(\mathsf{skip}) \sqsubseteq \mathsf{skip}$
(2) $\hat{\Psi}_{x,w}(x := e) \sqsubseteq \mathsf{M} := \mathsf{M} \oplus \{\Psi x \mapsto \hat{\Psi}_{x,w}(e)\}$
(3) $\hat{\Psi}_w(x := e) \sqsubseteq x := \hat{\Psi}_w(e)$ if x does not occur in w
(4) $\hat{\Psi}_w(p;\, q) \sqsubseteq \hat{\Psi}_w(p);\, \hat{\Psi}_w(q)$
(5) $\hat{\Psi}_w(p \lhd b \rhd q) \sqsubseteq \hat{\Psi}_w(p) \lhd \hat{\Psi}_w(b) \rhd \hat{\Psi}_w(q)$
(6) $\hat{\Psi}_w(b * p) \sqsubseteq \hat{\Psi}_w(b) * \hat{\Psi}_w(p)$
(7) $\hat{\Psi}_w(b_\perp) \sqsubseteq (\hat{\Psi}_w(b))_\perp$
(8) $\hat{\Psi}_w(b \rightarrow p) \sqsubseteq \hat{\Psi}_w(b) \rightarrow \hat{\Psi}_w(p)$
(9) $\hat{\Psi}_w(p \sqcap q) \sqsubseteq \hat{\Psi}_w(p) \sqcap \hat{\Psi}_w(q)$

Proof: (1), (4) and (9) follow directly from the fact that $\hat{\Psi}_w$ is a simulation function (see Theorem 3.4). Below we verify the others.

$$
\begin{array}{lll}
(2) & & \hat{\Psi}_{x,w}(x := e) \\
& = & \{\text{Definition of } \hat{\Psi}_{x,w} \text{ and (end change scope)(3.19.2)}\} \\
& & \mathsf{var}\, x, w;\; x, w := \mathsf{M}[\Psi x, w];\; x := e;\; \mathsf{end}\, \mathsf{M};\; \hat{\Psi}_{x,w}^{-1} \\
& = & \{(:=\ \text{combination})(3.15.4),\ (:=\ \text{identity})(3.15.2) \text{ and} \\
& & \text{Definition 4.6(Simulation as substitution)}\} \\
& & \mathsf{var}\, x, w;\; x, w := \hat{\Psi}_{x,w}(e), \mathsf{M}[\Psi w];\; \mathsf{end}\, \mathsf{M};\; \hat{\Psi}_{x,w}^{-1} \\
& \sqsubseteq & \{\text{Definition of } \hat{\Psi}_{x,w}^{-1} \text{ and (end} - \text{var simulation)(3.19.6)}\} \\
& & \mathsf{var}\, x, w;\; x, w := \hat{\Psi}_{x,w}(e), \mathsf{M}[\Psi w];\; \mathsf{M} := \mathsf{M} \oplus \{\Psi x, w \mapsto x, w\};\; \mathsf{end}\, x, w \\
& = & \{(:=\ \text{combination})(3.15.4),\ (:=\ \text{identity})(3.15.2) \text{ and} \\
& & \text{(end change scope)(3.19.2)}\} \\
& & \mathsf{var}\, x, w;\; x, w := \hat{\Psi}_{x,w}(e), \mathsf{M}[\Psi w];\; \mathsf{end}\, x, w;\; \mathsf{M} := \mathsf{M} \oplus \{\Psi x \mapsto \hat{\Psi}_{x,w}(e)\} \\
& = & \{(\text{end} - :=\ \text{final value})(3.19.5) \text{ and (end} - \text{var simulation)(3.19.6)}\}
\end{array}
$$

$\mathsf{M} := \mathsf{M} \oplus \{\Psi x \mapsto \hat{\Psi}_{x,w}(e)\}$

(3) Similar to (2).

(5) $\hat{\Psi}_w(p \triangleleft b \triangleright q)$

$=$ {Definition of $\hat{\Psi}_w$ and (end $- \triangleleft\ \triangleright$ right dist)(3.19.3)}

$\mathsf{var}\ w;\ w := \mathsf{M}[\Psi w];\ ((\mathsf{end}\,\mathsf{M};\ p) \triangleleft b \triangleright (\mathsf{end}\,\mathsf{M};\ q));\ \hat{\Psi}_w^{-1}$

$=$ {(:= $- \triangleleft\ \triangleright$ right dist)(3.15.7) and Definition 4.6(Simulation as substitution)}

$\mathsf{var}\ w;\ ((w := \mathsf{M}[\Psi w];\ \mathsf{end}\,\mathsf{M};\ p) \triangleleft \hat{\Psi}_w(b) \triangleright (w := \mathsf{M}[\Psi w];\ \mathsf{end}\,\mathsf{M};\ q));\ \hat{\Psi}_w^{-1}$

$=$ {(var $- \triangleleft\ \triangleright$ right dist)(3.19.3) and (; $- \triangleleft\ \triangleright$ left dist)(3.14.6)}

$\hat{\Psi}_w(p) \triangleleft \hat{\Psi}_w(b) \triangleright \hat{\Psi}_w(q)$

(6) $\hat{\Psi}_w(b * p)$

$\sqsubseteq$ {Theorem 3.4(Distributivity of simulation through) μ}

$\mu\, X \bullet \hat{\Psi}_w((p;\ \hat{\Psi}_w^{-1}(X)) \triangleleft b \triangleright \mathsf{skip})$

$\sqsubseteq$ {(5) and (1)}

$\mu\, X \bullet (\hat{\Psi}_w(p;\ \hat{\Psi}_w^{-1}(X)) \triangleleft \hat{\Psi}_w(b) \triangleright \mathsf{skip})$

$\sqsubseteq$ {(4) and Theorem 3.3(Lift of simulation)}

$\hat{\Psi}_w(b) * \hat{\Psi}_w(p)$

(7) Similar to (5).

(8) Recall that $(b \rightarrow p) = (b_\perp;\ p)$. Therefore the proof follows from (4), (7).

■

The above rule deals with the global variables. But the local variables v (introduced either by the programmer or during the simplification of expressions) also require locations to hold their values during execution. For simplicity, we assume that all local variables are distinct, and that they are also different from the global variables. We extend the symbol table Ψ to cover all the local variables v:

$$\Phi \stackrel{def}{=} \Psi \cup \{v \mapsto n\}$$

where n is a list of addresses distinct from the ones already used by Ψ.

The next lemma states that the encoding program $\hat{\Psi}_w$ (when followed by a declaration of the local variables v) can be refined to an encoding program $\hat{\Phi}_{v,w}$ that deals with the global and the local variables.

Lemma 4.8 (Extending the encoding program)

$$\hat{\Psi}_w;\ \mathsf{var}\ v\ \sqsubseteq \hat{\Phi}_{v,w}$$

Proof:

$$
\begin{array}{ll}
 & \hat{\Psi}_w;\ \mathsf{var}\ v \\
\sqsubseteq & \{\langle \mathsf{var}\ \text{change scope}\rangle(3.19.2),\ \langle \mathsf{var}\ \text{assoc}\rangle(3.19.1)\ \text{and} \\
 & \ \langle \mathsf{var}- := \ \text{initial value}\rangle(3.19.4)\} \\
 & \mathsf{var}\ v, w;\ v := \mathsf{M}[n];\ w := \mathsf{M}[\Psi w];\ \mathsf{end}\ \mathsf{M} \\
= & \{\langle := \ \text{combination}\rangle(3.15.4)\ \text{and}\ \langle := \ \text{identity}\rangle(3.15.2)\} \\
 & \mathsf{var}\ v, w;\ v, w := \mathsf{M}[n, \Psi w];\ \mathsf{end}\ \mathsf{M} \\
= & \{\text{Definition of}\ \Phi\ \text{and}\ (\Phi v, \Phi w) = \Phi(v, w)\} \\
 & \mathsf{var}\ v, w;\ v, w := \mathsf{M}[\Phi(v, w)];\ \mathsf{end}\ \mathsf{M} \\
= & \{\text{Definition 4.4(Encoding program)}\} \\
 & \hat{\Phi}_{v,w}
\end{array}
$$

■

The decoding program can be extended in an analogous way.

Lemma 4.7 (Extending the decoding program)

$$\mathsf{end}\ v;\ \hat{\Psi}_w^{-1} \sqsubseteq \hat{\Phi}_{v,w}^{-1}$$

Proof:

$$
\begin{array}{ll}
 & \mathsf{end}\ v;\ \hat{\Psi}_w^{-1} \\
\sqsubseteq & \{\langle \mathsf{end}\ \text{change scope}\rangle(3.19.2),\ \langle \mathsf{end}\ \text{assoc}\rangle(3.19.1)\ \text{and} \\
 & \ \langle \mathsf{var}- := \ \text{initial value}\rangle(3.19.4)\} \\
 & \mathsf{var}\ \mathsf{M};\ \mathsf{M} := \mathsf{M} \oplus \{n \mapsto v\};\ \mathsf{M} := \mathsf{M} \oplus \{\Psi w \mapsto w\};\ \mathsf{end}\ v, w \\
= & \{\langle := \ \text{combination}\rangle(3.15.4)\ \text{and Definition of}\ \Phi\} \\
 & \mathsf{var}\ \mathsf{M};\ \mathsf{M} := \mathsf{M} \oplus \{\Phi(v, w) \mapsto v, w\};\ \mathsf{end}\ v, w \\
= & \{\text{Definition 4.5(Decoding program)}\} \\
 & \hat{\Phi}_{v,w}^{-1}
\end{array}
$$

■

Using the above two lemmas we show how to assign locations in the memory M to hold the values of the local variables v.

Rule 4.14 (Allocating local variables)

$$\hat{\Psi}_w(\mathsf{dec}\ v, \mathsf{P}, \mathsf{A} \bullet p) \sqsubseteq \mathsf{dec}\ \mathsf{P}, \mathsf{A} \bullet \hat{\Phi}_{v,w}(p)$$

Proof:

$$\hat{\Psi}_w(\mathsf{dec}\ v, \mathsf{P}, \mathsf{A} \bullet p)$$

$$
\begin{aligned}
&= \{\text{Definition 3.6(Simulation function)}\}\\
&\quad \hat{\Psi}_w;\ (\mathsf{dec}\ v, \mathsf{P}, \mathsf{A} \bullet p);\ \hat{\Psi}_w^{-1}\\
&= \{(\mathsf{dec}\ \text{assoc})(3.18.1)\ \text{and}\ (\mathsf{dec} - (\mathsf{var}, \mathsf{end})\ \text{conversion})(3.19.10)\}\\
&\quad \hat{\Psi}_w;\ \mathsf{var}\ v;\ (\mathsf{dec}\ \mathsf{P}, \mathsf{A} \bullet p);\ \mathsf{end}\ v;\ \hat{\Psi}_w^{-1}\\
&\sqsubseteq \{\text{Lemmas 4.6 and 4.7}\}\\
&\quad \hat{\Phi}_{v,w};\ (\mathsf{dec}\ \mathsf{P}, \mathsf{A} \bullet p);\ \hat{\Phi}_{v,w}^{-1}\\
&= \{(;-\mathsf{dec}\ \text{left dist})(3.18.7)\ \text{and}\ (;-\mathsf{dec}\ \text{right dist})(3.18.7)\}\\
&\quad \mathsf{dec}\ \mathsf{P}, \mathsf{A} \bullet \hat{\Phi}_{v,w}(p)
\end{aligned}
$$

■

The next theorem summarises the outcome of this phase of compilation.

Theorem 4.11 (Data refinement)
Consider a program of the form

$$\mathsf{dec}\ v, \mathsf{A} \bullet q$$

where q contains no local declaration, all assignments in q are simple and the only boolean condition in q is the variable A. Then there is a program r such that

$$\hat{\Psi}_w(\mathsf{dec}\ v, \mathsf{A} \bullet q)\ \sqsubseteq\ \mathsf{dec}\ \mathsf{A} \bullet r$$

where r preserves the control structure of q but operates exclusively on the concrete state represented by M.
Proof: Using Rule 4.14 (Allocating local variables), we transform

$$\hat{\Psi}_w(\mathsf{dec}\ v, \mathsf{A} \bullet q)$$

into

$$\mathsf{dec}\ \mathsf{A} \bullet \hat{\Phi}_{v,w}(q)$$

Then by structural induction using Rule 4.13 (Piecewise data refinement), we transform $\hat{\Phi}_{v,w}(q)$ into r. ■

It is worth noting that this theorem does not prevent q from being a normal form program. This suggests that we can carry out data refinement either before or after control refinement. In the latter case, note that Rule 4.13 (in particular (7),(8) and (9)) covers the additional operators used to describe a normal form program.

4.7 The Compilation Process

In principle, there is no reason for imposing any order on the phases of compilation. But there are practical considerations which favour some permutations. In particular, we

suggest that the simplification of expressions should be the first phase. Performing data refinement as a first step would not be appealing, because the simplification of expressions normally generates new local variables. Therefore a second phase of data refinement would be required to deal specifically with the local declarations.

We have also explored the possibility of carrying out control elimination as a first step. It turned out to be necessary to allocate relative addresses to commands of the source program, and (in the end of the process) to convert them into absolute addresses. The standard approach would be to model an address as a pair. For example, the normal form of an assignment statement $x := \mathsf{uop}\, y$ (where x and y are single variables) would be

$$\mathsf{P} : [(k,0), \mathsf{P} = (k,0) \rightarrow (x := \mathsf{uop}\, y;\ \mathsf{P} := (k+1,0)), (k+1,0)]$$

As a result of expression simplification, the above guarded command would be transformed into

$$\mathsf{P} = (k,0) \rightarrow ((\mathbf{dec}\, \mathsf{A} \bullet \mathsf{A} := y;\ \mathsf{A} := \mathsf{uop}\, \mathsf{A};\ x := \mathsf{A});\ \mathsf{P} := (k+1,0))$$

containing only simple assignments. In a similar way, all the other assignments and conditions of the source program would be simplified. However, we must eventually end with a simple normal form program, and there are two remaining tasks. One is to move the local declarations generated by this process out from the body of the loop; this is justified by the distribution laws of declaration with the other operators of our language. The other task is to split the above command into a series of guarded commands (one for each assignment) and model the sequencing by incrementing the second component of the pair representing a relative address:

$$\begin{array}{l} \mathsf{P} = (k,0) \rightarrow \mathsf{A} := y;\ \mathsf{P} := (k,1)\ \Box \\ \mathsf{P} = (k,1) \rightarrow \mathsf{A} := \mathsf{uop}\, \mathsf{A};\ \mathsf{P} := (k,2)\ \Box \\ \mathsf{P} = (k,2) \rightarrow x := \mathsf{A};\ \mathsf{P} := (k+1,0) \end{array}$$

This is necessary to ensure that each instruction will be placed in a separate memory location. Assuming that the relative address $(k,0)$ will eventually turn into the absolute address j, the above becomes

$$\begin{array}{l} \mathsf{P} = j \rightarrow \mathsf{A}, \mathsf{P} := y, (\mathsf{P}+1)\ \Box \\ \mathsf{P} = (j+1) \rightarrow \mathsf{A}, \mathsf{P} := (\mathsf{uop}\, \mathsf{A}), (\mathsf{P}+1)\ \Box \\ \mathsf{P} = (j+2) \rightarrow x, \mathsf{P} := \mathsf{A}, (\mathsf{P}+1) \end{array}$$

which is in the required form. While the conversion from relative to absolute addresses is in principle a simple process, there is the associated proof obligation to show that the iteration is not affected by this change of data representation. It seems sensible to avoided these complications by starting the compilation process with the simplification of expressions.

Once the expressions are simplified, the order in which data refinement and control elimination are carried out is irrelevant. The following theorem summarises the compilation process.

Theorem 4.12 (Compilation Process)
Let p be an arbitrary source program. Given a constant s, and a symbol table Ψ which maps each global variable of p to the address of the memory M allocated to hold its value, there is a constant f and a sequence of machine instructions held in m between locations s and f such that

$$\hat{\Psi}_w(p) \sqsubseteq \mathsf{dec}\, \mathsf{P}, \mathsf{A} \bullet \mathsf{P} := s;\ (s \leq \mathsf{P} < f) * \mathsf{m}[\mathsf{P}];\ (\mathsf{P} = f)_\perp$$

Proof: From theorems 4.8 (Expression simplification), 4.9 (Control elimination) and 4.11 (Data refinement), $\hat{\Psi}_w(p)$ is transformed into

$$\mathsf{dec}\, \mathsf{P}, \mathsf{A} \bullet \mathsf{P} := s;\ (s \leq \mathsf{P} < f) * p;\ (\mathsf{P} = f)_\perp$$

where p is a guarded command set of the form

$$\Box_{s \leq k < f} \mathsf{P} = k \rightarrow q_k$$

and each q_k is an assignment which corresponds to one of the patterns used to define the machine instructions. This guarded command set is an abstract representation of the memory m, whose contents are affected by the compilation process as follows:

$$\mathsf{m}[k] \; = \; q_k, \quad \text{for } s \leq k < f$$

and the value of m outside the range $s..(f-1)$ is arbitrary. This last step corresponds to the actual loading process. ■

A more detailed description of how the reduction theorems can be used as rewrite rules to carry out compilation is given in Chapter 6, which deals with the mechanisation of compilation and proofs.

Chapter 5

Procedures, Recursion and Parameters

> The main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation [...]
> Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts.
>
> — E. W. Dijkstra

In this chapter we extend the source language with more elaborate notions: procedures, recursion and parameters. We show how each of these can be eliminated through reduction to normal form; but we leave open the choice of a target machine to implement them.

Most practical programming languages group these notions into a single construction which allows parameterised recursive procedures. Here we follow Hoare [40], Morgan [54] and Back [4], and treat them separately—both syntactically and semantically. Existing practice is in most cases realised by appropriate combinations of these features; but the separation gives more freedom and elegance, and helps to simplify the overall task.

5.1 Notation

In order to prove the correctness of the elimination rule for recursion, we need to extend our language with sequence variables together with some usual operations. By convention, a variable name decorated with _ denotes a sequence variable. The following operators

are used:

$\langle\rangle$	the empty sequence
$\langle x\rangle$	the singleton sequence with element x
$\bar{x} \frown \bar{y}$	the concatenation of $\bar{x}$ and $\bar{y}$
head $\bar{x}$	the leftmost element of $\bar{x}$
last $\bar{x}$	the rightmost element of $\bar{x}$
front $\bar{x}$	the sequence which results from removing the last element of $\bar{x}$
tail $\bar{x}$	the sequence which results from removing the head element of $\bar{x}$
$\#\bar{x}$	the number of elements of $\bar{x}$

The result of head, last, front and tail, when applied to empty sequences, is arbitrary. Some familiar laws of sequences are reviewed below.

Law 5.1.1 (laws of sequences)
(1) $\mathsf{head}(\langle x\rangle \frown \bar{x}) = x = \mathsf{last}(\bar{x} \frown \langle x\rangle)$
(2) $\mathsf{front}(\bar{x} \frown \langle x\rangle) = \bar{x} = \mathsf{tail}(\langle x\rangle \frown \bar{x})$
(3) If $\bar{x}$ is non-empty then
$(\langle \mathsf{head}\ \bar{x}\rangle \frown \mathsf{tail}\ \bar{x}) = \bar{x} = (\mathsf{front}\ \bar{x} \frown \langle \mathsf{last}\ \bar{x}\rangle)$

Although the notion of sequences is necessary in the intermediate steps of our proof, the elimination rule for recursion mentions only patterns which can be implemented by stack operations. To emphasise this point we define the following:

Definition 5.1 (Stack operations)

$$\begin{array}{lcl} \mathsf{push}(x,\bar{x}) & \stackrel{def}{=} & (\bar{x} := \langle x\rangle \frown \bar{x}) \\ \mathsf{pop}(x,\bar{x}) & \stackrel{def}{=} & (x,\bar{x} := \mathsf{head}\ \bar{x}, \mathsf{tail}\ \bar{x}) \\ \mathsf{empty}\ \bar{x} & \stackrel{def}{=} & (\bar{x} = \langle\rangle) \end{array}$$

■

For a list of variables $x = x_1, \ldots, x_n$ we use the abbreviations

$$\begin{array}{lcl} \mathsf{push}(x,\bar{x}) & \stackrel{def}{=} & \mathsf{push}(x_1,\bar{x}_1);\ \ldots;\ \mathsf{push}(x_n,\bar{x}_n) \\ \mathsf{pop}(x,\bar{x}) & \stackrel{def}{=} & \mathsf{pop}(x_1,\bar{x}_1);\ \ldots;\ \mathsf{pop}(x_n,\bar{x}_n) \end{array}$$

From the laws of sequence, it follows that the pair $(\mathsf{pop}(x,\bar{x}), \mathsf{push}(x,\bar{x}))$ is a simulation.

Law 5.1.2 $\mathsf{pop}(x,\bar{x});\ \mathsf{push}(x,\bar{x}) \sqsubseteq \mathsf{skip} = \mathsf{push}(x,\bar{x});\ \mathsf{pop}(x,\bar{x})$
(pop – push simulation)

The following law suggests that var and end operate on an implicit stack that can be made explicit by using push and pop.

Law 5.1.3 If $\bar{x}$ is not free in p or q, then
$\text{dec}\, x \bullet p[X \leftarrow \text{var}\, x;\ q;\ \text{end}\, x] \sqsubseteq \text{dec}\, x, \bar{x} \bullet p[X \leftarrow \text{push}(x, \bar{x});\ q;\ \text{pop}(x, \bar{x})]$
((var, end) – (push, pop) conversion)

Recall that $p[X \leftarrow r]$ is the result of substituting r for every free occurrence of X in p, where capture of free identifiers of r is avoided by renaming local declarations in p. The inequality in the above law is a consequence of the fact that $\text{push}(x, \bar{x})$ leaves x unchanged, while $\text{var}\, x$ assigns an arbitrary value to x.

5.2 Procedures

We use the notation

$$\text{proc}\ X \mathrel{\hat{=}} p \bullet q$$

to declare a non-recursive, parameterless procedure named X with body p. The program q following the symbol $\bullet$ is the scope of the procedure. Occurrences of X in q are interpreted as call commands. The semantics of a call is textual substitution, like the *copy rule* of Algol 60.

Definition 5.2 (Procedures)

$$(\text{proc}\ X \mathrel{\hat{=}} p \bullet q) \stackrel{def}{=} q[X \leftarrow p]$$

■

Notice that the above definition could be used as a rewrite rule to eliminate procedures even prior to the reduction of the procedure body and scope to normal form; this technique is known as *macro-expansion*. But this may substantially increase the size of the target code if the scope of the procedure contains a large number of call statements. An alternative is to compile the procedure body and calls into separate segments of code so that, during execution of a procedure call, control passes back and forth between these segments in a manner that simulates the copy rule.

As usual, the idea is to assume that the components of the procedure construction (the body and the scope) are in normal form, as in

$$\text{proc}\ X \mathrel{\hat{=}} v : [a_0,\ b_0 \to p,\ c_0] \bullet v : [a, \begin{pmatrix} \quad b_1 \to (X;\ v :\in r_1) \\ \square\ \ \dots \\ \square\ \ b_n \to (X;\ v :\in r_n) \\ \square\ \ b \to q \end{pmatrix}, c]$$

where the scope of the procedure may have an arbitrary number of call commands still to be eliminated; each r_i stands for the return address of the corresponding call. The guarded

command $b \to q$ stands for the remaining part of the code which does not contain any calls of procedure X. By definition, the above is equivalent to

$$v : [a, \begin{pmatrix} & b_1 \to (v : [a_0,\ b_0 \to p,\ c_0];\ \ v :\in r_1) \\ \Box & \ldots \\ \Box & b_n \to (v : [a_0,\ b_0 \to p,\ c_0];\ \ v :\in r_n) \\ \Box & b \to q \end{pmatrix}, c]$$

Therefore the reduction rule for procedures is a special case of a theorem about removal of nested normal form. We adopt a standard strategy, keeping a single copy of the code of the procedure body. Whenever any of the conditions $b_1, \ldots, b_n$ is true, the corresponding return address is saved in a fresh variable, say w, and the start address of the code of the procedure body is assigned to the control variables v. On exit from the execution of the procedure body, the value of w is copied back into v. For this to be valid, it is necessary that each r_i is not changed by the procedure body (that is, no free variable in any r_i is assigned by the procedure body). But this is not a serious limitation, since in practice the free variables of r_i are the control variables v, and these are local to the normal form program which implements the procedure body. The following theorem formalises the overall strategy.

Theorem 5.1 (Nested normal form)
If w is not free on the left-hand side of the following inequation, and r_i is not changed by $v : [a_0,\ b_0 \to p,\ c_0]$, then

$$v : [a, \begin{pmatrix} & b_1 \to (v : [a_0,\ b_0 \to p,\ c_0];\ \ v :\in r_1) \\ \Box & \ldots \\ \Box & b_n \to (v : [a_0,\ b_0 \to p,\ c_0];\ \ v :\in r_n) \\ \Box & b \to q \end{pmatrix}, c]$$

$$\sqsubseteq$$

$$v, w : [a,\ T,\ c]$$

$$\text{where } T = \begin{pmatrix} & b_1 \to (w :\in r_1[v \leftarrow w];\ v :\in a_0) \\ \Box & \ldots \\ \Box & b_n \to (w :\in r_n[v \leftarrow w];\ v :\in a_0) \\ \Box & c_0 \to v := w \\ \Box & b_0 \to p \ \Box\ b \to q \end{pmatrix}$$

Proof: First we show how each copy of the procedure body can be recovered by performing symbolic execution on the right-hand side of the above inequation. Let r_i stand for any of the return addresses $r_1, \ldots, r_n$, and $d = (b_1 \vee \ldots \vee b_n \vee c_0 \vee b_0 \vee b)$.

$$\begin{array}{rl} (1) & w :\in r_i[v \leftarrow w];\ v :\in a_0;\ d * T \\ = & \{(* \text{ sequence})(3.17.7) \text{ and } (* - \Box \text{ elim})(3.17.4)\} \\ & w :\in r_i[v \leftarrow w];\ v :\in a_0;\ b_0 * p;\ d * T \\ \sqsupseteq & \{c_{0\bot} \sqsubseteq \textbf{skip} \text{ and } (* - \Box \text{ unfold})(3.17.3)\} \\ & w :\in r_i[v \leftarrow w];\ v :\in a_0;\ b_0 * p;\ c_{0\bot};\ v := w;\ d * T \end{array}$$

$$
\begin{array}{ll}
= & \{\langle \mathbf{dec}\ \text{introduction}\rangle(3.19.11) \text{ and Definition 4.1(Normal form)}\} \\
 & w :\in r_i[v \leftarrow w];\ v : [a_0,\ b_0 \rightarrow p,\ c_0];\ v := w;\ d * T \\
\sqsupseteq & \{w :\in r_i[v \leftarrow w] \text{ commutes with } v : [a_0,\ b_0 \rightarrow p,\ c_0]\ (3.16.9)\} \\
 & v : [a_0,\ b_0 \rightarrow p,\ c_0];\ w :\in r_i[v \leftarrow w];\ v := w;\ d * T \\
\sqsupseteq & \{\langle :\in \text{ refined by } :=\rangle(3.16.8)\} \\
(2) & v : [a_0,\ b_0 \rightarrow p,\ c_0];\ w :\in r_i[v \leftarrow w];\ v :\in r_i;\ d * T
\end{array}
$$

Then we have:

$$
\begin{array}{ll}
 & RHS \\
\sqsupseteq & \{\langle * \text{ replace guarded command}\rangle(3.17.5) \text{ and } (1) \sqsupseteq (2)\} \\
 & v, w : [a, \left(\begin{array}{ll} & b_1 \rightarrow (v : [a_0,\ b_0 \rightarrow p,\ c_0];\ w :\in r_1[v \leftarrow w];\ v :\in r_1) \\ \Box & \ldots \\ \Box & b_n \rightarrow (v : [a_0,\ b_0 \rightarrow p,\ c_0];\ w :\in r_n[v \leftarrow w];\ v :\in r_n) \\ \Box & c_0 \rightarrow v := w \\ \Box & b_0 \rightarrow p \ \Box\ b \rightarrow q \end{array}\right), c] \\
\sqsupseteq & \{\text{Lemma 4.3(Eliminate guarded command)}\} \\
 & v, w : [a, \left(\begin{array}{ll} & b_1 \rightarrow (v : [a_0,\ b_0 \rightarrow p,\ c_0];\ w :\in r_1[v \leftarrow w];\ v :\in r_1) \\ \Box & \ldots \\ \Box & b_n \rightarrow (v : [a_0,\ b_0 \rightarrow p,\ c_0];\ w :\in r_n[v \leftarrow w];\ v :\in r_n) \\ \Box & b \rightarrow q \end{array}\right), c] \\
\sqsupseteq & \{\langle \mathbf{dec} - * \text{ dist}\rangle(3.18.11) \text{ and } \langle \mathbf{dec}\ \text{elim}\rangle(3.18.3)\} \\
 & v : [a, \left(\begin{array}{ll} & b_1 \rightarrow (v : [a_0,\ b_0 \rightarrow p,\ c_0];\ (\mathbf{dec}\, w \bullet w :\in r_1[v \leftarrow w]);\ v :\in r_1) \\ \Box & \ldots \\ \Box & b_n \rightarrow (v : [a_0,\ b_0 \rightarrow p,\ c_0];\ (\mathbf{dec}\, w \bullet w :\in r_n[v \leftarrow w]);\ v :\in r_n) \\ \Box & b \rightarrow q \end{array}\right), c] \\
\sqsupseteq & \{\langle \mathbf{dec} - :\in \text{ final value}\rangle(3.18.6) \text{ and } \langle \mathbf{dec}\ \text{elim}\rangle(3.18.3)\} \\
 & LHS
\end{array}
$$

■

5.3 Recursion

We have already introduced the notation

$$\mu X \bullet p$$

which defines a recursive, parameterless program named X with body p. Unlike a procedure, a recursive program cannot be called from outside its body: only recursive calls are allowed. Occurrences of X in p are interpreted as recursive calls. The semantics of recursion is given by fixed point laws (see Section 3.8).

Before giving the reduction rule for recursive programs, we introduce some abbreviations which will help in structuring the proof. The left-hand side of the reduction rule is a recursive program of the form

$$LHS \;=\; \mu X \bullet v : [a_0, \begin{pmatrix} & b \rightarrow (X;\;\; v :\in r) \\ \Box & b_0 \rightarrow p \end{pmatrix}, c_0]$$

where its body is in normal form, except for the recursive calls. For conciseness, we assume that there is only one call to be eliminated (no free occurrence of X in p); the theorem is easily generalised for an arbitrary number of calls, as in the case of procedures.

The recursive definition can be eliminated by reducing the above to

$$MID \;=\; v, \bar{v} : [a_0 \wedge \mathsf{empty}\, \bar{v},\; S,\; c_0 \wedge \mathsf{empty}\, \bar{v}]$$

$$\text{where} \quad S \;=\; \begin{pmatrix} & b \rightarrow (v :\in r;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & (c_0 \wedge \neg \mathsf{empty}\, \bar{v}) \rightarrow \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \rightarrow p \end{pmatrix}$$

As in the previous section, a call is implemented by saving the return address before control is transferred; this address is then used to resume control. In the case of procedures, a local variable was used for this purpose, but for recursive calls we need the notion of a stack. By assuming that the stack is empty initially, we can distinguish between the exit from a recursive call of the program and the end of its execution. In the former case, the condition c_0 is true, but the stack is not yet empty; then control is resumed by popping the stack and assigning the popped value to v. The exit condition of the entire program is $c_0 \wedge \mathsf{empty}\, \bar{v}$.

Although our emphasis is on the control structure of the program, note that v may be an arbitrary list of variables, possibly including data variables declared by the programmer. The overall task is simplified by not distinguishing between these two kinds of variable.

An alternative implementation of *LHS* is given by the program:

$$RHS \;=\; v, \bar{v} : [a \wedge \mathsf{empty}\, \bar{v},\; T,\; c \wedge \mathsf{empty}\, \bar{v}]$$

$$\text{where} \quad T \;=\; \begin{pmatrix} & a \rightarrow (v :\in c;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & b \rightarrow (v :\in r;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & c_0 \rightarrow \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \rightarrow p \end{pmatrix}$$

Its first action is to push onto the stack a value which satisfies the condition c, an exit condition for the loop associated with the above normal form program. In this way we ensure that, whenever c_0 is true, the stack is non-empty, since the last value to be popped satisfies a termination condition of the loop. The advantage of this implementation is that it avoids the use of the condition $\neg\mathsf{empty}\, \bar{v}$ as part of a guard. Therefore *RHS* is more suitable for a low-level implementation.

A convenient way to prove $LHS \sqsubseteq RHS$ is to show that $LHS \sqsubseteq MID$ and that $MID \sqsubseteq RHS$. We use the following lemmas.

Lemma 5.1 (Symbolic execution of S) Let $d = (b \vee (c_0 \wedge \neg \mathsf{empty}\,\bar{v}) \vee b_0)$.

$$b^\top;\ d * S \ \sqsupseteq\ (\mathsf{empty}\,\bar{v})_\perp;\ MID;\ v :\in r;\ d * S$$

■

Lemma 5.2 (Symbolic execution of T) Let $d = (a \vee b \vee c_0 \vee b_0)$.

$$a^\top;\ d * T \ \sqsupseteq\ (\mathsf{empty}\,\bar{v})_\perp;\ MID;\ v :\in c;\ d * T$$

■

The proof of Lemma 5.1 is given in Appendix B. The proof of Lemma 5.2 is similar. The reduction theorem for recursive programs can now be proved.

Theorem 5.2 (Recursion)
Let LHS, MID and RHS be as defined above. If X is not free in p, and $\bar{v}$ occurs only where explicitly shown, then $LHS \sqsubseteq RHS$.

Proof:
($LHS \sqsubseteq MID$)

$$MID$$

$$\sqsupseteq \{(* \text{ replace guarded command})(3.17.5),\ \text{Lemma 5.1(Symbolic execution of } S)\}$$

$$v, \bar{v} : \left[(a_0 \wedge \mathsf{empty}\,\bar{v}), \begin{pmatrix} & b \rightarrow (\mathsf{empty}\,\bar{v})_\perp;\ MID;\ v :\in r \\ \Box & (c_0 \wedge \neg\mathsf{empty}\,\bar{v}) \rightarrow \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \rightarrow p \end{pmatrix}, (c_0 \wedge \mathsf{empty}\,\bar{v})\right]$$

$$\sqsupseteq \{\text{Lemma 4.3(Eliminate guarded command)}\}$$

$$v, \bar{v} : \left[(a_0 \wedge \mathsf{empty}\,\bar{v}), \begin{pmatrix} & b \rightarrow (\mathsf{empty}\,\bar{v})_\perp;\ MID;\ v :\in r \\ \Box & b_0 \rightarrow p \end{pmatrix}, (c_0 \wedge \mathsf{empty}\,\bar{v})\right]$$

$$\sqsupseteq \{(b_\perp - b^\top \text{ simulation})(3.11.10)\}$$

$$v, \bar{v} : \left[a_0, \begin{pmatrix} & b \rightarrow (\mathsf{empty}\,\bar{v})^\top;\ (\mathsf{empty}\,\bar{v})_\perp;\ MID;\ v :\in r;\ (\mathsf{empty}\,\bar{v})_\perp \\ \Box & b_0 \rightarrow p \end{pmatrix}, c_0\right]$$

$$\sqsupseteq \{b^\top;\ b_\perp \ =\ b^\top \text{ and } (b^\top;\ p \text{ commute})(3.11.11)\}$$

$$v, \bar{v} : \left[a_0, \begin{pmatrix} & b \rightarrow (MID;\ v :\in r;\ (\mathsf{empty}\,\bar{v})^\top;\ (\mathsf{empty}\,\bar{v})_\perp \\ \Box & b_0 \rightarrow p \end{pmatrix}, c_0\right]$$

$$\sqsupseteq \{b^\top;\ b_\perp \ =\ b^\top \ \sqsupseteq\ \mathsf{skip} \text{ and } (\text{dec elim})(3.18.3)\}$$

$$v : \left[a_0, \begin{pmatrix} & b \rightarrow (MID;\ v :\in r) \\ \Box & b_0 \rightarrow p \end{pmatrix}, c_0\right]$$

From the above and (μ least fixed point)(3.8.2), it follows that $LHS \sqsubseteq MID$.

(MID ⊑ RHS)

$$
\begin{array}{ll}
& RHS \\
\sqsupseteq & \{(*\text{ replace guarded command})(3.17.5),\ \text{Lemma 5.2(Symbolic execution of } T)\} \\
& v, \bar{v} : \left[(a \land \mathsf{empty}\,\bar{v}), \begin{pmatrix} & a \to (\mathsf{empty}\,\bar{v})_\bot;\ MID;\ v :\in c \\ \Box & b \to (v :\in r;\ \mathsf{push}(v, \bar{v});\ v :\in a_0) \\ \Box & c_0 \to \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \to p \end{pmatrix}, (c \land \mathsf{empty}\,\bar{v})\right] \\
\sqsupseteq & \{\text{Lemma 4.3(Eliminate guarded command)}\} \\
& v, \bar{v} : [(a \land \mathsf{empty}\,\bar{v}), (a \to (\mathsf{empty}\,\bar{v})_\bot;\ MID;\ v :\in c), (c \land \mathsf{empty}\,\bar{v})] \\
\sqsupseteq & \{(b_\bot - b^\top \text{ simulation})(3.11.10)\} \\
& v, \bar{v} : [a, (a \to (\mathsf{empty}\,\bar{v})^\top;\ (\mathsf{empty}\,\bar{v})_\bot;\ MID;\ v :\in c;\ (\mathsf{empty}\,\bar{v})_\bot), c] \\
\sqsupseteq & \{b^\top;\ b_\bot\ =\ b^\top \text{ and } (b^\top;\ p \text{ commute})(3.11.11)\} \\
& v, \bar{v} : [a, (a \to (MID;\ v :\in c;\ (\mathsf{empty}\,\bar{v})^\top;\ (\mathsf{empty}\,\bar{v})_\bot), c] \\
\sqsupseteq & \{b^\top;\ b_\bot\ =\ b^\top\ \sqsupseteq\ \mathsf{skip} \text{ and } (\text{dec elim})(3.18.3)\} \\
& v : [a, (a \to (MID;\ v :\in c), c] \\
\sqsupseteq & \{\text{Lemma 4.1(Primitive commands)}\} \\
& MID
\end{array}
$$

■

5.4 Parameterised Programs

Here we show that parameterisation can be treated in complete isolation from procedures. Let p be a program and x a variable. Then

$$par\, x \bullet p$$

is a parameterised program, where *par* stands for some parameter transmission mechanism; here we will deal with value-result (valres), value (val), result (res) and name parameters. The latter kind is restricted so that the actual parameter must be a variable and no *aliasing* must occur; in this case we can prove that parameterisation by name and by value-result have the same effect. Although we do not address parameterisation by reference explicitly, it coincides with parameterisation by name when variables (rather than arbitrary expressions) are used as arguments.

We adopt the conventional notation of function application for the instantiation of a parameterised program. The effect of an instantiation varies according to the type of parameterisation. The definitions are given below. In all cases, z must be a fresh variable, occurring only where explicitly shown.

Definition 5.3 (Value-result parameters)

$$(\mathsf{valres}\, x \bullet p)(y) \ \stackrel{def}{=}\ \mathsf{dec}\, z \bullet z := y;\ p[x \leftarrow z];\ y := z$$

■

Definition 5.4 (Value parameters)

$$(\mathsf{val}\, x \bullet p)(e) \stackrel{def}{=} \mathsf{dec}\, z \bullet z := e;\ p[x \leftarrow z]$$

■

Definition 5.5 (Result parameters)

$$(\mathsf{res}\, x \bullet p)(y) \stackrel{def}{=} \mathsf{dec}\, z \bullet p[x \leftarrow z];\ y := z$$

■

Definition 5.6 (Name parameters)
If y is not free in p, then

$$(\mathsf{name}\, x \bullet p)(y) \stackrel{def}{=} p[x \leftarrow y]$$

■

These definitions are reasonably standard. They appear, for example, in [4, 54, 60]. In [4] the notion of refinement is generalised for parameterised statements

> Let $P = par\ x \bullet p$ and $Q = par\ x \bullet q$. Then
> $P \sqsubseteq Q \stackrel{def}{=} P(t) \sqsubseteq Q(t)$ for all valid arguments t

and it is shown that the crucial property of monotonicity with respect to refinement is retained by the new constructs:

(1) $p \sqsubseteq q \Rightarrow par\ x \bullet p \sqsubseteq par\ x \bullet q$
(2) Let P and Q be parameterised programs. Then, for any valid argument t
$P \sqsubseteq Q \Rightarrow P(t) \sqsubseteq Q(t)$

For name parameters, this result is not true in general. The instantiation may lead to aliasing, in which case monotonicity is lost. This is why we need the condition attached to Definition 5.6.

Multiple parameterisation of a particular kind can be achieved by allowing programs to be parameterised by lists of variables. The corresponding instantiations are similarly extended to allow lists of arguments. In this case, the two lists must be of equal length and the association between parameters and arguments is positional. Notice that we do not need to change the previous definitions, as our language allows multiple declaration and multiple assignment, and we have already introduced the notation for multiple substitution. For example,

$$(\mathsf{res}\, x_1, x_2 \bullet p)(y_1, y_2) \stackrel{def}{=} \mathsf{dec}\, z_1, z_2 \bullet p[x_1, x_2 \leftarrow z_1, z_2];\ y_1, y_2 := z_1, z_2$$

5.4 $p \mathrel{\hat{=}} x := x+1$
$(\mathsf{val}\ x \bullet p)(e)$
$= \mathsf{dec}\ z \bullet z := e;\ z := z+1$
$=$ SKIP

However, except for call by value, an extra restriction must be observed: the list of actuals must be disjoint. For example, in the above case this is necessary to ensure that the multiple assignment $y_1, y_2 := z_1, z_2$ is defined.

Multiple parameterisation of (possibly) different kinds can be achieved by combining the effects of the related definitions. In this case we use a semicolon to separate the parameter declarations. As an example we have

$$(\mathsf{val}\, x_1;\ \mathsf{res}\, x_2 \bullet p)(e, y) \overset{def}{=} \mathsf{dec}\, z_1, z_2 \bullet z_1 := e;\ p[x_1, x_2 \leftarrow z_1, z_2];\ y := z_2$$

In the remainder of this chapter we will confine our attention to single parameterisation, but the results can be easily extended to multiple parameterisation.

Definitions 5.3–5.6 above could be used directly as elimination rules for parameter passing. However, this would not allow sharing the code of a parameterised program when instantiated with distinct arguments. This is a consequence of the renaming of variables on the right-hand sides of these definitions. Below we show how to avoid the renaming. This will need the following lemma.

Lemma 5.3 (Data refinement as substitution)
Let $\Theta \overset{def}{=} \mathsf{var}\, x;\ x := y;\ \mathsf{end}\, y$ and $\Theta^{-1} \overset{def}{=} \mathsf{var}\, y;\ y := x;\ \mathsf{end}\, x$. Then we have:
(1) (Θ,Θ^{-1}) is a simulation.
(2) If y is not free in p, then $\Theta(p) = p[x \leftarrow y]$
(Recall that $\Theta(p) = \Theta;\ p;\ \Theta^{-1}$.)

Proof:
(1) Similar to the proof that $(\hat{\Psi},\hat{\Psi}^{-1})$ is a simulation (see Theorem 4.10).
(2) By structural induction as in Theorem 4.13 (Piecewise data refinement). In this case we have an equation, rather than an inequation, because Θ^{-1} is the *exact* inverse of Θ, and vice versa. Formally, $\Theta;\ \Theta^{-1} = \mathsf{skip} = \Theta^{-1};\ \Theta$. ∎

Then we have the following elimination rules for parameter passing.

Theorem 5.3 (Elimination of value-result parameters)
If x and y are distinct, then

$$(\mathsf{valres}\, x \bullet p)(y) = \mathsf{var}\, x;\ x := y;\ p;\ y := x;\ \mathsf{end}\, x$$

Proof:

$$\begin{aligned}
& LHS \\
= \ & \{\text{Definition 5.3(Value} - \text{result parameters)}\} \\
& \mathsf{dec}\, z \bullet z := y;\ p[x \leftarrow z];\ y := z \\
= \ & \{\text{Lemma 5.3(Data refinement as substitution)}\} \\
& \mathsf{dec}\, z \bullet z := y;\ \mathsf{var}\, x;\ x := z;\ \mathsf{end}\, z;\ p;\ \mathsf{var}\, z;\ z := x;\ \mathsf{end}\, x;\ y := z \\
= \ & \{(\mathsf{end\ change\ scope})(3.19.2) \text{ and } (\mathsf{end} - \mathsf{var\ skip})(3.19.7)\}
\end{aligned}$$

$$\begin{array}{ll} & \mathsf{dec}\, z \bullet z := y;\ \mathsf{var}\, x;\ x := z;\ p;\ z := x;\ \mathsf{end}\, x;\ y := z \\ = & \{\langle \mathsf{end}\ \text{change scope}\rangle(3.19.2),\ \langle \mathsf{var}\ \text{change scope}\rangle(3.19.2)\ \text{and} \\ & \ \langle := \ \text{combination}\rangle(3.15.4)\} \\ & \mathsf{var}\, x;\ \mathsf{dec}\, z \bullet z := y;\ x := y;\ p;\ y := x;\ \mathsf{end}\, x;\ z := y \\ = & \{\langle \mathsf{dec}- := \ \text{final value}\rangle(3.18.6)\ \text{and}\ \langle ; -\mathsf{dec}\ \text{left dist}\rangle(3.18.7)\} \\ & \mathsf{var}\, x;\ (\mathsf{dec}\, z \bullet z := y);\ x := y;\ p;\ y := x;\ \mathsf{end}\, x \\ = & \{\langle \mathsf{dec}- := \ \text{final value}\rangle(3.18.6)\ \text{and}\ \langle \mathsf{dec}\ \text{elim}\rangle(3.18.3)\} \\ & RHS \end{array}$$

■

Theorem 5.4 (Elimination of value parameters)
If x and y are distinct and x does not occur in e, then

$$(\mathsf{val}\, x \bullet p)(e) \ = \ \mathsf{var}\, x;\ x := e;\ p;\ \mathsf{end}\, x$$

Proof: Similar to Theorem 5.3. ■

Theorem 5.5 (Elimination of result parameters)
If x and y are distinct, then

$$(\mathsf{res}\, x \bullet p)(y) \ = \ \mathsf{var}\, x;\ p;\ y := x;\ \mathsf{end}\, x$$

Proof: Similar to Theorem 5.3. ■

A mechanism to implement name (or reference) parameters by allowing sharing of code requires an explicit account of variable addresses and is not treated here. But the following theorem establishes tbat, in the absence of aliasing, parameterisation by name is identical to parameterisation by value-result.

Theorem 5.6 (Equivalence of name and value-result parameters)
If x and y are distinct, and y is not free in p, then

$$(\mathsf{name}\, x \bullet p)(y) \ = \ (\mathsf{valres}\, x \bullet p)(y)$$

Proof:

$$\begin{array}{ll} & RHS \\ = & \{\text{Theorem 5.3(Elimination of value} - \text{result parameters)}\} \\ & \mathsf{var}\, x;\ x := y;\ p;\ y := x;\ \mathsf{end}\, x \\ = & \{\langle \mathsf{end} - \mathsf{var}\ \text{skip}\rangle(3.19.7)\ \text{and}\ \langle \mathsf{end}\ \text{change scope}\rangle(3.19.2)\} \\ & \mathsf{var}\, x;\ x := y;\ \mathsf{end}\, y;\ p;\ \mathsf{var}\, y;\ y := x;\ \mathsf{end}\, x \\ = & \{\text{Lemma 5.3(Data refinement as substitution)}\} \\ & LHS \end{array}$$

■

It should be clear that the conditions on the above theorems impose no practical limitations; they can be automatically satisfied by using locally declared variables as arguments.

5.5 Parameterised Procedures

As a consequence of the results in the previous section, we can treat a parameterised procedure in the same way as a parameterless one. The same notation is used

$$\textit{proc}\ X \mathrel{\widehat{=}} (\textit{par}\ x \bullet p) \bullet q$$

except that now the body is a parameterised program (by any of the mechanisms discussed above) and all occurrences of X in q are of the form $X(t)$, for some appropriate actual parameter t.

This allows the meaning of a parameterised procedure to be given by the copy rule, as before. Therefore the above is equivalent to

$$q[X \leftarrow (\textit{par}\ x \bullet p)]$$

This textual substitution could be used to eliminate parameterised procedures by macro-expansion. Sharing the code of the procedure body can be achieved by transforming a parameterised procedure into a parameterless one, and then using the reduction theorem for parameterless procedures (Theorem 5.2). Another source of optimisation is the sharing of local variables used to eliminate parameterisation. This is established by the following lemma.

Lemma 5.4 (Sharing of local variables)
If x is not free in p, then

$$p[X \leftarrow (\mathsf{var}\,x;\ q;\ \mathsf{end}\,x)] \sqsubseteq \mathsf{var}\,x;\ p[X \leftarrow q];\ \mathsf{end}\,x$$

Proof: By structural induction, using the distribution laws of var and end with the other program constructs. ■

Then we have the following elimination rule for parameterised procedures. (We use value-result parameters as illustration; the rules for the other parameterisation mechanisms are similar.)

Theorem 5.7 (Value-result parameters of procedures)
If x is not free in q, then

$$\mathsf{proc}\,X \mathrel{\widehat{=}} (\mathsf{valres}\,x \bullet p) \bullet q \quad \sqsubseteq \quad \mathsf{dec}\ x \bullet (\mathsf{proc}\ X \mathrel{\widehat{=}} p \bullet q[X \leftarrow (\mathsf{valres}'\,x \bullet X)])$$

where

$$(\mathsf{valres}'x \bullet p)(y) \stackrel{def}{=} x := y;\ p;\ y := x$$

Proof:
Let q' be such that

$$(1)\ q = q'[Y_1, \ldots, Yn \leftarrow X(y_1), \ldots, X(y_n)]$$

and q' contains no calls of procedure X. Clearly, for all q it is always possible to find a q' that satisfies the above equation. Then we have:

$$
\begin{array}{ll}
& LHS \\
= & \{\text{Definition of procedures and } (1)\} \\
& q'[Y_1, \ldots, Yn \leftarrow (\textsf{valres}\, x \bullet p)(y_1), \ldots, (\textsf{valres}\, x \bullet p)(y_n)] \\
= & \{\text{Theorem 5.3(Elimination of value} - \text{result parameters)}\} \\
& q'[Y_1, \ldots, Yn \leftarrow (\textsf{var}\, x;\ x := y_1;\ p;\ y_1 := x;\ \textsf{end}\, x), \ldots, \\
& \qquad (\textsf{var}\, x;\ x := y_n;\ p;\ y_n := x;\ \textsf{end}\, x)] \\
\sqsubseteq & \{\text{Lemma 5.4(Sharing of local variables) and} \\
& (\textsf{dec} - (\textsf{var}, \textsf{end})\ \text{conversion})(3.19.10)\} \\
& \textsf{dec}\, x \bullet q'[Y_1, \ldots, Yn \leftarrow (x := y_1;\ p;\ y_1 := x), \ldots, (x := y_n;\ p;\ y_n := x)] \\
= & \{\text{property of substitution and } (1)\} \\
& \textsf{dec}\, x \bullet q[X \leftarrow (\textsf{valres}'\, x \bullet p)] \\
= & \{\text{Definition of procedures}\} \\
& RHS
\end{array}
$$

■

5.6 Parameterised Recursion

The notation for a parameterised recursive program X is the same as before, except that now its body is a parameterised program and all the recursive calls are of the form $X(t)$, for some appropriate actual parameter t. The meaning of a parameterised recursive program is given by

Definition 5.7 (Parameterised recursion)

$$\mu X \bullet (par\, x \bullet p) \ \stackrel{def}{=}\ par\, x \bullet (\mu X \bullet p[X \leftarrow (par\, x \bullet X)])$$

■

The initial value of the formal parameter x is given by a non-recursive call which can be dealt with in the same way as procedure calls. The local declarations created during the elimination of the parameterised recursive calls are implemented by a stack. We have the following elimination rule for value-result parameters of recursive programs.

Theorem 5.8 (Value-result parameters of recursion)
If $x \neq z$ and $\bar{x}$ is not free in p then

$$(\mu X \bullet (\textsf{valres}\, x \bullet p))(z) \ \sqsubseteq\ \textsf{dec}\, x, \bar{x} \bullet (x := z;\ \mu X \bullet p[X \leftarrow (\textsf{valres}'\, x \bullet X)];\ z := x)$$

where

$$(\mathsf{valres}'x \bullet p)(y) \stackrel{def}{=} \mathsf{push}(x, \bar{x});\ x := y;\ p;\ y := x;\ \mathsf{pop}(x, \bar{x})$$

Proof: Let p' be such that

$$(1)\ p\ =\ p'[Y \leftarrow X(y)]$$

and p' contains no calls of procedure X. (For conciseness we will assume that all the calls are parameterised by y. Arbitrary calls can be treated as in the previous theorem.) Then we have:

$$
\begin{array}{ll}
 & LHS \\
= & \{\text{Theorem 5.3(Elimination of value} - \text{result parameters) and (1)}\} \\
 & \mathsf{var}\, x;\ x := z;\ \mu X \bullet p'[Y \leftarrow (\mathsf{var}\, x;\ x := y;\ X;\ y := x;\ \mathsf{end}\, x)];\ z := x;\ \mathsf{end}\, x \\
= & \{(\mathsf{dec} - (\mathsf{var}, \mathsf{end})\ \text{conversion})(3.19.10)\} \\
 & \mathsf{dec}\, x \bullet (x := z;\ \mu X \bullet p'[Y \leftarrow (\mathsf{var}\, x;\ x := y;\ X;\ y := x;\ \mathsf{end}\, x)];\ z := x) \\
\sqsubseteq & \{((\mathsf{var}, \mathsf{end}) - (\mathsf{push}, \mathsf{pop})\ \text{conversion})(5.1.3)\} \\
 & RHS
\end{array}
$$

■

Recall that var and end operate on an implicit stack which can be made explicit by using push and pop. This fact was used in the last step of the above proof.

5.7 Discussion

The main advantage of handling procedures, recursion and parameters separately is that the overall task becomes relatively simple and modular. Furthermore this imposes no practical limitation, since more complex structures can be defined by combining the basic ones. For example, we have already illustrated how parameterised procedures, parameterised recursion and multiple parameterisation can be achieved. Recursive procedures (whether parameterised or not) can also be easily introduced. Consider the procedure declaration

$$\mathsf{proc}\ X \mathrel{\widehat{=}} p \bullet q$$

where p and q may respectively contain recursive and non-recursive calls of X, and p may be a parameterised program. This can be defined as the non-recursive procedure

$$\mathsf{proc}\ X \mathrel{\widehat{=}} (\mu X \bullet p) \bullet q$$

whose body is a recursive program.

One aspect not fully addressed is how the reduction theorems given in this chapter can be used as compilation rules. Notice that the theorems about parameterisation establish

that it can be completely eliminated by transformations at the source level. The implementation of procedures and recursion requires a more powerful target machine than the one defined in the previous chapter. Basically, new instructions are necessary to execute the call and return sequences. These instructions can be defined by the new patterns which appear in the normal form of procedures and recursion. For procedures, we have shown that the call and return sequences can be implemented by allocating temporary variables; for recursion, new instructions to model the basic stack operations are required.

We have allocated a separate stack to implement each recursive program. This not only simplifies the proof of the elimination of recursion, but will be essential if we decide to extend our source language with a parallel operator. In the present case of our sequential language (and for non-nested recursion), the local stacks can be replaced by a global stack. This is systematically achieved by the elimination of sequential composition

$$\begin{array}{ll} & v,\bar{v}:[a, b_1 \rightarrow p, c_0];\ v,\bar{v}:[c_0, b_2 \rightarrow q, c] \\ \sqsubseteq & v,\bar{v}:[a, \begin{pmatrix} & b_1 \rightarrow p \\ \Box & b_2 \rightarrow q \end{pmatrix}, c] \end{array}$$

However, in the case of nested recursion, we still need a separate stack for each level of nesting (where the declarations at the same level can be shared in the way shown above). An implementation could use pointers to link the stacks in a chain representing the nesting; this technique is known as *cactus stacks* [37]. A single stack implementation is discussed in the final chapter, where it is suggested as a topic for future work. For a realistic implementation, we will need a more concrete representation of these stacks. In particular, as the storage available in any machine is finite, it is necessary to impose a limit on the size of the stacks.

The compilation of programs now including procedures and recursion (possibly with parameters) should proceed as follows. As for the simple source language considered in the previous chapter, the first step is the simplification of expressions. Recall that one result of this phase of compilation is to extend the scope of local variables as much as possible, so that they can be implemented in the same way as global variables. However, it is not possible in general to move a local declaration out of a recursive program; rather, as explained before, it is implemented by a stack. Also recall that for the simplified language, the order of the remaining two phases (control elimination and data refinement) is not relevant. However, the reduction theorems for procedures and parameters introduce new local variables for which storage will have to be allocated. Therefore once the expressions are simplified, it is more sensible to carry out control elimination first and leave data refinement for the very last step.

Chapter 6

Machine Support

> [...] the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing is unsound and clumsy because people who do it do not have a clear understanding of the fundamental principles underlying their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. [...] this separation cannot happen.
>
> — C. Strachey

Because of the algebraic nature of this approach to compilation, it is possible to use a term rewriting system to check the correctness of the reductions. Furthermore, the reduction theorems can be taken as rewrite rules to carry out the compilation task.

The purpose of this chapter is to show how this can be achieved using the OBJ3 system [31]. There are three main activities involved:

- The formalisation (specification) of concepts such as the reasoning language, its algebraic laws, the normal form, the target machine, and so on, as a collection of *theories* in OBJ3.
- The verification of the related theorems.
- Compiling with theorems: the reduction theorems are collected together and used as a compiler prototype.

We adopt the well established algebraic approach to specifications. For a formal presentation of the concepts involved we refer the reader to, for example, [24, 27, 75]. Here we describe some of the main concepts informally.

Broadly, we will consider a specification as consisting of three parts:

- a *signature*, which comprises a family of sorts (names for carrier sets) and operator symbols (names for operators) with given functionalities;

- a set of *axioms*, given by equations (and inequations) relating the operators of the signature, and
- a set of theorems, which can be deduced from the axioms.

The idea is to construct specifications incrementally. We start with a specification whose signature is an abstract syntax for the reasoning language, and the axioms are the basic algebraic laws. The set of theorems is initially empty; it is gradually built by proving that new algebraic laws are logical consequences of the basic ones. This is one way in which a specification can be *extended*. We then proceed extending the specification in a more general way, adding sorts, operators and axioms (and proving more theorems) to describe the remaining concepts.

One important aspect not addressed by this strategy is *consistency*. In particular, how to ensure that we started with a consistent set of algebraic laws? As discussed in Chapter 3, we consider this a separate task. A reasonable alternative is to formalise a given model, such as predicate transformers, and then derive the basic laws, one by one. This is illustrated in [6], using the HOL system [34].

For simplicity, we deal with the material presented in chapters 3 and 4; the normal form theorems for procedures and recursion are not considered.

Our main concern here is the overall structure of the specification and proofs, rather than a detailed description of all the steps and technicalities of the verification of the theorems in OBJ3. The specification of most concepts used (together with the complete verification of some of the main theorems) is given in Appendix C.

6.1 OBJ3

OBJ3 is the latest in a series of OBJ systems, all based upon first order equational logic. A detailed description of OBJ3 can be found in [31]. In this section we give a brief overview of the system (based on Release 2) and discuss how it supports the strategy presented above. More specific features are explained when necessary.

OBJ3 is a general-purpose declarative language, especially useful for specification and prototyping. A specification in OBJ3 is a collection of modules of two kinds: *theories* and *objects*. A theory has a *loose* semantics, in the sense that it defines a variety of models. An object has a *tight* or *standard* semantics; it defines, up to isomorphism, a specific model—its initial algebra [30]. For example, we use objects to define abstract data types such as lists and maps; the reasoning language and its algebraic laws are described as a theory, since we are not concerned with a particular model.

A module (an object or a theory) is the unit of a specification. It comprises a signature and a set of (possibly conditional) equations—the axioms. The equations are regarded as rewrite rules and computation is accomplished by term rewriting, in the usual way.

An elaborate notation for defining signatures is provided. As OBJ3 is based upon *order sorted algebra* [28], it provides a notion of subsorts which is extremely convenient in

practice. For example, by declaring the sort representing variables as a subsort of the one representing expressions, we can use variables both on the left and on the right-hand sides of an assignment statement; no conversion function is needed to turn a variable into an expression.

A general *mixfix* syntax can be used to define operators. In our case, we use names for operators which coincide with the LaTeX [51] representation of the desirable mathematical symbols. This allowed us to use the same notational conventions introduced in earlier chapters, with the hope that this will make the encoding in OBJ3 more comprehensible.

Moreover, operators may have attributes describing useful properties such as associativity, commutativity and identity. This makes it possible to document the main properties of a given operator at the declaration level. As a consequence, the number of equations that need to be input by the user is considerably reduced in some cases. Most importantly, OBJ3 provides rewriting modulo these attributes.

Modules may be parameterised by theories which define the structure and properties required of an actual parameter for meaningful instantiation. The instantiation of a generic module requires a *view*—a mapping from the entities in the requirement theory to the corresponding ones in the actual parameter (module). As a simple example, an object to describe lists of arbitrary elements, say LIST, should be parameterised by a theory which requires the argument module to have (at least) one sort. In this simple case, we may instantiate LIST with a sort, since the view is obvious. Thus, assuming that Var and Exp are sorts representing variables and expressions, we can create the instances LIST[Var] and LIST[Exp]. A more interesting example is the parameterisation of the module describing the reasoning language by a theory of expressions; this is further discussed in the next section.

Apart from mechanisms for defining generic modules and instantiating them, OBJ3 provides means for modules to import other modules and for combining modules. For example, A + B creates a new module which combines the signature and the equations of A and B. This support is essential for our incremental strategy to specifications.

OBJ3 can also be used as a theorem prover. In particular, computation is accomplished by term rewriting which is a widely accepted method of deduction. If the rewrite rules of the application theory are *confluent* (or *Church-Rosser*) and *terminating*, the exhaustive application of the rules works as a decision procedure for the theory: the equivalence of two expressions can always be determined by reducing each one to a *normal form*; the equivalence holds if the two normal forms are the same (syntactically speaking). This proof mode is normally called *automatic*, and is supported by OBJ3.

In our case, this proof mode is useful to discharge side conditions about non-freeness and to perform substitution. Furthermore, once the reduction theorems are proved, this mode can be used to carry out compilation automatically, since these reduction theorems are complete in that they allow the reduction of an arbitrary source program to a normal form; this will be illustrated later in this chapter. Unfortunately, the proof of these theorems cannot be carried out automatically, since there is no decision procedure for our algebraic system in general (that is, including all the algebraic laws listed in Chapter 3). As a

consequence, automatic term rewriting may fail to decide if two program fragments are equivalent (the process may even fail to terminate). Therefore there is the need for a mechanism for applying rules in a controlled way, as we have done in the manual proofs presented in earlier chapters. OBJ3 supports the *step by step* application of rewrite rules either forwards (from left to right) or backwards (from right to left).

Proofs in OBJ3 are confined to these two modes. There is no built-in support for proof by induction or by case analysis of any other kind. But the most serious limitation concerning our application is the lack of a mechanism to deal with inequational rewriting. We encode inequations as equations whose right-hand sides refine the corresponding left-hand sides, but as will be discussed later this is far from satisfactory.

Other limitations are related to proof management. OBJ3 does not distinguish between units of information such as an axiom, a conjecture and a theorem. Although specifications and proofs can be stored in files, the management must be done by the user.

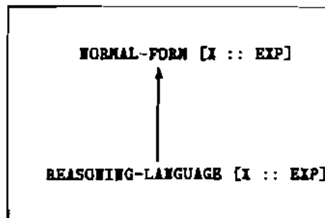
6.2 Structure of the Specification

We use the module facilities of OBJ3 to structure the specification in such a way that each concept is described by a separate module. Figure 6.1 shows the hierarchy of the main modules, where an arrow from module `A` to module `B` indicates that `B` imports `A`. First we explain part (a) of the figure. The module which describes the reasoning language is generic with respect to the expression language. This can be elegantly described in OBJ3 by defining a theory of expressions and parameterising the module `REASONING-LANGUAGE` with this theory. Then the commands (and their algebraic laws) can be instantiated with different expression languages, according to the particular needs.

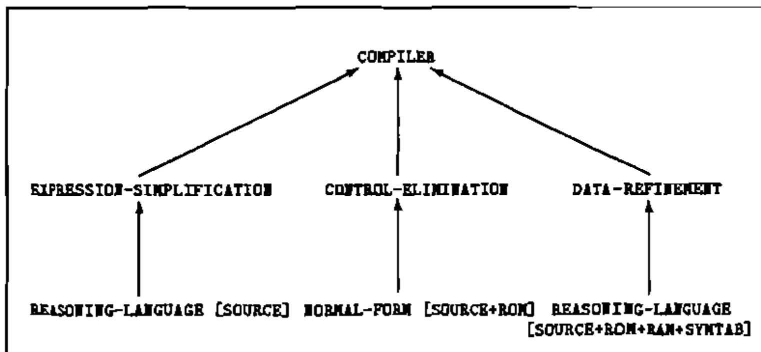
The module describing the normal form is equally independent of a particular expression language (and in particular of a way of encoding control state). Clearly, it needs to import the previous module so that the normal form operator can be defined and the associated reduction theorems can be proved.

Part (b) of the figure presents the structure of the specification of our simple compiler, where each phase of compilation is described by a separate module. It also shows how the previous modules are instantiated for this application. For example, the module describing the simplification of expressions is concerned with the notation of expressions of the source language. To reason about this phase of compilation, we instantiate the module `REASONING-LANGUAGE` with this particular kind of expressions. Similarly, the module concerned with control elimination instantiates `NORMAL-FORM` with the combination (+) of two kinds of expressions: the expressions of the source language and the ones used to represent addresses in the memory `ROM` of our target machine.

The module describing the data refinement phase instantiates `REASONING-LANGUAGE` with a complex expression language formed from the two kinds discussed above and (map) expressions to represent the memory `RAM` and the symbol table. The compiler is formed from the modules describing the three phases of compilation.



(a) Generic Modules



(b) Structure of the Compiler

Figure 6.1: Structure of the Specification.

The internal structure of these modules is described in the remaining sections, and further details are given in Appendix C.

6.3 The Reasoning Language

In the previous chapters we dealt with expressions in an informal way; new operators were introduced as we needed them. For example, to describe the data refinement phase of compilation we used map expressions to model the symbol table and the store of the target machine; in the control elimination phase we used arithmetic expressions to encode control state. This was possible because the algebraic laws are independent of a particular expression language. As mentioned above, this can be captured in OBJ3 by defining a theory of expressions and parameterising the module describing the reasoning languages with this theory.

The theory of expressions must include the boolean values with the usual operators. This is necessary to enable us to describe the algebraic laws of conditional commands. The boolean expressions (conditions) are described by the following module:

```
obj COND is

  sorts CondVar CondExp .
  subsorts CondVar < CondExp .

  op true  : -> CondExp .
  op false : -> CondExp .
  op _∨_  : CondExp CondExp -> CondExp [assoc comm idem id: false] .
  op _∧_  : CondExp CondExp -> CondExp [assoc comm idem id: true] .
  op ¬_   : CondExp -> CondExp .

  var a : CondExp .

  eq true ∨ a = true .
  eq false ∧ a = false .
  eq ¬true = false .
  eq ¬false = true .

endo
```

The subsort relation states that boolean expressions may contain variables (elements of sort CondVar). The symbol _ which appears in the declaration of the operators determines the position of their arguments. The attributes of a given operator are given inside square brackets. For example, ∨ is associative, commutative, idempotent and has identity **false**. The additional properties of the operators are described by equations.

The module describing our theory of expressions declares sorts Var and Exp to represent

arbitrary variables and expressions. The requirement that the expression language must include boolean expressions is captured by subsort relations, as shown in the following.

```
th EXP is
  protecting COND .
  sorts Var Exp .
  subsorts Var < Exp .
  subsorts CondVar < Var .
  subsorts CondExp < Exp .
endth
```

The module COND is imported using protecting. This mode of importation is used to state that EXP does not add or identify elements of sorts from COND.

The above theory is then used to parameterise the module which describes the reasoning language. This means that any actual parameter used for instantiation must be a *model* of the above theory. Informally, any particular expression language must be equipped with (at least) what is stated by the theory EXP. A partial description of the module describing the operators of the reasoning language is given below.

```
th REASONING-LANGUAGE[X :: EXP] is
  sorts Prog ProgId .
  subsorts ProgId < Prog .

  define ListVar is LIST[Var] .
  define ListExp is LIST[Exp] .

*** Source language
  op skip : -> Prog .
  op-as _:=_ : ListVar ListExp -> Prog
        for x := e if (len x == len e) and (disj x) [prec 52] .
  op _;_ : Prog Prog -> Prog [assoc prec 56] .
  op _⊓_ : Prog Prog -> Prog [assoc comm idem id: ⊤ prec 57] .
  op _◁ _ ▷_ : Prog CondExp Prog -> Prog [prec 58] .
  op _*_ : CondExp Prog -> Prog [prec 54] .
  op dec _ •_ : ListVar Prog -> Prog [prec 60] .

*** Additional specification features
  op ⊥ : -> Prog .
  op ⊤ : -> Prog .
  op _⊑_ : Prog Prog -> Bool [prec 70] .
  op μ _ •_ : ProgId Prog -> Prog [prec 59] .
     ...       ...
endth
```

The sort ProgId declared above stands for program identifiers, used to name recursive programs. Furthermore, by declaring ProgId as a subsort of Prog we can use program identifiers as call commands. Following the sort declarations, the define clause is used to instantiate of the module LIST (omitted here) to create lists of variables and lists of expressions. These are used, for instance, in the declaration of the multiple assignment command, which is a partial operator[1] only defined for equal-length lists of variables and expressions. In addition, the list of variables must be disjoint; no variable may appear more than once in the list. The declaration of most operators includes an attribute which determines their precedence. The lower the precedence, the tighter the operator binds.

Some auxiliary operators are needed to implement the concepts of non-freeness, non-occurrence and substitution. The non-occurrence operator is used to state that a given identifier does not occur in a program, not even bound. Their declaration is given below.

```
op _\_    : ListVar Prog -> Bool [memo] .          *** non-freeness
op _\\_    : ListVar Prog -> Bool [memo] .         *** non-occurrence
op-as _[_<-_] : Prog ListVar ListExp  -> Prog     *** substitution
   for p[x <- e] if (len x == len e) and (disj x) [memo] .
```

Notice that the last operator allows multiple substitution, and therefore it has the same precondition as that of the assignment operator. Overloaded versions of the above operators (omitted here) deal with expressions.

Especially when using the rewrite rules to carry out compilation, these operators are applied to the same arguments, over and over again. The number of rewrites is substantially reduced by giving them the memo attribute which causes the results of evaluating a term headed by any of these operators to be saved; thus the evaluation is not repeated if that term appears again.

These operators are defined in the usual way. The complete definition requires a large number of equations, one to deal with each operator of the language, and therefore is omitted here. However, the implementation of safe substitution as a set of rewrite rules deserves some attention, due to the need to rename local variables to avoid variable capture. This is actually an instance of the more general problem of creating fresh identifiers using a formalism with a stateless semantics[2]. A possible (although cumbersome) solution is to pass a (virtually) infinite list of unique identifiers as parameter. As we never use substitution in a context which requires renaming, we solve the problem by using the following conditional equation

```
cq (dec x • p) [y <- f]  =  (dec x • (p[y <- f]))
if x \\ (y,f) .
```

where the condition that x does not occur in the list (y,f) prevents variable capture.

[1]In Release 2 of OBJ3 partiality is only syntactic. Its semantics is being implemented [77].

[2]Actually, OBJ3 allows *built-in* equations which provide direct access to Lisp (and therefore the possibility of modelling a global state), but we have avoided the use of this feature.

Another observation regarding these auxiliary operators is that they must be defined in a context separate from that of the algebraic laws. Their equations entail syntactic transformations, whereas the laws express semantic properties. While a formal distinction between the syntactic and the semantic natures of programs is necessary in principle, it would require the explicit definition of some kind of semantic function which would make the mechanisation extremely laborious. We avoid a formal distinction between these two views of programs by defining the equations of the auxiliary operators in a separate module. As OBJ3 allows one to specify the context for a given reduction, we can ensure that syntactic and semantic transformations are never intermixed.

The algebraic laws are described as labelled equations. The labels are used in the verification phase as references to the equations. We use the same name conventions introduced earlier. For example, Law $\langle := \text{ skip}\rangle(3.15.1)$ is coded as

```
[:=~skip]   eq   (x := x) = skip .
```

The codification of most of the laws is straightforward. However, there is a drawback concerning the laws which are inequalities. Although, for example, $b_\perp \sqsubseteq$ skip can be precisely described by the equation

```
eq   (b⊥ ⊑ skip) = true .
```

this encoding is not convenient for use in proofs. This equation allows us to rewrite the left-hand side with `true`, or vice-versa; but we are concerned with reducing nondeterminism (by rewriting $b_\perp$ to skip) or, conversely, increasing nondeterminism, by rewriting in the opposite direction. The way we overcome this problem is coding inequations as equations whose left-hand sides are less deterministic than the corresponding right-hand sides. We document this fact by adding the ordering relation as an additional label to the equations. The above then becomes

```
[b⊥~skip ⊑]   eq   b⊥ = skip .
```

But this is clearly unsatisfactory, as OBJ3 treats it as an ordinary equation; we have the obligation to check (with the aid of the annotations) if an application of a rule makes sense in a given context. For example, in a refinement process, a left to right application of the above rule is valid, but an application in the opposite direction is obviously invalid. An appropriate solution is to build support for inequational rewriting into the system. In the final section of this chapter we briefly discuss a system based on OBJ3 which accepts as input an OBJ3 specification and treats the rules annotated with the ordering relation as inequations.

The laws of recursion also deserve some attention. OBJ3 is based on first order logic, and therefore we cannot quantify over functions, as implicitly done in the fixed point and least fixed point laws. However, this can be easily overcome using substitution:

```
[μ~fp]        eq   (μ X • p)  =  p[X <- (μ X • p)] .
[μ~lfp ⇐]     eq   (μ X • p) ⊑ q  =  (p[X <- q]) ⊑ q .
```

Note that the second equation should have been coded as an implication, with its left-hand side implied by its right-hand side.

It is convenient to define instances of the above laws to deal with iteration commands, as in this particular case we can get rid of the substitution operator.

```
[*~fp]       eq  b * p  =  (p ; b * p ◁ b ▷ skip) .
[*~lfp ⇐]   eq  (b * p) ⊑ q  =  (p ; q ◁ b ▷ skip) ⊑ q .
```

These are easily derived from the definition of iteration and the laws of recursion.

6.3.1 An Example of a Proof

To illustrate how proofs are carried out in OBJ3, we chose a simple example which highlights both some positive points and some limitations. Other proofs are presented in Appendix C.

The example used here is the first part ($RHS \sqsubseteq LHS$) of the proof of Law 3.17.6:

$$(b * p);\ q \ = \ \mu X \bullet (p;\ X \lhd b \rhd q)$$

OBJ3 supports the step by step style of proof that we have used in the manual proofs. First we define constants LHS and RHS to stand for the respective sides of the above equation. Then we start the proof process with

```
start  (b * p) ; q ⊑ LHS .
```

which is equivalent to `true` (by the reflexivity of $\sqsubseteq$). The proof strategy is to gradually transform the term `(b * p) ; q` into RHS by requesting OBJ3 to *apply* equations which encode the appropriate laws. For example, the following is a request to apply the fixed point law

```
OBJ> apply .*~fp within term .
```

where `within term` is used to apply a given equation to all possible subterms of the term in focus. In our case there is only one match for the left-hand side of the fixed point equation, and the system replies with the expected result:

```
result Bool: (p ; b * p ◁ b ▷ skip) ; q ⊑ LHS
```

In a similar way, we apply equations to move q inside the conditional and to eliminate skip (and then we rewrite `b * p ; q` to LHS), resulting in

```
result Bool: (p ; LHS ◁ b ▷ q) ⊑ LHS
```

which suggests the backward (right to left) application of the least fixed point equation (see the equation with label μ~lfp in the previous section). Note, however, that the right-hand side of that equation mentions the substitution operator explicitly, and therefore cannot be matched by the above term. The desired form can be achieved by using the equations of substitution. For the moment, we add an equation which allows us to perform the desired transformation:

```
[subst1] eq (p ; LHS ◁ b ▷ q) = (p ; X ◁ b ▷ q)[X <- LHS] .
```

Then we have

```
OBJ> apply .subst1 within term .
result Bool: (p ; X ◁ b ▷ q)[X <- LHS] ⊑ LHS
```

which enables us to apply the least fixed point equation:

```
OBJ> apply -.μ~lfp with within term .
result Bool: μ X • (p ; X ◁ b ▷ q) ⊑ LHS
```

where the minus sign preceding a label requests backward application of the corresponding equation. The final result follows directly from the definition of RHS.

But we still need to discharge the proof obligation introduced by the added equation subst1. As the equations defining substitution are confluent and terminating, we can prove this equation automatically, rather than step by step. In OBJ3 this is achieved by

```
OBJ> select SUBST .
OBJ> reduce (p ; LHS ◁ b ▷ q) == (p ; X ◁ b ▷ q)[X <- LHS] .
rewrites: 4
result Bool: true
```

where the command select is used to specify the context in which the reduction is carried out. The module SUBST contains the relevant equations of substitution. Recall that we need to collect these equations in a separate module, since a reduction involving substitution entails syntactic transformations; the algebraic laws express semantic properties and their use must be disallowed in such a proof. A similar technique is employed to reduce side conditions about non-freeness.

This simple reduction has actually uncovered one hidden assumption in the manual proof: the law being verified is true only if X is not free in p or q. So, in order to prove the above equation, we must add this as an assumption:

```
eq X \ p = true .
eq X \ q = true .
```

Although the mechanical proof closely corresponds to its manual version, some limitations can be observed. One has already been discussed earlier and relates to reasoning with inequations. Note that we have no means to tell the system that the term resulting from the application of least fixed point is implied by (rather than equivalent to) the term that precedes the application. This is only informally documented in the definition of the corresponding equation.

First order logic is sufficient to express most of the algebraic system we use; however, higher-order matching would enable a more convenient description (and application) of the laws of (least) fixed point, without mentioning substitution explicitly. As illustrated above, the use of substitution requires the user to provide the matching; this turns out to be cumbersome if the laws are repeatedly used. However, the problem was made easier by defining instances of these laws to deal with iteration, since they do not mention the substitution operator.

Concerning proof management, no support is provided. For example, there is no facility to deal with assumptions; to be used as rewrite rules, they have to be added by the user as ordinary equations, as shown above. Also, it would be useful if the theorem we have just proved were automatically added to the module in question, and made available in later proofs. In OBJ3 this has to be carried out by the user.

6.4 The Normal Form

The module describing the normal form extends that describing the reasoning language with a new operator and its defining equation:

```
op-as (_:[_, _->_, _]) : ListVar Cond Cond Prog Cond -> Prog
      for v :[a, b -> p, c] if pwd(b,c) [prec 50] .

[nf~def] eq  v :[a, b -> p, c]  =  dec v • v :∈ a ; b * p ; c⊥ .
```

where the precondition states the required disjointness of the conditions b and c.

The reduction theorems can then be proved from the above definition and the equations describing the algebraic laws, in much the same way as illustrated in the previous section. Then they can be added to the module as ordinary equations. For example, the reduction of sequential composition is captured by the equation

```
[T:sequential~composition ⊑]
  cq  v :[a, b1 -> p, co]  ;  v :[co, b2 -> q, c]  =
      v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]
  if pwd(b1,b2,c) .
```

which is true only under the condition that `b1`, `b2` and `c` are pairwise disjoint[3]. Recall that these disjointness conditions were implicit in the manual proofs.

6.5 A Compiler Prototype

Here we formalise the design of the compiler presented in Chapter 4. The components of the target machine are gradually introduced by the modules describing the phases of compilation: simplification of expressions, control elimination and data refinement; these are the concern of the first three subsections. Apart from these phases, it is necessary an additional step to replace the assignment statements (used as patterns to define instructions) with the corresponding instruction names. This is discussed in Subsection 6.5.4. In the last subsection we illustrate how compilation is carried out using equations which encode the reduction theorems. The complete description of the modules concerned with the compilation phases (and the verification of some of the related theorems) is given in Appendix C.

6.5.1 Simplification of Expressions

Rather than defining a particular notation of expressions in our source language, we want to illustrate how to simplify expressions involving arbitrary binary and unary operators:

```
sorts SourceVar SourceExp .
subsorts SourceVar < SourceExp .

op uop_  : SourceExp -> SourceExp .
op _bop_ : SourceExp SourceExp -> SourceExp .
```

The only machine component relevant to this phase is the general purpose register. It is represented here by the following constant

```
op A : -> SourceVar .
```

which must be of the same sort as an ordinary source variable because, during the process of simplification, A is assigned expressions of the source language.

The theorems related to this phase can be verified (and then introduced as equations) in the way already explained. But one aspect to be addressed is the creation of fresh local variables that may be required during the simplification of expressions. Like the implementation of safe substitution (mentioned before) this is an instance of the more general problem of creating fresh identifiers using a formalism with a stateless semantics.

[3] When the semantic effect of partial operators is implemented, this kind of condition will be automatically inserted by the system, as it can be deduced from the declaration of the operators.

One approach is to generate a fresh identifier every time a temporary variable is needed (which is confined to the equation concerned with the simplification of binary operators). While this obviously works, it does not optimise the use of space. An optimisation is to create distinct identifiers for variables in nested blocks, but identify variables (with the same name) which appears in disjoint blocks. This can be accomplished with

```
let n = depth(e bop f) in
  cq (A := e bop f) = dec tn • A:=f ; tn:=A ; A:=e ; A:=A bop tn
  if (A,tn) \\ (e bop f) .
```

where we use the depth[4] of a given expression as the basis to generate fresh identifiers. The term t_n comprises an invisible operator which from an identifier t and a natural number n generates a new identifier t_n. The `let` clause above was used to improve readability; in OBJ3 it is available only for defining constants.

Observe that the local variables of the nested blocks generated by the simplification of a given expression are guaranteed to be distinct, as the depth of the expression decreases during the simplification process. However, the same identifiers may be generated to simplify another expression; in particular, we always use the same *base* identifier t to ensure the maximum reuse of temporaries.

Recall that an optimisation of the above rule is possible when the expression f is a variable, in which case the allocation of temporary storage is unnecessary:

```
cq (A := e bop x) = A:=e ; A:=A bop x
if A \\ (e bop x) .
```

As variables are also expressions, any assignment which matches the left-hand side of this rule also matches that of the previous rule. Of course, we always want the optimising rule to be used in such cases. But OBJ3 does not provide means to express rule priorities (one cannot even rely on the fact that the system will attempt to apply rules in the order in which they are presented). To ensure application of the optimising rule, we need to add an extra condition to the previous rule, say `is-not-var(f)`, where `is-not-var` is a boolean function which yields `true` if and only if the expression to which it is applied is not a variable.

6.5.2 Control Elimination

The steps to formalise this phase of compilation are very similar to those of the previous phase. Expressions are further extended to include natural numbers which are used to represent addresses in the memory ROM. Furthermore, the program counter is declared as a special (program) variable to which expressions representing ROM addresses may be assigned. These are specified as follows:

[4] From a tree representation of an expression, we define its depth to be the number of nodes in the longest path of the tree.

```
sorts RomAddrVar RomAddrExp .
subsorts RomAddrVar < RomAddrExp .
subsorts Nat < RomAddrExp .

op P : -> RomAddrVar .
```

where Nat is built-in to OBJ3; the usual numerical representation is available, thus we do not need to write large numbers in terms of a successor function.

The theorems for control elimination are easily verified by instantiating the normal form theorems. Additional transformations are required in some cases, but they do not illustrate any new aspect.

A problem similar to the generation of fresh identifiers is the allocation of distinct addresses for the machine instructions yielded by the compilation process. The solution we have adopted is to preprocess the source program to tag each construct with a distinct natural number representing an address in ROM. Then the reduction theorems are encoded as, for example:

```
cq {s}(x:=e) = P:[s, (P=s) -> (x,P := e,P+1), (s+1)] if P \\ e .
```

where s is the address allocated to place the instruction generated by x := e. We carry out the tagging after simplifying expressions, since each simple assignment will give rise to a single instruction.

It is worth stressing that tagging is merely a syntactic annotation to ensure the disjointness of the addresses allocated for the machine instructions. It has no semantical effect; more precisely, {s} p = p, for all programs p.

6.5.3 Data Refinement

This phase entails more sophisticated components such as the symbol table and the store for variables; they are represented as maps, rather than as single variables:

```
define SymTab is MAP[SourceVar,RamAddr] .
define Ram is MAP[RamAddr,SourceExp] .

op M : -> Ram .
var Ψ : SymTab .
```

This formalises the fact that a symbol table is a map from identifiers to addresses of locations in Ram, which is itself a map from addresses to expressions denoting the corresponding values. In practice we use natural numbers to represent addresses; this is possible by making Nat a subsort of RamAddr.

As the other machine components, M was introduced as a constant of the appropriate sort. However, the symbol table may change from program to program, and is therefore declared as a variable.

The simulation used to carry out data refinement can then be defined. Below we give the declaration and the definition of its first component:

```
op-as ^_ : SymTab ListSourceVar -> Prog
     for Ψ̂w if elts(w) == (dom Ψ) and disj(w) .

eq Ψ̂w = var w ; w := M[Ψ[w]] ; end M .
```

where the precondition requires that each variable in the domain of Ψ occurs exactly once in the list of global variables w. Note the similarity between the above equation and the original definition of $\widehat{\Psi}_w$ given in Section 4.6. In particular, we have implemented list application (among other map operations) to allow a straightforward encoding.

The distributivity properties of the simulation $\widehat{\Psi}_w$ can be verified in the usual way. Perhaps the most interesting is the one which allocates space for the local variables:

```
cq Ψ̂w(dec v,P,A • p) = dec P,A • Φ̂(w,v)(p)
if disj(v,w) and disj(Φ[v],Ψ[w]) .
```

where $\Phi = \Psi \cup \{$v $\mapsto$(base + len(w) + 1 .. base + len(w,v))$\}$.

It is required that the global variables w are distinct from the local variables v, and that the new addresses Φ[v] are different from the ones already used, Ψ[w]. We have already discussed how to satisfy the first condition. The other one can be easily satisfied by allocating for the global variables the addresses base + 1 .. base + len(w), where base is an arbitrary natural number; the definition of Φ then guarantees that the addresses allocated for the local variables are distinct from those.

The complete verification of the above theorem is given in Appendix C.

6.5.4 Machine Instructions

The machine instructions are defined as assignments that update the machine state. Therefore the instructions should also be regarded as elements of the sort Prog. However, to make it clear that we are introducing a new concept, we declare a subsort of Prog whose elements are the machine instructions:

```
sort Instruction .
subsort Instruction < Prog .

op load   : RamAddr -> Instruction .
eq (A,P := M[n],P + 1) = load(n) .
      ⋮        ⋮
```

The reason to order the equations in this way is that they are used as (left to right) rewrite rules at the last stage of the compilation process, to translate the semantics to the

syntax of the assembly language. In other words, when the assignment statements (used as patterns to define the instructions) are generated, they are automatically replaced by the corresponding instructions names; numeric values could be used instead, if the purpose was to produce binary code.

6.5.5 Compiling with Theorems

Now we present one of the main achievements of the mechanisation: the provably correct reduction theorems can be used effectively as rewrite rules to carry out the compilation task. All that needs to be done is to input the source program with a symbol table that allocates addresses for its global variables **v**, and ask OBJ3 to reduce this program using the reduction theorems[6]. The output is a normal form program which represents the target machine executing the corresponding instructions.

The process is carried out automatically. Every subterm matching the left-hand side of a one of the reduction theorems is transformed in the way described by the right-hand side of the theorem. As we have ordered the theorems in such a way that their right-hand sides refine the corresponding left-hand sides, each application can only lead to refinement. Therefore compilation is itself a proof that the final normal form program is a refinement of the initial source program. A very simple example is given below.

```
OBJ> let v = x,y,z .
OBJ> let Ψ = {x ↦ 101} ∪ {y ↦ 102} ∪ {z ↦ 103} .
OBJ> reduce Ψ̂_v(x := y bop (uop z)) .
rewrites: 386
result:  P,A :[0, (P=0 ∨ P=1 ∨ P=2 ∨ P=3 ∨ P=4 ∨ P=5) ->
                                 (P=0) → load(103)
                               □ (P=1) → uop-A
                               □ (P=2) → store(104)
                               □ (P=3) → load(102)
                               □ (P=4) → bop-A(104)
                               □ (P=5) → store(101),
               6]
OBJ> show time .
10.367 cpu     20.033 real
```

The application of the simulation function to the source program ensures that the data refinement phase will be accomplished by using the associated distributivity equations. Note in particular that the new address 104 was allocated to hold the value of a temporary variable created during the simplification of the expression. The last line shows the time (in seconds) consumed to carry out this reduction on a Sun 4/330 with 32 MB RAM.

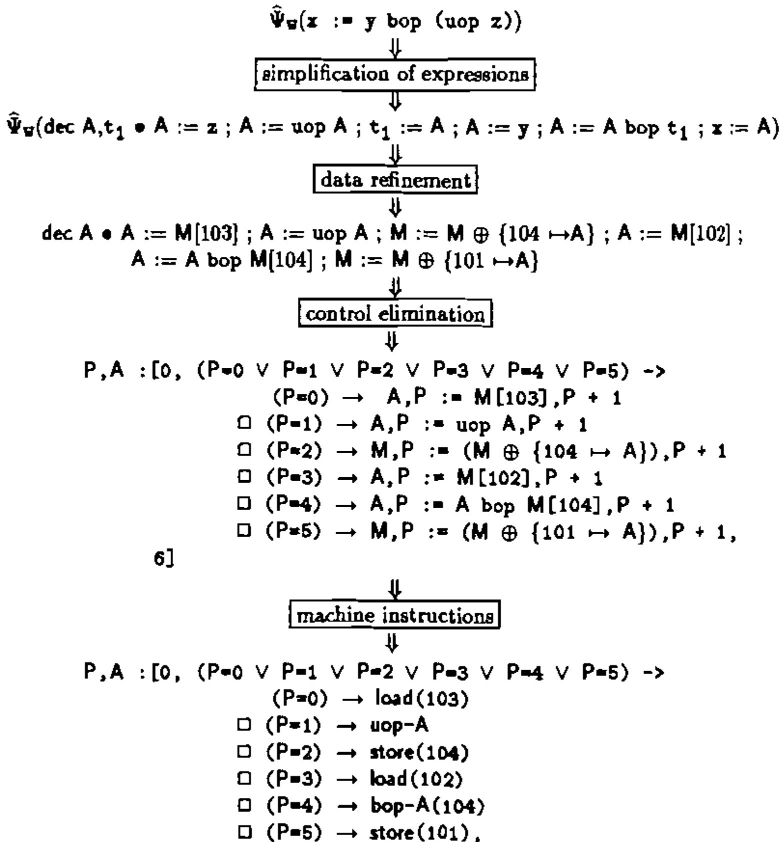
[6] Recall that the algebraic laws play no role here; they were needed only to prove the reduction theorems, and not to carry out compilation.

The guarded command set of a normal form program is an abstract representation of m (the store for instructions). We have not mechanised the actual loading process, which corresponds to extracting (from the guarded command set) the mapping from addresses to instructions, and use this to initialise m. For the above example this mapping is

{0 ↦ load(103)} ∪ {1 ↦ uop-A} ∪ {2 ↦ store(104)} ∪
{3 ↦ load(102)} ∪ {4 ↦ bop-A(104)} ∪ {5 ↦ store(101)}

and the value of m outside the range 0..5 is arbitrary. In this case, the execution of the guarded command set has the same effect as the execution of m[P].

Although not apparent to the user of the compiler, the compilation is carried out phase by phase. For the above example, the result of each phase is given below.



The last step above entails a simple syntactic transformation of the patterns used to define instructions with the corresponding instruction names.

As discussed in Chapter 4, the simplification of expressions must be performed first; no restriction was imposed regarding the order of the other two phases of compilation. In practice, however, it turned out to be much more efficient to carry out data refinement before control elimination. One reason is that the transformations associated with this last phase increase the size of the program significantly, with expressions involving the program counter which are irrelevant for data refinement. But most importantly, the normal form uses the operator $\square$ which is both associative and commutative, and the matching of associative-commutative operators is an exponential problem. As data refinement is carried out by applying distributivity equations all the way down to the level of variables, the matching involved becomes very expensive, and (performing it after control elimination) makes the process impractical even for prototyping purposes.

It is possible to control the sequence of application of rules in OBJ3 using *evaluation strategies*. For example, by declaring the simulation function $\widehat{\Psi}_{\Psi}$ as *strict*, we ensure that its argument is fully reduced before any of the distributivity equations are applied; this means that the simplification of expressions is carried out before data refinement. As explained before, the equations related to control elimination are not applied before the program is tagged (this is easily controlled by pattern matching). We also use evaluation strategies to ensure that the tagging is performed only when the data refinement is complete. As a consequence, control elimination is the last phase to be accomplished.

6.6 Final Considerations

We have shown how to use the OBJ3 term rewriting system to mechanise a non-trivial application. In particular, we believe to have successfully achieved three main results using a single system:

- A formal specification of all the concepts involved in this approach to compilation.
- Verification of some of the related theorems.
- Use of the theorems as a compiler prototype.

Although we have not verified all the theorems (as this was not the main purpose) the verification of a relevant subset gives some evidence that the task is feasible and relatively straightforward, especially considering that a complete framework is now in place.

Below we present a more detailed analysis of the mechanisation. First we discuss general aspects, and then we consider more specific features related to the use of OBJ3; finally we discuss some related works which report on the use of other systems to automate similar applications.

6.6.1 The Mechanisation

Even a simple attempt to automate an application gives (at least) a better insight into the problem. This is because many aspects are usually left out of (or implicit in) an informal presentation. For example, the OBJ3 presentation formally records the fact that the algebraic laws are independent of a particular expression language, provided this language includes the boolean expressions; explicit instantiations were defined when necessary.

We also had to deal with three related aspects not addressed initially: the creation of fresh local variables for the simplification of expressions, allocation of distinct addresses for local variables (also distinct from the ones used for the global variables), and, similarly, the allocation of distinct addresses for the machine instructions yielded by the compilation process.

Another major aim of a mechanisation is to check the correctness of hand proofs. With this respect, no serious error or inconsistency was found; but the mechanisation helped to uncover a few hidden assumptions (especially concerning non-freeness conditions) as well as the omission of references to laws necessary to justify some proof steps.

The only proof steps carried out completely automatically were the simplification of terms involving substitution, and the reduction of non-freeness conditions. Application of the laws required our full guidance. This is a consequence of the fact that our algebraic system is non-confluent and, furthermore, it includes non-terminating rules. Even so, the automated proofs are less laborious (and safer) than their manual versions in that the user is freed from handwriting the results of application of laws, especially regarding long terms which occur as intermediate steps in the proofs.

On the other hand, in a manual proof we sometimes allow ourselves to justify a given transformation by citing the necessary laws, leaving implicit the details of how the laws are actually used to achieve the transformation. But in a mechanical verification, every single step has to be explicitly justified, and the process may become extremely tedious. Therefore the encoding of new laws which combine the effect of more basic ones deserves special consideration. They play an increasingly important role as the number of theorems to be verified grows.

For example, the combination of assignments to distinct variables can be achieved by first normalising the left-hand sides (by adding identity assignments and using the symmetry law of multiple assignments), and then applying the law to combine assignments to the same variables. However, this process may require many applications of these laws; the same effect can be achieved using the law

```
cq  (x := e ; y := f) = (x,y := e,f[x <- e])  if x \\ y .
```

which is easily derived from the ones mentioned above. Similarly, it is possible (in some cases) to swap the order of two assignments using only the basic laws of assignment. But the process is more concisely captured by the law

```
cq  (x := e ; y := f) = (y := f[x <- e] ; x := e)  if y \\ (x := e) .
```

Another simple example is the instantiation of the (least) fixed point laws to deal with iteration. This saved us from rewriting iteration in terms of recursion only to apply these laws and then rewrite the result back to the iteration form. Further investigation may reveal more powerful strategies to combine laws.

One of the main benefits of the mechanisation is the possibility of compiling with theorems. Once the compiling specification (given by a set of reduction theorems) was in place, no additional effort was required to produce a (prototype) implementation. This was a consequence of the fact that the reduction theorems have the form of rewrite rules.

The only unexpected result of carrying out all the work was to realise that data refinement could be performed before control elimination. This was motivated by the fact that an initial version of the prototype which executed data refinement after control elimination was extremely inefficient, as discussed in the previous section.

One final aspect we wish to address is the reliability of our mechanisation. As mentioned before, programs have both a syntactic and a semantic existence, and we have not formally distinguished between them. Rather, we have grouped the equations which define the syntactic operators in a separate context (module) from that of the algebraic laws. But this does not prevent an unadvised (or badly intentioned) user from combining the modules and derive an inconsistency such as (we assume that the variables **x**, **y** and **z** are distinct)

```
  false
= {from equations defining non-freeness}
  y \ (x:=y; x:=z)
= {combine assignments}
  y \ (x:=z)
= {from equations defining non-freeness}
  true
```

This is a consequence of applying a semantic transformation (the combination of assignments) to a program which should have been treated as a purely syntactic object. Such derivations would be automatically disallowed if these syntactic operators were built-in to OBJ3, since in this case the user would not have access to their equations.

6.6.2 OBJ3 and 2OBJ

The normal form approach to compilation was conceived as a carefully structured hierarchy of concepts which is worth preserving in any attempted mechanisation. The parameterised modules of OBJ3 were a fundamental tool to achieve this objective.

For example, we have used distinct instantiations of the module describing the reasoning language to deal with each phase of compilation, since each phase has its own requirements regarding expressions. But clearly, the reasoning language may serve many other useful purposes such as to prove properties about programs, to perform optimisations or to reduce programs to a different normal form. The module describing the reasoning language

could actually be part of a library concerning program development. The module which groups the normal form reduction theorems is equally generic and can be instantiated to deal with different target machines.

We have also demonstrated the convenience of subsorting and the operation declaration facilities of OBJ3. The fact that associativity, commutativity and identity properties can be declared (rather than stated by explicit equations) substantially simplifies the proof process. Apart from rewriting modulo these properties, OBJ3 provides mechanisms for selecting subterms (for the purpose of localised transformations) which take them into account. For example, as sequential composition is associative, the term

```
p ; q ; r
```

stands for an equivalence class containing `(p ; q) ; r` and `p ; (q ; r)`. Furthermore, in this case a direct selection of any contiguous subterm is possible, as `(p ; q)` or `(q ; r)`. For terms with a top operator which is both associative and commutative, a subset selection mechanism is available. Consider the term

```
p ⊓ q ⊓ r
```

We can apply a given transformation to the subterm (p ⊓ r) by directly selecting this subterm using the notation provided.

Our experience [69] with systems which do not include such facilities showed that the proofs are (at least) twice as long as the ones carried out using OBJ3. The explicit application of associativity and commutativity laws is very laborious and diverts the user's attention from more relevant proof steps.

As a language to describe theories, the only significant limitation of OBJ3 for our application is the lack of inequational rewriting. Higher-order logic would allow a more natural encoding and use of the (least) fixed point laws, but this was overcome by using substitution. Although not as convenient, it was not a major problem in practice.

The main drawbacks of OBJ3 are related to proof support; this was extensively discussed earlier. A more pragmatic limitation is efficiency. Although the speed of the rewrites is reasonable for interactive theorem proving, it is not as satisfactory for automatic reductions involving a large number of rewrites. In particular, the use of the theorems to carry out compilation is only acceptable for prototype purposes, as illustrated in the previous section.

Most of the problems discussed above are being taken into account in the development of the 2OBJ system [29] which is being built on top of OBJ3. Broadly, this is a meta-logical framework theorem prover in the sense that it is independent of a particular logical system. The desired logical system can be programmed by the user by encoding it in equational logic. 2OBJ has a user-friendly interface and allows user-defined tactics for the particular application domain.

One of the logical systems available supports order sorted, conditional, (in)equational rewriting. In particular, the inequational rewriting module of 2OBJ accepts as input an

OBJ3 specification and treats the rules annotated with the ordering relation (as illustrated earlier) as inequations.

A closer investigation of 2OBJ is one of the suggested topics for future work; not only more confidence will be gained in the proofs (because of a proper account of inequations), but user-defined tactics can be defined to improve the proof process.

6.6.3 Other systems

In a previous study [69], we explored the suitability of some systems to formalise a small subset of this approach to compilation. Apart from OBJ3, we considered the B-Tool [74], the Veritas+ Environment [19] and the occam Transformation system [32]. A summary of our experience with each system is given below. Related experience of others, using the Larch Prover (LP) [26] and Higher-Order Logic (HOL) [34] is also discussed.

A specification in the B-tool is formed of a collection of units called *theories*. Unlike OBJ3, these units do not embody any idea of a module—they are just "rule containers". The lack of constructions for type definition and variable as well as operator declarations is another drawback of the B-tool. A helpful feature is the built-in implementation of non-freeness and substitution. Not only this saves a substantial specification effort in our case, but the associated reductions are carried out very efficiently. As discussed previously, this also avoids the need to distinguish between the syntactic and the semantic views of programs. Regarding theorem proving, the rewriting facilities are similar to those available in OBJ3, but there is no support for dealing with associativity, commutativity or identity.

As a specification language, Veritas includes some interesting features. Although *signatures* (the specification unit) may not be parameterised, similar facilities may be obtained by using higher-order functions and polymorphic datatypes. Besides polymorphism, the type system includes subtypes and dependent types. This allows us to express the precise domain of partial operators such as multiple assignment statements. The main drawback for our application is the difficulty of coding the algebraic laws. Defining the reasoning language as a datatype, it is impossible to postulate or prove any of the algebraic laws. The reason is that a datatype is a free algebra, and therefore terms built from the constructors cannot be equated. The laws have to be established as theorems, from a semantic function which expresses their effect, through a very laborious process.

The occam Transformation system implements an application similar to ours. Its purpose is to allow semantic preserving transformations of occam processes, where the transformations are justified by the algebraic laws obeyed by the language [68]. As occam includes some of the operators of our reasoning language, it is possible to use the system, for example, to prove some derived laws. In principle, it is even possible to extend the system (which is implemented in SML [36]) with additional features which would allow us to reason about the whole compilation process. For example, new operators (especially the specification ones) with their algebraic laws would be necessary. While most of the desirable features can be easily coded in SML, the implementation of theorem proving facilities such as mechanisms to deal with associativity and commutativity is a complex

task.

A work closely related to ours is reported in [70]. It investigates the use of LP to verify the proof of a compiler for a small subset of occam, not including parallelism or communication. As in our case, the reasoning framework is an extension of the source language with its algebraic laws. The emphasis in [70] is the specification of the reasoning framework; only a few simple proofs were mechanically checked. Also, the aspects related to data refinement were not formalised.

As LP is also a term rewriting system, the specification described in [70] shares many features with ours. However, it is relatively less concise and readable since LP provides no module facilities, and is based on multi-sorted, rather than order sorted, logic; therefore subsorting is not available. Also, the operation declaration facilities are not as flexible as in OBJ3. There is a mechanism to deal with associative-commutative operators, but nothing is provided for operators which are only associative. Identity properties also have to be stated by explicit equations. On the theorem proving side, LP incorporates more elaborate mechanisms than OBJ3; apart from term rewriting, it supports proof by induction, case analysis and contradiction. However, the support for an interactive (step by step) application of rules is not as flexible as in OBJ3.

The work reported in [6] deals with an important aspect that we have not addressed: the correctness of the basic algebraic laws. A specification language (similar to ours) with weakest precondition semantics is formalised using the HOL system, and a number of refinement laws are (mechanically) proved. Although in principle we could do the same in OBJ3 (or perhaps 2OBJ), a system based on higher-order logic like HOL seems more appropriate for this purpose. The reason is that in the predicate transformer model, programs are regarded as functions on predicates, and therefore the reasoning is essentially higher order. However, it is our view that, for the purpose of using the laws for program transformation, a system like OBJ3 is more suitable, as it provides powerful rewriting capabilities.

Chapter 7

Conclusions

> There is a great danger associated with people's perception of new concepts. If improved methods are used to tackle the same sort of problems previously handled by *ad hoc* methods, the systems created could be far safer. If, on the other hand, the improved methods are used to justify tackling systems of even greater complexity, no progress has been made.
>
> — C.B. Jones

We have presented an innovative approach to compilation, and showed how it can be used to design a compiler which is correct by construction. The compilation process was characterised as a normal form theorem where the normal form has the same structure as the target executing mechanism. The whole process was formalised within a single (and relatively simple) semantic framework: that of a procedural language which extends the source language with additional specification features.

The specification (reasoning) language was defined as an algebraic structure whose axioms are equations and inequations (laws) characterising the semantics of the language. The advantage of this semantic style is abstraction; it does not require the construction of explicit mathematical models, as with denotational or operational semantics. As a consequence, extensions or modifications to the semantics may require the alteration of only a few laws, unlike, for example, a denotational description which would require alterations to the mathematical model and consequent revision of every semantic clause. In the operational style, proofs are typically by structural induction and/or by induction on the length of derivations or on the depth of trees. Therefore if the language is extended, the proofs need to be revised. In the approach we have adopted, we hardly use induction; this is implicitly encoded in the fixed point laws. A purely algebraic reasoning gives some hope concerning the modularity of the approach.

By no means are we claiming a general superiority of the algebraic approach over the other approaches to semantics. Each style has its appropriate areas of application. For example, postulating algebraic laws can give rise to complex and unexpected interactions between programming constructions; this can be avoided by deriving the laws from a mathematical

model, as briefly illustrated in Chapter 3. Similarly, an operational semantics is necessary to address practical aspects of the implementation and efficiency of execution. A general theory of programming dealing with these three semantic approaches is suggested in [44]. In particular, it is shown how an algebraic presentation can be derived from a denotational description, and how an operational presentation is derived from the former.

The identification of compilation as a task of normal form reduction allowed us to capture the process in an incremental way, by splitting it into three main phases: simplification of expressions, control elimination and data refinement. The ideas are not biased to a particular source language or target machine. Notable are the theorems for control elimination which can be instantiated to deal with a variety of ways of encoding control state. The independence from a source language is not so evident, as we had to adopt a particular notation in order to formulate the mathematical laws and illustrate the task of designing a compiler. However, our source language includes features commonly available in existing procedural languages. It can serve as a target for a *front-end* compiler for languages which use more conventional notations.

We initially dealt with a very simple source language to illustrate how a complete compiler can be designed. The source language was then extended with more elaborate features (procedures, recursion and parameters), and the associated reduction theorems were proved. This extension gave some evidence about the modularity of the approach; each new feature was treated in complete isolation from all the other constructions of the language. The reuse of laws, lemmas and theorems is of particular relevance. For example, the (derived) laws of iteration and the lemmas used to prove the reduction theorem for sequential composition have been of more general utility; they were used somehow to prove the reduction theorems for all the language features treated subsequently.

The whole approach has been carefully structured to simplify its mechanisation. We have illustrated the process for the simple version of the source language, using the OBJ3 term rewriting system. The concepts were formalised as a collection of algebraic theories, and some of the related theorems were verified and used as rewrite rules to carry out compilation automatically. The mechanisation preserves the original structure of the algebraic theories. As discussed in the previous chapter, it can be useful for many other purposes, such as proving properties about programs, performing optimisations or reducing programs to a different normal form.

In summary, we believe our work to be a modest contribution to three important fields of software engineering:

- formal methods and techniques — with a relatively large application of refinement algebra;
- compiler design and correctness — with the exploration of a new approach comprising aspects such as simplicity and modularity; and
- mechanical theorem proving — particularly, the use of term rewriting systems as a tool for specification, verification and (prototype) implementation.

But there is much more to be done before we can claim that this approach will generalise to more complex source languages or target machines. In the following section we discuss related work. Some extensions to our work are discussed in Section 7.2. We finish with a critical analysis of the overall approach to compilation.

7.1 Related Work

In Chapter 2 we gave a brief overview of refinement calculi and techniques; in the previous chapter we compared OBJ3 to some other theorem provers. Here we concentrate on compiler design and correctness. Nevertheless, there is an extensive (and expanding) literature and we have no intention of covering the field. Rather, we consider closely related work and comment on a few approaches based on distinct semantic styles.

Closely Related Approaches

Nelson and Manasse [64] have previously characterised the compilation process as a normal form theorem. But the authors formalise the reduction process in more concrete terms, using a program pointer to indicate the location in memory of the next instruction to be executed. By using assumptions, assertions and generalised assignment, we have abstracted from a particular way of encoding control state; the use of a program pointer is one possible instantiation. Another difference is the reasoning framework used. They justify the correctness of the transformations by appealing to the weakest precondition calculus. We have formalised the normal form reduction process as an algebra where the central notion is a relation of refinement between programs. The use of algebraic laws seem to allow conciser and more readable proofs, apart from the fact that it makes the mechanisation easier. We have also dealt with programming features not addressed by them; these include procedures, recursion and parameters.

The first approach to prove correctness of compiling specifications using algebraic laws (in the style we have used here) was suggested by Hoare [43]. In this approach, compilation is captured by a predicate $C\,p\,s\,f\,m\,\Psi$ stating that the code stored in m with start address s and finish address f is a correct translation of the source program p; Ψ is a symbol table mapping the global variables of p to their addresses. C is defined by

$$C\,p\,s\,f\,m\,\Psi \stackrel{def}{=} \hat{\Psi}(p) \sqsubseteq \mathcal{I}\,s\,f\,m$$

where $\hat{\Psi}(p)$ is a simulation function defined in the usual way and $\mathcal{I}$ is an interpreter for the target code which can be considered a specialisation of the normal form for a particular machine. Compilation is specified by a set of theorems, one for each program construction. For example,

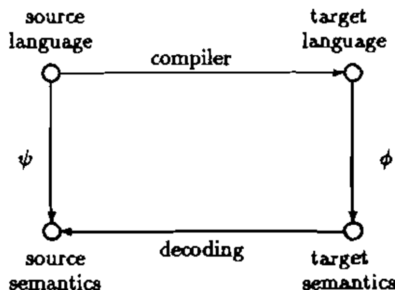
$$\begin{array}{l}\text{If } m[s] = \mathsf{load}(\Psi y) \text{ and } m[s+1] = \mathsf{store}(\Psi x), \text{ then}\\ C(x := y)\,s\,(s+2)\,\Psi\end{array}$$

The reasoning is conducted in much the same way as we have illustrated in previous chapters. As the theorems have the form of Horn Clauses, they can be easily translated into a logic program [10]; although a formal proof of correctness of this translation has not been attempted. Despite the similarities between the two approaches, there is a significant conceptual difference. As discussed above, the idea of an abstract normal form allowed us to capture compilation in an incremental way. The separation of the process into phases allowed the formalisation of control elimination independently of a target machine. Furthermore, as the theorems have the form of rewrite rules, we could use a term rewriting system both to verify the proofs and to carry out compilation.

Work has been undertaken to transform programs written in a subset of occam into a normal form suitable for an implementation in hardware [39]. A circuit is described in a way similar to that we have represented a stored program computer. Broadly, the state of a synchronous circuit is formed from a variable representing its control path and a list of variables representing its data path. The normal form comprises an assumption about the activation of the circuit; a loop (executed while the circuit is activated) which specifies state changes; and an assertion which determines the final control state of the circuit, if the loop terminates. An extra feature is the use of timed processes to specify the execution time (in clock cycles) of assignments which are used to model the state change of both the control path and the data path of the circuit. The authors show how an arbitrary source program can be reduced to normal form. However, the translation from the normal form to a *netlist* (a list of gates and latches, which is a standard form of hardware description) is addressed only informally.

Algebraic and Denotational Approaches

In more conventional algebraic approaches, compiler correctness is expressed by the commutativity of diagrams of the form



where the nodes are algebras and the arrows are homomorphisms. This form of diagram was first introduced by Morris [58], building on original work of Burstall and Landin [14]. Thatcher, Wagner and Wright [73] and many others put forward similar ideas.

Similar commutative diagrams are usually adopted for proving compilation based on denotational semantics. Noteworthy is the work of Polak [67]. He gives the denotational semantics of both the source language (a large subset of Pascal) and the target language, a high-level assembly language. His work is not confined to code generation; rather, he treats the complete compilation process. Furthermore, the compiler itself is written in a version of Pascal extended with features to allow a formal documentation in terms of pre- and postconditions in Hoare-style semantics. The compiler is systematically developed from the denotational semantics of the source and the target languages. The proofs were mechanically verified.

Chirica and Martin [17] also deal with compiler implementation. They developed an approach for proving the correctness of compiler implementations from given specifications. The approach is similar to that of Polak, but a different boundary between compiler specification and implementation is suggested. Broadly, semantic correspondence between source and target programs is dealt with at the specification level; implementation correctness deals only with the syntax of source and target programs.

Operational Approaches

The first attempt to formalise the compilation process is attributed to McCarthy and Painter [52]. Although their work is limited in scope (they treated a simple expression language), it gave rise to a style of verification known as *interpreter equivalence*. In general terms, the semantics of the source and the target languages is given by interpreters which characterise the meaning of programs by describing their effect upon the corresponding *execution environments*. The translation is described by a relation between the execution environments of source and target programs. Correctness can be expressed by a diagram with a similar form to that presented in the previous section.

More recently, an approach to systems verification based on interpreter equivalence was suggested by a group at Computational Logic, Inc. [7]. The approach has been applied to the development and mechanical verification of a *stack* of system components, including:

- a compiler for a subset of the high-level procedural language Gypsy, where the target is the Piton assembly language [78];
- a link-assembler from Piton to binary code for the FM8502 microprocessor [53]; and
- a gate-level design of a subset of the FM8502 microprocessor [48] .

All the components were formalised by interpreters written as functions in the Boyer-Moore logic; the verification was carried out using the Boyer-Moore theorem prover [13]. This is perhaps the most significant effort in the field of (mechanically) verified systems. Their approach is independent of any particular component, and it deals with the integration of components to form a verified stack. However, reuse of design and proofs is not addressed. Every translation from a source to a target language is designed and proved from scratch.

Compiler Generators and Partial Evaluation

Although our approach is not biased towards a source language or a target machine, we have not gone as far as addressing the design of compiler generators. Many systems (based on distinct semantic approaches) have been developed. As an example we can cite the classical work of Mosses [62], using denotational semantics. Only recently, the correctness of such systems has gained some attention. The Cantor system [65] generates compilers for imperative languages defined using a subset of action semantics [63]. Many imperative features can be expressed, but the considered subset of action semantics is not powerful enough to express recursion. An algebraic framework was used to design the system and prove its correctness.

Partial evaluation is a very powerful program transformation technique for specialising programs with respect to parts of its input. Applications of partial evaluation include compilation, compiler generation or even the generation of compiler generators. We quote an explanation from [49]:

> Consider an interpreter for a given language S. The specialisation of this interpreter to a known source program s (written in S) *already* is a target program for s, written in the same language as the interpreter. Thus, partial evaluation of an interpreter with respect to a fixed source program amounts to compiling. [...]
>
> Furthermore, partially evaluating a partial evaluator with respect to a fixed interpreter yields a compiler for the language implemented by the interpreter. And even more mind-boggling: partially evaluating the partial evaluator with respect to itself yields a compiler generator, namely, a program that transforms interpreters into compilers.

Partial evaluation is a very active research topic, and some powerful systems have been developed. For example, Jones *et al* [50] implemented a self-applicable partial evaluator, called λ-mix, for the untyped lambda calculus. It has been used to compile, generate compilers and generate a compiler generator. Furthermore, it is perhaps the only existing provably correct partial evaluator [33].

One aspect not yet addressed by the partial evaluation approach is the generation of compilers which produce code for conventional machine architectures. Code is usually emitted in a lambda notation.

7.2 Future Work

Our work can be extended in many ways, from the treatment of more elaborate source languages and/or target machines to a provably correct compiler combining both software and hardware compilation.

More on Control Structures

In Chapter 5 our strategy to implement recursion was to allocate a separate stack for each recursive program. This stack was represented by a local variable in the resulting normal form program. In the case of nested recursion, the normal form reduction process generates stacks of stacks. This can be implemented using the *cactus stack* technique, as previously discussed.

For a single stack implementation, further work is necessary. One possibility is to use the reduction rule for recursion as it stands now and then perform an additional refinement step to implement the nested stacks using a single stack. As this is by no means a trivial data refinement problem, it might be easier to avoid nested stacks from the beginning. In this case, the theorem for recursion has to be modified to reuse a stack variable which may have been allocated for compiling an inner recursion.

A further topic of investigation is the extension of our source language with even more complex structures such as parallelism, communication and external choice (as, for example, in occam). An initial attempt to handle these features is described in [38], where a a communication-based parallel program is transformed into another (yet parallel) program whose components communicate via shared variables.

Because of the high-level of abstraction of constructions to implement concurrency, it seems more appropriate to carry out their elimination at the source level, rather than generate a normal form directly. If the target program yielded by the process of eliminating concurrency is described solely in terms of our source language, then we can reduce this program to normal form and thus obtain a low level implementation of concurrency.

Types

We have taken the simplified view of not dealing with type information. It is possible to extend our reasoning language with data types in the usual way. For example, typed variables can be introduced by

$$\mathbf{dec}\, x : T \bullet p$$

where x is a list of variables and T an equal-length list of type names, and the association of types with variables is positional.

Types restrict the values that can be assigned to variables. As a consequence, they introduce a proof obligation to check if these restrictions are respected. This is known as *type checking*. Therefore our algebraic system must ensure a consistent use of types.

Most of the algebraic laws do not need to be changed, as their use would not give rise to type inconsistences, provided the term to be transformed is *well-typed* to start with. However, a few laws would require extra conditions. For example, the law

$$\mathbf{dec}\, x : T \bullet p \sqsubseteq \mathbf{dec}\, x : T \bullet x := e;\ p$$

(when used from left to right) allows the introduction of the assignment $x := e$. Clearly, e must have type T in this case (or at least a type compatible with T if the type system

supports subtypes or any form of type conversion). The laws which allow the introduction of free (meta-)variables, such as x and e in the above case, are the only ones which can violate type information. One way to deal with the problem is by carrying type information around; for example, by tagging occurrences of variables and expressions with their types.

The introduction of types also affects compiler design. More specifically, it increases the complexity of the data refinement phase—we have to show how the various types of the source language can be represented in a usually untyped (or single-typed) target machine.

The implementation of *basic* types is normally achieved by very simple translation schemes. For example, there is a standard technique for translating booleans using a numerical representation [1]. Type constructors such as arrays and records are more difficult to implement. In Appendix A, we suggest a scheme for compiling static arrays and discuss its (partial) implementation in OBJ3, with a small example.

The scheme to implement arrays was designed in complete isolation from the remaining features of our language. None of the previous results needed to be changed. This gives some more evidence about the modularity of our approach to compilation. But further investigation concerning the compilation of basic types and type constructors is required. For example, non-static types such as dynamic arrays or linked lists will certainly require a much more elaborate scheme than the one devised for static arrays.

Code Optimisation

We have briefly addressed store optimisation when dealing with the creation of temporary variables for the elimination of nested expressions. Regarding code optimisation, only a very simple rule was included in connection with the compilation of boolean expressions. An important complement to our work would be the investigation of more significant optimising transformations which could be performed both on the source and on the target code.

The most difficult optimisations are those which require data flow analysis. The main problem is the need to generate (usually) complex structures to store data flow information, as well as carrying these structures around so that the optimisation rules can access them. In our algebraic framework, a promising direction seems to be the encoding of data flow information as assumptions and assertions; they satisfy a wide set of laws which allow them to be flexibly manipulated within a source program.

Local optimisations are much easier to describe and prove. Some algebraic laws can be used directly to perform optimisations on the source code. For example,

$$(x := e;\ x := f) \ = \ (x := f[x \leftarrow e])$$

may be useful for eliminating consecutive assignments to the same variable. Another example is the law which allows the transformation of tail recursion into iteration:

$$\mu X \bullet ((p;\ X) \lhd b \rhd q) \ = \ (b * p);\ q$$

Local optimisations on a target program are known as *peephole optimisations*. The general aim is to find sequences of instructions which can be replaced by shorter or more efficient sequences. The (very abstract) normal form representation of the target machine may provide an adequate framework to carry out optimisations and prove them correct. As a simple example, the following equation allows the elimination of *jumps to jumps*

$$\mathsf{P} : [s, \begin{pmatrix} & (\mathsf{P} = j) \to \mathsf{jump}(k) \\ \square & (\mathsf{P} = k) \to \mathsf{jump}(l) \\ \square & Q \end{pmatrix}, f] \;=\; \mathsf{P} : [s, \begin{pmatrix} & (\mathsf{P} = j) \to \mathsf{jump}(l) \\ \square & (\mathsf{P} = k) \to \mathsf{jump}(l) \\ \square & Q \end{pmatrix}, f]$$

The aim of this transformation is to eliminate all jumps to k, one at a time. When there are no jumps to k then it is possible to eliminate the guarded command $(\mathsf{P} = k) \to \mathsf{jump}(l)$ provided it is preceded by an unconditional jump instruction (something of the form $(\mathsf{P} = k - 1) \to \mathsf{jump}(n)$, with n different from k). Q stands for an arbitrary context containing the remaining instructions. Notice that commutative matching as provided in OBJ3 allows the above rule to be implemented straightforwardly.

It would also be interesting to investigate some machine-dependent optimisations such as register allocation and the replacement of sequences of machine instructions with other sequences known to execute more efficiently. But this only makes sense in the context of a more complex target machine than the one we have considered here. Information necessary to optimise register allocation can be encoded as assumptions and assertions, following the style suggested in [11].

More on Mechanisation

As described in the previous chapter, a significant amount of work concerning the mechanisation of our approach to compilation has been carried out; but much more could be done. For example, we have not considered procedures, recursion or parameters. Extending our OBJ3 specification with rewrite rules to eliminate these features would produce a prototype for a more interesting language. Clearly, it would be necessary to extend the target machine with stacks and related operations in order to support recursion.

The more complex the theorems and their proofs are, the more likely the occurrence of errors is. The complete verification of the reduction theorem for recursion would be a worthwhile exercise.

The ideal would be to mechanise any new translation scheme which is designed. Apart from the benefit of verification, the mechanisation (as we have addressed in this work) helps to ensure that the scheme is described in sufficient detail to be implemented.

As this will eventually become a relatively large application of theorem proving, a good deal of proof management is required. A promising direction seems to be "customising" a system like 2OBJ for this application, as discussed in the previous chapter.

Compiler Development

The only limitation that prevents the specification of the compiler (written in OBJ3) to be used as an actual implementation is efficiency. As discussed in the previous chapter, one reason for its inefficient execution is the use of commutative and associative matching, which is an exponential problem. Therefore the elimination of these features from the present specification is an essential step towards improving efficiency.

A more serious constraint is imposed by the implementation of OBJ3 itself; currently the language is interpreted and term rewriting is carried out at a speed which is acceptable only for prototyping purposes. Therefore there is the need to develop an implementation of the compiler in some programming language that is implemented efficiently.

As our reference against which to check the implementation is a declarative specification of the compiler (given by a set of rewrite rules), it might be easier to develop a functional implementation. A language such as ML [66] could be useful for this purpose.

The main task is therefore to derive a compilation function, say $\mathcal{C}$, from the rewrite rules. It is possible to develop the implementation with a similar structure as the OBJ3 specification; $\mathcal{C}$ is defined by the composition of three functions, one for each phase of compilation (simplification of expressions, control elimination and data refinement):

$$\mathcal{C}\, p\, \Psi\, s \stackrel{def}{=} \hat{\Psi}(ControlElim(ExpSimp(p), s))$$

where p is a source program, Ψ its symbol table and s the start address of code. The definition of each of the above functions can be systematically generated from the rewrite rules in the corresponding OBJ3 modules. For example,

$$\begin{array}{lcl} ExpSimp(x := e) & = & \mathsf{dec}\ \mathsf{A} \bullet ExpSimp(\mathsf{A} := e);\ x := \mathsf{A} \\ ExpSimp(\mathsf{A} := \mathsf{uop}\ e) & = & ExpSimp(\mathsf{A} := e);\ \mathsf{A} := \mathsf{uop}\ e \\ \vdots & & \vdots \end{array}$$

The aim of this transformation is to end up with a functional program still in OBJ3, but including only features available, say, in ML; so a *simple* syntactic transformation would produce an ML program which could then be translated to machine code using an efficient ML compiler [2].

While this is a relatively simple way of producing an efficient compiler from the specification in OBJ3, the approach is rigorous rather than completely formal. One source of insecurity is that even an apparently trivial syntactic transformation from OBJ3 into ML (more generally, from a language to any other language) may be misleading; the other is the ML compiler itself (unless its correctness had been proved).

An approach which avoids the need for an already verified compiler is suggested in [15] and further discussed in [18]. It is based on the classical technique of ***bootstrapping***. The main goal is to obtain an implementation of the compiler written in the source language itself. [15] discusses work which was carried out to develop an implementation of a compiler for an occam-like language extended with parameterless recursive procedures. The implementation (in the source language itself) is formally derived from a specification of the compiler given in an inductive function definition style that uses a subset of LISP.

In principle, we could adopt a similar technique to derive an implementation of our compiler (written in our source language) from the specification in OBJ3. In this case, we could run the specification to automatically translate the implementation of the compiler into machine code. But as pointed out in [15], the formal derivation of the implementation from the specification is by no means a simple task.

Hardware and Software Co-design

Software compilation is cheap but produces code which is usually slow for many of the present applications; hardware compilation produces components which execute fast, but are expensive. A growing trend in computing is the design of systems which are partially implemented in software and partially in hardware. This of course needs the support of compilers which give the choice of compiling into software or hardware (or both).

We have addressed software compilation and, as discussed in the previous section, some work has been done for hardware compilation using a similar approach. A very ambitious project is to develop a common approach to support the design (and correctness proof) of a hardware/software compiler. The main challenge is to discover an even more general normal form which would be an intermediate representation of code describing hardware or software. Broadly, the structure of such a compiler would be as in Figure 7.1.

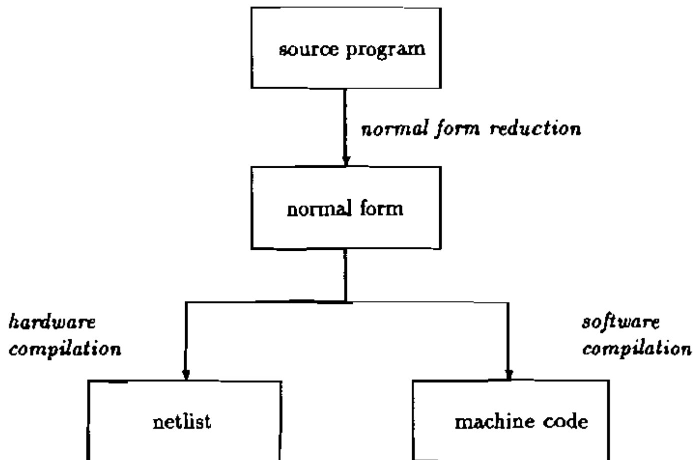


Figure 7.1: Hardware and software co-design.

7.3 A Critical View

In this final section we present a critical analysis of our work and try to answer the following questions:

- Will this ever be a practical way to write compilers?
- If not, is there any independent reason for pursuing research of this kind?
- To what extent should we believe in provably correct compilers (and systems in general)?

It is more than 25 years since the first approach to compiler correctness was suggested, and this still remains one of the most active research topics in computing. In the previous section we briefly described a few approaches, some based on different formalisms, but these are only a small fraction of the enormous effort that has been dedicated to the field.

We have defended the use of yet another approach. We believe it is very uniform and is based on a comparatively simple semantic framework. In spite of that, the overall task of constructing a correct compiler was not at all trivial. Many frustrating alternatives were attempted until we could discover reasonable ways of structuring the proofs. On the positive side, however, there is the hope that these will be of more general utility. The main purpose was not to show that a particular compiler correctly translates programs from a particular source language to a particular target language; but rather, to build a set of transformations that may be useful for tackling the problem in a more general sense.

Uniformity was achieved by reducing the task of compilation to one of program refinement. Although this is extremely difficult in general, it is much more manageable when dealing with a particular class of problems, such as compiler design. Our understanding of and intuition about compilation helped us to achieve a modular design; our knowledge about programming helped us in the reasoning (using algebraic laws) necessary to discharge the related proof obligations.

But the approach is not sufficiently mature yet. The search for deeper and more specific theorems to support the design of compilers for more powerful languages should continue. We do not have enough grounds to believe that it will ever play a role in writing practical compilers. In any case, we believe our work to be a contribution in this direction. Hopefully, it will be useful as a reference for further work in the field.

Concerning the problem of provably correct systems in general, there will always be a gap between any mathematical model and its implementation. Even if compilers, assemblers, loaders and the actual hardware are provably correct, at some stage we move from the mathematical world into the real world; and this transition can never be formalised or reasoned about. The purpose of verification is to reduce the occurrence of errors; their total absence can never be proved.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture* (LNCS 274), pages 301–324, 1987.

[3] R. J. R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. Technical report, Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[4] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Departments of Computer Science and Mathematics, Swedish University of Åbo, Finland, 1987.

[5] R. J. R. Back and J. von Wright. Refinement Calculus: Part I: Sequential Nondeterministic Programs. In *Stepwise Refinement of Distributed Systems* (LNCS 430), pages 42–66, 1990.

[6] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[7] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5:411–428, 1989.

[8] G. Birkhoff. *Lattice Theory.* American Mathematical Society, 1961.

[9] D. Bjorner *et al.* Final Deliverable of the ProCoS Project. Technical report, Computer Science Department, Technical University of Denmark, Lyngby, DK, 1992.

[10] J. Bowen. From Programs to Object Code Using Logic and Logic Programming. In *Proc. CODE'91 International Workshop on Code Generation*, Springer-Verlag, Workshops in Computing, 1992.

[11] J. Bowen and J. He. Specification, Verification and Prototyping of an Optimized Compiler. Technical report, Oxford University Computing Laboratory, 1992.

[12] J. Bowen *et al.* A ProCoS II Project Description: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 50:128–137, June 1993.

[13] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.

[14] R. Burstall and P. Landin. Programs and their Proofs: an Algebraic Approach. *Machine Intelligence*, 7:17–43, 1969.

[15] B. Buth *et al*. Provably Correct Compiler Development and Implementation. In *4th International Conference on Compiler Construction* (LNCS 641), pages 141–155, 1992.

[16] W. Chen and J. T. Udding. Program Inversion: More Than Fun! Technical report, Groningen University, April 1989.

[17] L. M. Chirica and D. F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transaction on Programming Languages and Systems*, 8(2):185–214, 1986.

[18] P. Curzon. Of What Use is a Verified Compiler Specification? Technical report, University of Cambridge, 1992.

[19] N. Daeche. Guide to IVE, the Interactive Veritas+ Environment. Technical report, University of Kent, April 1990.

[20] E. W. Dijkstra. Notes on Structured Programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, 1972.

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

[22] E. W. Dijkstra. Program Inversion. Technical report, EWD671, University of Technology, Eindhoven, 1978.

[23] E. W. Dijkstra. The Equivalence of Bounded Nondeterminacy and Continuity. In *Selected Writings on Computing*. Springer, New York, 1982.

[24] H. Ehrig and H. Weber. Programming in the Large with Algebraic Module Specification. In H. J. Kugler, editor, *Proc. IFIP, 10*. North-Holland, 1986.

[25] P. Gardiner and P. K. Pandya. Reasoning Algebraically about Recursion. *Science of Computer Programming*, 18:271–280, 1992.

[26] S. J. Garland and J. V. Guttag. An Overview of LP, The Larch Prover. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications* (LNCS 355), pages 137–155. Springer-Verlag, 1989.

[27] J. Goguen. *Theorem Proving and Algebra*. MIT Press, 1993. To appear.

[28] J. Goguen and J. Meseguer. Order Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. Technical report, SRI International, SRI-CSL-89-10, July 1989.

[29] J. Goguen, A. Stevens, K. Hobley, and H. Hilberdink. 2OBJ, A Metalogical Framework Based on Equational Logic. In *Philosophical Transactions of the Royal Society, Series A, 339*, pages 69–86. 1992.

[30] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[31] J. Goguen *et al.* Introducing OBJ. Technical report, SRI International, 1993. To appear.

[32] M. Goldsmith. The Oxford occam Transformation System. Technical report, Oxford University Computing Laboratory, January 1988.

[33] C. K. Gomard. A Self-Applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[34] M. J. C. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[35] D. Gries. *The Science of Programming*. Springer Verlag, New York, 1981.

[36] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical report, Edinburgh University, LFCS Report Series, ECS-LFCS-86-2, March 1986.

[37] E. A. Hauck and B. Dent. Burroughs B6500 stack mechanism. In *Proceedings 1968 Spring Joint Computer Conference*, Thomson Book Company, Inc., Washington, D.C., pages 245–251, 1968.

[38] J. He. Introduction to Hybrid Parallel Programming. Technical report, Oxford University Computing Laboratory, 1992.

[39] J. He, I. Page, and J. Bowen. A Provably Correct Hardware Implementation of occam. Technical report, **ProCoS** Project Document [OU HJF 9/5], Oxford University Computing Laboratory, November 1992.

[40] C. A. R. Hoare. Procedures and Parameters: an Axiomatic Approach. In *Symposium on the Semantics of Algorithmic Languages*, Lecture Notes in Mathematics 188, Springer Verlag, pages 102–116, 1971.

[41] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.

[42] C. A. R. Hoare. Algebra and Models. Technical report, Oxford University Computing Laboratory, 1991.

[43] C. A. R. Hoare. Refinement Algebra Proves Correctness of Compiling Specifications. In *3rd Refinement Workshop, Springer-Verlag, Workshops in Computing*, pages 33–48, 1991.

[44] C. A. R. Hoare. A Theory of Programming: Denotational, Algebraic and Operational Semantics. Technical report, Oxford University Computing Laboratory, 1993.

[45] C. A. R. Hoare, J. He, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 1993. To appear.

[46] C. A. R. Hoare and J. He. The Weakest Prespecification. *Information Processing Letters*, 24(2):127–132, January 1987.

[47] C. A. R. Hoare *et al.* Laws of Programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[48] W. A. Hunt. Microprocessor Design and Verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[49] N. D. Jones, P. Sestoft, and H. Sondergaard. An Experiment in Partial Evaluation. In *Rewriting Techniques and Applications* (LNCS 202), 1985.

[50] N. D. Jones *et al.* A Self-Applicable Partial Evaluator for the Lambda Calculus. In *1990 International Conference on Computer Languages*, IEEE Computer Society, 1990.

[51] L. Lamport. LaTeX: *A Document Preparation System*. Addison-Wesley, 1986.

[52] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In *Proceedings of Symposium on Applied Mathematics*. American Mathematical Society, 1967.

[53] J. S. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[54] C. Morgan. Procedures, Parameters, and Abstraction: Separate Concerns. *Science of Computer Programming*, 11:17–27, 1988.

[55] C. Morgan. The Specification Statement. *Transactions on Programming Languages and Systems*, 10:403–419, 1988.

[56] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[57] C. Morgan and P. Gardiner. Data Refinement by Calculation. Technical report, Oxford University Computing Laboratory, 1988.

[58] F. Morris. Advice on Structuring Compilers and Proving them Correct. In *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, 1973.

[59] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9:287–306, 1987.

[60] J. M. Morris. Invariance Theorems for Recursive Procedures. Technical report, University of Glasgow, June 1988.

[61] J. M. Morris. Laws of Data Refinement. *Acta Informatica*, 26:287–308, 1989.

[62] P. D. Mosses. SIS—Semantics Implementation System. Technical report, Computer Science Department, Aarhus University, DAIMI MD-30, 1979.

[63] P. D. Mosses. An Introduction to Action Semantics. Technical report, Computer Science Department, Aarhus University, DAIMI IR-102, 1991.

[64] G. Nelson and M. Manasse. The Proof of a Second Step of a Factored Compiler. In *Lecture Notes for the International Summer School on Programming and Mathematical Method*, Marktoberdorf, Germany 1990.

[65] J. Palsberg. A Provably Correct Compiler Generator. Technical report, Computer Science Department, Aarhus University, DAIMI PB-362, 1992.

[66] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[67] W. Polak. *Compiler Specification and Verification*. Springer-Verlag (LNCS 124), 1981.

[68] A. Roscoe and C. A. R. Hoare. The Laws of occam Programming. *Theoretical Computer Science*, 60:177–229, 1988.

[69] A. Sampaio. A Comparative Study of Theorem Provers: Proving Correctness of Compiling Specifications. Technical report, Oxford University Computing Laboratory, PRG-TR-20-90, 1990.

[70] E. A. Scott and K. J. Norrie. Automating Algebraic Structures — A Case Study Involving the Correctness of a Specification for a PL_0 Compiler. Technical report, ProCos Document [RHC ESKN 1/1], 1991.

[71] A. Tarski. On the Calculus of Relations. *Symbolic Logic*, 6:73–89, 1941.

[72] A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. *Pacific Journal of Mathematics*, 5, 1955.

[73] 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.

[74] T. Vickers and P. Gardiner. A Tutorial on B: a Theorem Proving Assistant. Technical report, Oxford University Computing Laboratory, 1988.

[75] M. Wirsing. Algebraic Specification: Semantics, Parameterization and Refinement. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 259–318. Springer-Verlag, 1991.

[76] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[77] H. Yan, J. Goguen, and T. Kemp. Proving Properties of Partial Functions with Sort Constraints. Technical report, Programming Research Group, Oxford University, 1993. To appear.

[78] W. D. Young. A Mechanically Verified Code Generator. *Journal of Automated Reasoning*, 5:493–518, 1989.

Appendix A

A Scheme for Compiling Arrays

In order to illustrate how type constructors can be handled, we suggest a scheme for compiling arrays. We will confine ourselves to one-dimensional arrays. Furthermore, we will illustrate the process using a global array variable a and will assume that it has length l. But the scheme can be easily extended to cover an arbitrary number of multi-dimensional array variables.

We extend the source language to allow two operations on arrays: update of an element and indexing. We adopt the map notation, as described in Chapter 4:

$a := a \oplus \{i \mapsto e\}$	update the ith element of a with e
$a[i]$	yield the ith element of a

Arrays are indexed from 0. Thus given that the length of a is l, its elements are $a[0], \ldots, a[l-1]$. We will assume that indexing a with a value out of the range $0..(l-1)$ will lead to abortion; this avoids the need of runtime tests that indices are in bounds, as the implementation cannot be worse than abort.

In addition to the above, we will use lambda expressions in assignments to the entire array a; but this will be used only for reasoning, and will not be considered a source construction.

Our first task is to extend the symbol table Ψ to include the array variable a.

$$\Phi \stackrel{def}{=} \Psi \cup \{a \mapsto n\}$$
$$\text{where } \forall i : 0..(l-1) \bullet (n+i) \notin \operatorname{ran} \Psi$$

In practice, the symbol table must record the length of static arrays. As we are treating a single array, and are assuming that its length is l, we can stick to the same structure of our simple symbol table which maps identifiers to addresses. The address n associated with a determines the memory location where the element $a[0]$ will be stored; this is usually called the *base* address. The locations for an arbitrary element is calculated from n, using the index as an *offset*. Therefore the location associated with element $a[i]$ is given by $n + i$, for $0 \leq i < l$. The condition on the above definition requires that the addresses allocated for a must not have been used previously. In the case of more than

one array variable, this condition would be extended to each one. Furthermore, an extra condition would be required to guarantee non-overlapping of the addresses allocated to distinct arrays.

We can then define the *encoding* program $\hat{\Phi}_{a,w}$ which *retrieves* the abstract state (formed from the array variable a and the other global variables w) from the concrete store of the target machine.

$$\hat{\Phi}_{a,w} \stackrel{def}{=} \mathsf{var}\, a;\; a := \lambda\, i : 0..(l-1) \bullet \mathsf{M}[i+n];\; \hat{\Psi}_w$$

Recall that $\hat{\Psi}_w$ was defined in Chapter 4 to deal with the global variables w. The following *decoding* program maps the abstract state down to the concrete machine state.

$$\hat{\Phi}^{-1}_{a,w} \stackrel{def}{=} \hat{\Psi}^{-1}_w;\; \mathsf{M} := \mathsf{M} \oplus \lambda\, i : n..(n+l-1) \bullet a[i-n];\; \mathbf{end}\; a$$

The following proposition establishes that the pair $(\hat{\Phi}_{a,w}, \hat{\Phi}^{-1}_{a,w})$ is a simulation.

Proposition A.1 $((\hat{\Phi}_{a,w}, \hat{\Phi}^{-1}_{a,w})$ simulation)

$$\hat{\Phi}_{a,w};\; \hat{\Phi}^{-1}_{a,w} \;=\; \mathsf{skip} \;\sqsubseteq\; \hat{\Phi}^{-1}_{a,w};\; \hat{\Phi}_{a,w}$$

■

We already know how to carry out the data refinement of programs involving ordinary variables; now this is extended to cover the array operations.

Proposition A.2 (data refinement of array operations)
(1) $\hat{\Phi}_{a,w}(a := a \oplus \{i \mapsto e\}) \;\sqsubseteq\; \mathsf{M} := \mathsf{M} \oplus \{\hat{\Phi}_{a,w}(i) + n \mapsto \hat{\Phi}_{a,w}(e)\}$
(2) $\hat{\Phi}_{a,w}(a[i]) \;\sqsubseteq\; \mathsf{M}[\hat{\Phi}_{a,w}(i) + n]$ ■

The remaining task is the simplification of expressions involving arrays.

Proposition A.3 (Simplification of expressions)
If neither A nor t occur in i or e
(1) $(a := a \oplus \{i \mapsto e\}) \;=\; \mathbf{dec}\, \mathsf{A}, t \bullet \mathsf{A} := e;\; t := \mathsf{A};\; \mathsf{A} := i;\; a := a \oplus \{\mathsf{A} \mapsto t\}$
(2) $(\mathsf{A} := a[i]) \;=\; (\mathsf{A} := i;\; \mathsf{A} := a[A])$ ■

By structural induction, it is possible to show that the above two rules, together with the ones given in Chapter 4, are sufficient to simplify an arbitrary expression.

Applying data refinement to a simple assignment of the form

$$a := a \oplus \{\mathsf{A} \mapsto t\}$$

leads to

$$\mathsf{M} := \mathsf{M} \oplus \{\mathsf{A} + n \mapsto \mathsf{M}[\Psi\, t]\}$$

which can be taken as the definition of an update instruction for arrays. If preferred, we can store the value to update the array in a new register, say B, rather than in the auxiliary variable t. In this case, the update instruction would be defined by

$$\mathsf{update\text{--}array\text{--}at}(n) \stackrel{def}{=} \mathsf{M} := \mathsf{M} \oplus \{\mathsf{A} + n \mapsto \mathsf{B}\}$$

Similarly, the assignment

$$\mathsf{A} := a[\mathsf{A}]$$

will eventually be data refined to

$$\mathsf{A} := \mathsf{M}[\mathsf{A} + n]$$

which can be taken as the definition of a read instruction

$$\mathsf{read\text{--}array\text{--}at}(n) \stackrel{def}{=} \mathsf{A} := \mathsf{M}[\mathsf{A} + n]$$

We have actually added the rules for expression simplification to our OBJ3 prototype and performed some reductions. For example,

```
OBJ> reduce a := a ⊕ {(a[i bop j]) bop k -> a[dop i]}
rewrites: 178
result Prog: dec A,B • A := i ; A := dop A ; A := a[A] ;
                       B := A ; A := i ; A := A bop j ;
                       A := a[A] ; A := A bop k ;
                       a := a ⊕ {A -> B}
```

The original assignment updates array a at position `(a[i bop j]) bop k` with the value of `a[dop i]`. The resulting program declares variables A and B to play the roles of two registers. Before the final assignment, A and B hold, respectively, the value of the index and the element to update array a.

In the case of arrays local to a recursive program, a similar scheme could be adopted; but storage would be allocated in the runtime stack, rather than in the fixed memory M.

Appendix B

Proof of Lemma 5.1

The proof of Lemma 5.1 uses the following abbreviations, as introduced in Chapter 5.

$$MID \;=\; v, \bar{v} : [a_0 \wedge \mathsf{empty}\,\bar{v},\; S,\; c_0 \wedge \mathsf{empty}\,\bar{v}]$$

$$\text{where} \quad S \;=\; \begin{pmatrix} & b \rightarrow (v :\in r;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & (c_0 \wedge \neg \mathsf{empty}\,\bar{v}) \rightarrow \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \rightarrow p \end{pmatrix}$$

$$T \;=\; \begin{pmatrix} & a \rightarrow (v :\in c;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & b \rightarrow (v :\in r;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & c_0 \rightarrow \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \rightarrow p \end{pmatrix}$$

$$U \;=\; \begin{pmatrix} & b \rightarrow (v :\in r;\; \mathsf{push}(v, \bar{v});\; v :\in a_0) \\ \Box & (c_0 \wedge \#\bar{v} > 1) \rightarrow \mathsf{pop}(v, \bar{v}) \\ \Box & b_0 \rightarrow p \end{pmatrix}$$

We also need the following lemma. It establishes that a normal form program with guarded command set S (operating initially on an empty sequence) is refined by a normal form program obtained from this one, by replacing S with U and the empty sequence with a singleton sequence.

Lemma B.1 (Lift of sequence variables) If S and U are as defined above, and k occurs only where explicitly shown below, then

$$\begin{array}{ll} & v, \bar{v} : [(a_0 \wedge \mathsf{empty}\,\bar{v}),\; S,\; (c_0 \wedge \mathsf{empty}\,\bar{v})] \\ \sqsubseteq & \\ & v, \bar{v} : [(a_0 \wedge \bar{v} = \langle k \rangle),\; U,\; (c_0 \wedge \bar{v} = \langle k \rangle)] \end{array}$$

Proof: we can regard the above as a data refinement problem: although the data space of the two programs are apparently the same, note that the right-hand side requires the

stack $\bar{v}$ to be non-empty. Therefore, in order to compare the above programs we define a simulation.

$$\text{Let } \Theta \stackrel{def}{=} \mathsf{var}\ \bar{w}, w;\ (\neg \mathsf{empty}\ \bar{v})_\perp;\ (\bar{w}, w) := (\mathsf{front}\ \bar{v}, \mathsf{last}\ \bar{v});\ \mathsf{end}\ \bar{v}$$

$$\text{and } \Theta^{-1} \stackrel{def}{=} \mathsf{var}\ \bar{v};\ \bar{v} := \bar{w} \frown \langle w \rangle;\ \mathsf{end}\ \bar{w}, w$$

As before, we use $\Theta(p)$ to denote $\Theta;\ p;\ \Theta^{-1}$. The following facts are true of Θ and Θ^{-1}:

(1) (Θ, Θ^{-1}) is a simulation.
(2) $\Theta(\mathsf{push}(v, \bar{w})) \sqsubseteq \mathsf{push}(v, \bar{v})$
(3) $\Theta(\mathsf{pop}(v, \bar{w})) \sqsubseteq \mathsf{pop}(v, \bar{v})$
(4) $\Theta((\neg \mathsf{empty}\ \bar{w})^\top) \sqsubseteq (\#\bar{v} > 1)^\top$
(5) Let $d_1 = (b \vee (c_0 \wedge \neg \mathsf{empty}\ \bar{w}) \vee b_0)$ and $d_2 = (b \vee (c_0 \wedge \#\bar{v} > 1) \vee b_0)$. Then $\Theta(d1 * S[\bar{v} \leftarrow \bar{w}]) \sqsubseteq (d_2 * U)$

(1) $\quad \Theta;\ \Theta^{-1}$

$=$ {(end − var skip)(3.19.7), (:= combination)(3.15.4) and (laws of sequences)(5.1.1)}

$\mathsf{var}\ \bar{w}, w;\ (\neg \mathsf{empty}\ \bar{v})_\perp;\ (\bar{w}, w) := (\mathsf{front}\ \bar{v}, \mathsf{last}\ \bar{v});\ \mathsf{end}\ \bar{w}, w$

$\sqsubseteq$ {(end− := final value)(3.19.5), (end − var simulation)(3.19.6) and $b_\perp \sqsubseteq$ skip}

skip

$=$ {(end− := final value)(3.19.5), (end − var skip)(3.19.7) and (void $b_\perp$)(3.16.4)}

$\mathsf{var}\ \bar{v};\ \bar{v} := \bar{w} \frown \langle w \rangle;\ (\neg \mathsf{empty}\ \bar{v})_\perp;\ \mathsf{end}\ \bar{v}$

$=$ {(end − var skip)(3.19.7), (:= combination)(3.15.4) and (laws of sequences)(5.1.1)}

$\Theta^{-1};\ \Theta$

(2) $\quad \Theta(\mathsf{push}(v, \bar{w}))$

$=$ $\Theta;\ \mathsf{push}(v, \bar{w});\ \Theta^{-1}$

$=$ {(end change scope)(3.19.2) and (end − var skip)(3.19.7)}

$\mathsf{var}\ \bar{w}, w;\ (\neg \mathsf{empty}\ \bar{v})_\perp;\ (\bar{w}, w) := (\mathsf{front}\ \bar{v}, \mathsf{last}\ \bar{v});\ \mathsf{push}(v, \bar{w});$
$\bar{v} := \bar{w} \frown \langle w \rangle;\ \mathsf{end}\ \bar{w}, w$

$=$ {(:= combination)(3.15.4) and (end − var simulation)(3.19.6)}

$(\neg \mathsf{empty}\ \bar{v})_\perp;\ \mathsf{push}(v, \bar{v})$

$\sqsubseteq$ {$b_\perp \sqsubseteq$ skip}

$\mathsf{push}(v, \bar{v})$

(3) $\quad$ Similar to (**2**).

(4) Similar to (2).

(5) From (1) – (4) and distributivity of Θ over iteration.

Then we have:

$$
\begin{array}{ll}
 & RHS \\
\sqsupseteq & \{\langle \mathsf{end} - \mathsf{var}\ \mathsf{skip}\rangle(3.19.7) \text{ and } (5)\} \\
 & \mathsf{dec}\ v, \bar{v} \bullet v :\in a_0;\ \bar{v} := \langle k\rangle;\ \mathsf{var}\ \bar{w}, w;\ (\neg\mathsf{empty}\ \bar{v})_\perp; \\
 & (\bar{w}, w) := (\mathsf{front}\ \bar{v}, \mathsf{last}\ \bar{v});\ d_1 * S[\bar{v} \leftarrow \bar{w}];\ \bar{v} := \bar{w} \frown \langle w\rangle; \\
 & \mathsf{end}\ \bar{w}, w;\ (c_0 \wedge \bar{v} = \langle k\rangle)_\perp \\
= & \{\langle \text{void } b_\perp\rangle(3.16.4) \text{ and } \langle := \text{ combination}\rangle(3.15.4)\} \\
 & \mathsf{dec}\ v, \bar{v} \bullet v :\in a_0;\ \bar{v} := \langle k\rangle;\ \mathsf{var}\ \bar{w}, w;\ (\bar{w}, w) := (\langle\rangle, k); \\
 & d_1 * S[\bar{v} \leftarrow \bar{w}];\ \bar{v} := \bar{w} \frown \langle w\rangle;\ \mathsf{end}\ \bar{w}, w;\ (c_0 \wedge \bar{v} = \langle k\rangle)_\perp \\
= & \{\langle := \text{ combination}\rangle(3.15.4),\ \langle \mathsf{end}- := \text{ final value}\rangle(3.19.5) \text{ and} \\
 & \ \langle \mathsf{end} - \mathsf{var} \text{ simulation}\rangle(3.19.6)\} \\
 & \mathsf{dec}\ v, \bar{v} \bullet v :\in a_0;\ \bar{v} := \langle k\rangle;\ \mathsf{var}\ \bar{w};\ \bar{w} := \langle\rangle; \\
 & d_1 * S[\bar{v} \leftarrow \bar{w}];\ \bar{v} := \bar{w} \frown \langle k\rangle;\ \mathsf{end}\ \bar{w};\ (c_0 \wedge \bar{v} = \langle k\rangle)_\perp \\
= & \{\langle := -b_\perp \text{ commutation}\rangle(3.15.8) \text{ and } \langle \mathsf{dec}- := \text{ final value}\rangle(3.18.6)\} \\
 & \mathsf{dec}\ v, \bar{v} \bullet v :\in a_0;\ \bar{v} := \langle k\rangle;\ \mathsf{var}\ \bar{w};\ \bar{w} := \langle\rangle; \\
 & d_1 * S[\bar{v} \leftarrow \bar{w}];\ (c_0 \wedge \mathsf{empty}\ \bar{w})_\perp;\ \mathsf{end}\ \bar{w} \\
\sqsupseteq & \{\langle \mathsf{dec}- :\in \text{ initial value}\rangle(3.18.5) \text{ and } \langle \mathsf{dec} \text{ elim}\rangle(3.18.3)\} \\
 & \mathsf{dec}\ v \bullet v :\in a_0;\ \mathsf{var}\ \bar{w};\ \bar{w} := \langle\rangle;\ d_1 * S[\bar{v} \leftarrow \bar{w}];\ (c_0 \wedge \mathsf{empty}\ \bar{w})_\perp;\ \mathsf{end}\ \bar{w} \\
= & \{\langle \mathsf{dec} - (\mathsf{var}, \mathsf{end}) \text{ conversion}\rangle(3.19.10) \text{ and } \langle \mathsf{dec} \text{ rename}\rangle(3.18.4)\} \\
 & LHS
\end{array}
$$

■

Now we can prove Lemma 5.1. First we repeat the inequation to be proved. Let $d_1 = (b \vee (c_0 \wedge \neg\mathsf{empty}\ \bar{v}) \vee b_0)$, then we have:

$$b^\top;\ d_1 * S\ \sqsupseteq\ (\mathsf{empty}\ \bar{v})_\perp;\ MID;\ v :\in r;\ d_1 * S$$

Proof:

$$
\begin{array}{ll}
 & b^\top;\ d_1 * S \\
\sqsupseteq & \{\langle * - \square \text{ unfold}\rangle(3.17.3) \text{ and } b^\top \sqsupseteq \mathsf{skip}\} \\
 & v :\in r;\ \mathsf{push}(v, \bar{v});\ v :\in a_0;\ d_1 * S \\
= & \{\langle * \text{ sequence}\rangle(3.17.7) \text{ and Let } d_2 = (b \vee (c_0 \wedge \#\bar{v} > 1) \vee b_0)\}
\end{array}
$$

$v :\in r;\ \mathsf{push}(v, \bar{v});\ v :\in a_0;\ d_2 * U;\ d_1 * S$

$\sqsupseteq$ {$b_\perp \sqsubseteq \mathsf{skip}$ and $(* - \square$ unfold)(3.17.3)}

$v :\in r;\ \mathsf{push}(v, \bar{v});\ v :\in a_0;\ d_2 * U;\ (c_0 \wedge \#\bar{v} > 1)_\perp;\ \mathsf{pop}(v, \bar{v});\ d_1 * S$

$=$ {(dec introduction)(3.19.11) and Definition 4.1(Normal form)}

$v :\in r;\ \mathsf{push}(v, \bar{v});\ v : [a_0,\ U,\ (c_0 \wedge \#\bar{v} > 1)];\ \mathsf{pop}(v, \bar{v});\ d_1 * S$

$=$ {(dec rename)(3.18.4), assuming w is fresh and convention $t' = t[v \leftarrow w]$}

$v :\in r;\ \mathsf{push}(v, \bar{v});\ w : [a_0',\ U',\ (c_0' \wedge \#\bar{v} > 1)];\ \mathsf{pop}(v, \bar{v});\ d_1 * S$

$\sqsupseteq$ {$(c_0' \wedge \#\bar{v} > 1)_\perp \ \sqsupseteq\ (c_0' \wedge \#\bar{v} > 1)_\perp;\ (\bar{v} = \langle v \rangle)_\perp \ =\ (c_0' \wedge \bar{v} = \langle v \rangle)_\perp$}

$v :\in r;\ \mathsf{push}(v, \bar{v});\ w : [a_0',\ U',\ (c_0' \wedge \bar{v} = \langle v \rangle)];\ \mathsf{pop}(v, \bar{v});\ d_1 * S$

$=$ {Definition of push and pop}

$v :\in r;\ \bar{v} := \langle v \rangle \frown \bar{v};\ w : [a_0',\ U',\ (c_0' \wedge \bar{v} = \langle v \rangle)];\ \bar{v} := \langle\rangle;\ d_1 * S$

$\sqsupseteq$ {$b_\perp \sqsubseteq \mathsf{skip}$}

$v :\in r;\ (\mathsf{empty}\ \bar{v})_\perp;\ \bar{v} := \langle v \rangle;\ w : [a_0',\ U',\ (c_0' \wedge \bar{v} = \langle v \rangle)];\ \bar{v} := \langle\rangle;\ d_1 * S$

$=$ {(dec introduction)(3.19.11) and $(\mathsf{empty}\ \bar{v})_\perp;\ \bar{v} := \langle\rangle \ =\ (\mathsf{empty}\ \bar{v})_\perp$}

$v :\in r;\ (\mathsf{empty}\ \bar{v})_\perp;\ w, \bar{v} : [(a_0' \wedge \bar{v} = \langle v \rangle),\ U',\ (c_0' \wedge \bar{v} = \langle v \rangle)];\ d_1 * S$

$\sqsupseteq$ {Lemma B.1(Lift of sequence variables) and (dec rename)(3.18.4)}

$v :\in r;\ (\mathsf{empty}\ \bar{v})_\perp;\ MID;\ d_1 * S$

$\sqsupseteq$ {($x :\in b;\ p$ commute)(3.16.9)}

$(\mathsf{empty}\ \bar{v})_\perp;\ MID;\ v :\in r;\ d_1 * S$

■

Appendix C

Specification and Verification in OBJ3

Here we give further details about the mechanisation. Following the same structure as that of Chapter 6, we give the complete description of the main modules together with the automated proofs of some of the theorems.

C.1 The Reasoning Language

The reasoning language and its algebraic laws (including the derived ones) are described by the following theory. The next two sections illustrate the verification of two laws of while.

```
th REASONING-LANGUAGE [X :: EXP] is

  sorts Prog ProgId .
  subsorts ProgId < Prog .

  define ListVar is LIST[Var] .
  define ListExp is LIST[Exp] .

*** Program constructs
  op skip : -> Prog .
  op dec _ •_ : ListVar Prog -> Prog [prec 60] .
  op-as _:=_ : ListVar ListExp -> Prog
        for x := e if (len x == len e) and (disj x) [prec 62] .
  op _;_ : Prog Prog -> Prog [assoc prec 56] .
  op _⊓_ : Prog Prog -> Prog [assoc comm idem id: ⊤ prec 67] .
  op _◁ _ ▷_ : Prog CondExp Prog -> Prog [prec 58] .
  op _*_ : CondExp Prog -> Prog [prec 54] .

*** Additional reasoning features
  op ⊥ : -> Prog .
  op ⊤ : -> Prog .
  op _⊔_ : Prog Prog -> Prog [assoc comm idem id: ⊥ prec 57] .
  op _⊑_ : Prog Prog -> Bool [prec 70] .
```

```
  op μ _ •_ : ProgId Prog -> Prog [prec 59] .
  op _;̑_ : Prog Prog -> Prog [prec 56] .      *** Inverse of ;
  op _ _ : ProgId ListVar -> ProgId .
  op _ᵀ : CondExp -> Prog .
  op _⊥ : CondExp -> Prog .
  op _→_ : CondExp Prog -> Prog [prec 53] .
  op _□_ : Prog Prog -> Prog [assoc comm idem id: ⊤ prec 57] .
  op _:∈_ : ListVar CondExp -> Prog [prec 52] .
  op var_ : ListVar -> Prog .
  op end_ : ListVar -> Prog .

*** auxiliary operators ***
  op _\_   : ListVar Prog -> Bool [memo] .          *** Non-freeness
  op _\_   : ProgId  Prog -> Bool [memo] .
  op _\\_   : ListVar Prog -> Bool [memo] .         *** Non-occurrence
  op-as _[_<-_] : Prog ListVar ListExp  -> Prog     *** substitution
     for p[x <- e] if (len x == len e) and (disj x) [memo] .
  op _[_<-_] : Prog ProgId Prog  -> Prog [memo] .

*** variables declaration (for use in equations)
  var X Y Z : ProgId .
  var p q r : Prog .
  var x y z : ListVar .
  var a b c : CondExp .
  var e f g : ListExp .

*** Sequential composition
[;-skip~Lunit]  eq  (skip ; p) = p .
[;-skip~Runit]  eq  (p ; skip) = p .
[;-⊥~Lzero]     eq  (⊥ ; p) = ⊥ .
[;-⊤~Lzero]     eq  (⊤ ; p) = ⊤ .

*** Demonic nondeterminism
[⊓-⊥~zero] eq  (p ⊓ ⊥) = ⊥ .
[⊓-⊤~unit] eq  (p ⊓ ⊤) = p .

*** The ordering relation
[⊑-⊥~bottom ⊑] eq  ⊥ = p .
[⊑-⊤~top ⊑]    eq  p = ⊤ .
[⊑-⊓~lb ⊑]     eq  (p ⊓ q) = p .
[⊑-⊔~ub ⊑]     eq  p = (p ⊔ q) .

*** Angelic nondeterminism
[⊔-⊥~unit] eq  (p ⊔ ⊥) = p .
[⊔-⊤~zero] eq  (p ⊔ ⊤) = ⊤ .

*** Recursion
[μ~fp]      eq  (μ X • p) = p[X <- (μ X • p)] .
[μ~lfp ⇐]  eq     (μ X • p) ⊑ q = (p[X <- q]) ⊑ q .

*** Strongest inverse of sequential composition
[;-;̑~1]     eq (p ; q ⊑ r) = (p ⊑ r ;̑ q) .
[;-;̑~2 ⊑]  eq  (p ;̑ q) ; q = p .
```

```
*** Assumption and Assertion
[b^T~true~cond]     eq  true^T = skip .
[b_⊥~true~cond]     eq  true_⊥ = skip .
[b^T~false~cond]    eq  false^T = ⊤ .
[b_⊥~false~cond]    eq  false_⊥ = ⊥ .
[b^T~conjunction]   eq  (a^T ; b^T) = (a ∧ b)^T .
[b_⊥~conjunction]   eq  (a_⊥ ; b_⊥) = (a ∧ b)_⊥ .
*** The law of simulation gives rise to many laws
[b^T~void~b_⊥]      eq  (b^T ; b_⊥) = b^T .
[b_⊥~void~b^T]      eq  (b_⊥ ; b^T) = b_⊥ .
[b^T~skip ⊑]        eq  skip = b^T .
[b_⊥~skip ⊑]        eq  b_⊥ = skip .
[b_⊥-b^T~sim1 ⊑]    eq  (b_⊥ ; b^T) = skip .
[b_⊥-b^T~sim2 ⊑]    eq  skip = (b^T ; b_⊥) .

*** Guarded command
[→~def]              eq  b → p = (b^T ; p) .
[→~true~guard]       eq  (true → p) = p .
[→~false~guard]      eq  (false → p) = ⊤ .
[→-b^T~conversion]   eq  b → p = (b^T ; p) .
[→~conjunction]      eq  a → (b → p) = (a ∧ b) → p .
[→~disjunction]      eq  (a → p) ⊓ (b → p) = (a ∨ b)→ p .
[→-⊓~dist]           eq  b → (p ⊓ q) = (b → p) ⊓ (b → q) .
[;-→~Ldist]          eq  (b → p) ; q = b → (p ; q) .

*** Guarded command set
[□~def]     eq  (p □ q) = (p ⊓ q) .
[□~elim]    eq  a → (a → p □ b → q) = (a → p) .

*** Conditional
[◁ ▷~def]            eq  (p ◁ b ▷ q) = (b → p □ ¬b → q) .
[◁ ▷~true~cond]      eq  (a ∧ b)^T ; (p ◁ b ∨ c ▷ q) = (a ∧ b)^T ; p .
[◁ ▷~false~cond]     eq  (a ∧ ¬b)^T ; (p ◁ b ∧ c ▷ q) = (a ∧ ¬b)^T ; q .
[◁ ▷~void~b^T~1]     eq  (b^T ; p ◁ b ▷ q) = (p ◁ b ▷ q) .
[◁ ▷~void~b^T~2]     eq  (p ◁ b ▷ ¬b^T ; q) = (p ◁ b ▷ q) .
[◁ ▷~idemp]          eq  (p ◁ b ▷ p) = p .
[;-◁ ▷~Ldist]        eq  (p ◁ b ▷ q) ; r = (p ; r ◁ b ▷ q ; r) .
[guard-◁ ▷~dist]     eq  a → (p ◁ b ▷ q) = (a → p) ◁ b ▷ (a → q) .
[◁ ▷~cond~disj]      eq  p ◁ b ▷ (p ◁ c ▷ q) =  (p ◁ b ∨ c ▷ q) .
[◁ ▷~cond~conj]      eq  (p ◁ b ▷ q) ◁ c ▷ q = (p ◁ b ∧ c ▷ q) .

*** Assignment
[:=~skip]            eq  (x := x) = skip .
[:=~identity]        eq  (x,y := e,y) = (x := e) .
[:=~sym]             eq  (x,y := e,f) = (y,x := f,e) .
[:=~combination]     eq  (x := e ; x := f) = (x := f[x <- e]) .
[→-:=~subst]         eq  (x = e) → (y := f) = (x = e) → (y := f[x <- e]) .
[:=-⊓~Rdist]         eq  x := e ; (p ⊓ q) =  (x := e ; p) ⊓ (x := e ; q) .
[:=-◁ ▷~Rdist]       eq  x := e ; (p ◁ b ▷ q) = (x := e ; p) ◁ b[x <- e] ▷ (x := e ; q) .
*** The following 3 laws are not in Chapter 3
[:=~combination2]    cq  (x := e ; y := f) = (x,y := e,(f[x <- e])) if x \\ y .
[:=~commute]         cq  (x := e ; y := f) = (y := f[x <- e] ; x := e)  if y \\ (x := e) .
[:=~split]           cq  (x,y := e,f) = (x := e ; y := f)  if x \\ (y := f) .
```

```
*** Generalised assignment
[:∈~false~cond]    eq  (x :∈ false) = ⊤ .
[:∈~true~cond ⊑]   eq  (x :∈ true) = skip .
[:∈~void~b^⊤]      eq  x :∈ a ; a^⊤ = x :∈ a .
[:∈~void~b_⊥]      eq  x :∈ a ; a_⊥ = x :∈ a .
[:∈-b^⊤ ⊑]         eq  (x :∈ b) = b^⊤ .
[:∈-◁▷~Rdist]      eq  x :∈ a ; (p ◁ b ▷ q) = (x :∈ a ; p ◁ b ▷ x :∈ a ; q) .
[:∈-:=]            cq  x :∈ (x = e) = (x := e) if x \\ e .

*** Iteration
[*~def]         cq  b * p = μ X • (p ; X ◁ b ▷ skip) if X \ p .
[*~fp]          eq  b * p = (p ; (b * p)) ◁ b ▷ skip .
[*~lfp ⇐]       eq  (b * p ⊑ q) = (p ; q ◁ b ▷ skip ⊑ q) .
[*~elim1]       cq  x :∈ a ; b * p = x :∈ a if pwd(a,b) .
[*~elim2]       cq  a^⊤ ; b * p = a^⊤ if pwd(a,b) .
[*~unfold]      eq  x :∈ a ; (a ∨ b) * p = x :∈ a ; p ; (a ∨ b) * p .
[*-□~unfold]    cq  x :∈ a ; (a ∨ b) * (a → p □ b → q) =
                      x :∈ a ; p ; (a ∨ b) * (a → p □ b → q) if pwd(a,b) .
[*-□~elim]      cq a * (a → p □ b → q) = a * p if pwd(a,b) .
[*-→~elim]      eq a * (a → p) = a * p .
[*-μ~tail~rec]  eq (b * p) ; q = μ X • (p ; X ◁ b ▷ q) .
[*~sequence]    eq (b * p) ; (b ∨ c) * p = (b ∨ c) * p .

*** Static Declaration
[dec~assoc]        cq  dec x • (dec y • p) = (dec x,y • p) if x \\ y .
[dec~sym]          eq  dec x • (dec y • p) = dec y • (dec x • p) .
[dec~elim]         cq  (dec x • p) = p if x \ p .
[dec~rename]       cq  (dec x • p) = (dec y • p[x <- y])
                         if y \ p and contiguous-scope(x,p) .
[dec-:=~init ⊑]    eq  (dec x • p) = (dec x • x := e ; p) .
[dec-:∈~init ⊑]    eq  (dec x • p) = (dec x • x :∈ b ; p) .
[dec-:=~final]     eq  (dec x • p) = (dec x • p ; x := e) .
[dec-:∈~final ⊑]   eq  (dec x • p) = (dec x • p ; x :∈ b) .
[;-dec~Ldist]      cq  (dec x • p) ; q = (dec x • p ; q) if x \ q .
[;-dec~Rdist]      cq  q ; (dec x • p) = (dec x • q ; p) if x \ q .
[dec-;~dist ⊑]     eq  (dec x • p) ; (dec x • q) = (dec x • p ; q) .
[dec-□~dist]       cq  a → (dec x • p) □ b → (dec x • q) =
                       (dec x • a → p □ b → q) if x \\ (a,b) .
[dec-◁▷~dist]  cq  (dec x • p) ◁ b ▷ (dec x • q) = (dec x • p ◁ b ▷ q) if x \\ b .
[dec-*~dist ⊑]     cq  b * (dec x • p) = (dec x • b * p) if x \\ b .

*** Dynamic Declaration
[var~assoc]          cq  var x ; var y = var x,y if x \\ y .
[end~assoc]          cq  end x ; end y = end x,y if x \\ y .
[var~change~scope]   cq  (p ; var x) = (var x ; p) if x \ p .
[end~change~scope]   cq  (end x ; p) = (p ; end x) if x \ p .
[var-◁▷~Rdist]     cq  (var x ; p) ◁ b ▷ (var x ; q) = var x ; (p ◁ b ▷ q) if x \\ b .
[end-◁▷~Rdist]     cq  (end x ; p) ◁ b ▷ (end x ; q) = end x ; (p ◁ b ▷ q) if x \\ b .
[var-:=~init ⊑]      eq  var x = (var x ; x := e) .
[var-:∈~init ⊑]      eq  var x = (var x ; x :∈ b) .
[end-:=~final]       eq  end x = (x := e ; end x) .
[end-:∈~final ⊑]     eq  end x = (x :∈ e ; end x) .
[end-var~sim1]       eq  (var x ; end x) = skip .
```

```
[end-var~sim2 □]    eq  (end x ; var x) = skip .
[end-var~skip]      cq  (end x ; var x ; x := e) = (x := e) if x \\ e .
[var~elim1]        eq  (dec x • var x ; p) = (dec x • p) .
[end~elim1]        eq  (dec x • end x ; p) = (dec x • p) .
[var~elim2]        eq  (dec x • p ; var x) = (dec x • p) .
[end~elim2]        eq  (dec x • p ; end x) = (dec x • p) .
[dec-var~end]       cq  (dec x • p) = (var x ; p ; end x)
                        if contiguous-scope(x,p) and block-structured(p,x) .

endth
```

C.1.1 Proof of Theorem 3.17.6

Here we give the complete mechanisation of the proof of the equation

```
[•-μ~tail~rec] eq (b • p) ; q = μ X • (p ; X ◁ b ▷ q) .
```

which actually relies on the assumption that X is not free in p or q. Most of the features of OBJ3 used in the proof have been introduced before. Others are explained as the need arises.

First we need to specify the context in which the reasoning will be conducted. In this case, it is given by the module describing the reasoning language together with the assumptions and abbreviations used in the proof. As rewriting in OBJ3 is carried out only for *ground* terms (that is, terms involving only constants), we need to explicitly declare a constant to play the role of each variable appearing in the equation. Of course, we must not assume anything about these constants, except the conditions associated with the theorem. The context of our proof is defined by (this is omitted for the next proofs):

```
open REASONING-LANGUAGE[EXP] .
op X : -> ProgId .
op p : -> Prog .
op q : -> Prog .
op b : -> CondExp .
[hyp1] eq X \ p = true .
[hyp2] eq X \ q = true .
[LHS~def] let LHS = (b • p) ; q .
[RHS~def] let RHS = μ X • (p ; X ◁ b ▷ q) .
```

We also add an equation (for later use) which expresses a simple lemma about substitution. It can be automatically proved from the definition of the substitution operator:

```
[subst1]   eq (p ; X ◁ b ▷ q)[X <- LHS] = (p ; LHS ◁ b ▷ q) .
***> Proof
OBJ> reduce in SUBST : (p ; X ◁ b ▷ q)[X <- LHS] == (p ; LHS ◁ b ▷ q)
rewrites: 4
result Bool: true
```

where SUBST is the module which contains the relevant equations of substitution.

We split the proof of the theorem into two steps: (RHS ⊑ LHS) and (LHS ⊑ RHS). Recall that the symbol - preceding the label of an equation (in an apply command) means that the equation is to be applied in reverse (from right to left).

```
***> Proof of (RHS ⊑ LHS)

OBJ> start (b * p) ; q ⊑ LHS .
==========================================
OBJ> apply .*~fp within term .
result Bool: (p ; b * p ◁ b ▷skip) ; q ⊑ LHS
==========================================
OBJ> apply .;-◁ ▷~Ldist within term .
result Bool: p ; (b * p) ; q ◁ b ▷ skip ; q ⊑ LHS
==========================================
OBJ> apply .;-skip~Lunit within term .
result Bool: p ; (b * p) ; q ◁ b ▷ q ⊑ LHS
==========================================
OBJ> apply -.LHS~def within term .
result Bool: p ; LHS ◁ b ▷ q ⊑ LHS
==========================================
OBJ> apply -.subst1 within term .
result Bool: (p ; X ◁ b ▷ q)[X <- LHS] ⊑ LHS
==========================================
OBJ> apply -.μ~lfp at term .
result Bool: μ X • p ; X ◁ b ▷ q ⊑ LHS
==========================================
OBJ> apply -.RHS~def within term .
result Bool: RHS ⊑ LHS
```

The other part of the proof illustrates ways of selecting a particular subterm. A natural number is used to specify the desired subterm, where the arguments of a given operator are numbered from 1. For example, in LHS ⊑ RHS, (1) selects LHS. Nested selections are specified using the keyword of. The application of a given equation may require the user to provide an explicit instantiation (biding) of some of its variables. This usually happens when applying an equation in reverse and its left-hand side contains variables which do not appear on the right-hand side. The biding for these extra variables is defined using the keyword with, followed by a list of equations of the form *var* = *term*.

```
***> Proof of (LHS ⊑ RHS)

OBJ> start μ X • p ; X ◁ b ▷ q ⊑ RHS .
==========================================
OBJ> apply .μ~fp at (1) .
result Bool: (p ; X ◁ b ▷ q)[X <- μ X • p ; X ◁ b ▷ q] ⊑ RHS
==========================================
OBJ> select SUBST .                        *** Contains equations for substitution
==========================================
OBJ> apply red at (1) .                    *** Does the substitution automatically
result Bool: p ; (μ X • p ; X ◁ b ▷ q) ◁ b ▷ q ⊑ RHS
==========================================
OBJ> select REASONING-LANGUAGE[EXP] . *** Back to context providing the laws
==========================================
OBJ> apply -.RHS~def within term .
result Bool: p ; RHS ◁ b ▷ q ⊑ RHS
==========================================
OBJ> apply -.;-⊔;~2 with p = RHS within (1) .
result Bool: p ; (RHS ⊔; q) ; q ◁ b ▷ q ⊑ RHS
==========================================
```

```
OBJ> apply -.;-skip~Lunit at (3) of (1) .
result Bool: p ; (RHS ∪; q) ; q ◁ b ▷ skip ; q ⊑ RHS
========================================
OBJ> apply -.;-◁ ▷~Ldist within term .
result Bool: (p ; (RHS ∪; q) ◁ b ▷ skip) ; q ⊑ RHS
========================================
OBJ> apply .;-∪;~1 at term .
result Bool: p ; (RHS ∪; q) ◁ b ▷ skip ⊑ RHS ∪; q
========================================
OBJ> apply -.*~lfp at term .
result Bool: b * p ⊑ RHS ∪; q
========================================
OBJ> apply -.;-∪;~1 at term .
result Bool: b * p ; q ⊑ RHS
========================================
OBJ> apply -.LHS~def within term .
result Bool: LHS ⊑ RHS
```

C.1.2 Proof of Theorem 3.17.7

The equation to be verified is

```
[*~sequence]    eq (b * p) ; (b ∨ c) * p = (b ∨ c) * p .
```

The proof strategy is very similar to the that of the last section. First we prove a simple lemma about substitution (for later use)

```
[subst1] eq (p ; X ◁ b ▷ (b ∨ c) * p)[X <- RHS] = (p ; RHS ◁ b ▷ (b ∨ c) * p) .
***> Proof
OBJ> reduce in SUBST : (p ; X ◁ b ▷ (b ∨ c) * p)[X <- RHS] ==
                       (p ; RHS ◁ b ▷ (b ∨ c) * p) .
rewrites: 4
result Bool: true
```

and then split the proof in two parts, as above.

```
***> Proof of (LHS ⊑ RHS)

OBJ> start (b ∨ c) * p ⊑ RHS .
========================================
OBJ> apply -.◁ ▷~idemp at (1) .
result Bool: (b ∨ c) * p ◁ b ▷ (b ∨ c) * p ⊑ RHS
========================================
OBJ> apply .*~fp at (1) of (1) .
result Bool: (p ; (b ∨ c) * p ◁ b ∨ c ▷ skip) ◁ b ▷ (b ∨ c) * p ⊑ RHS
========================================
OBJ> apply -.RHS~def within (1) of (1) .
result Bool: (p ; RHS ◁ b ∨ c ▷ skip) ◁ b ▷ (b ∨ c) * p ⊑ RHS
========================================
OBJ> apply -.◁ ▷~void~b⊤~2 at (1) of (1) .
result Bool: (p ; RHS ◁ b ∨ c ▷ ¬(b ∨ c)⊤ ; skip) ◁ b ▷ (b ∨ c) * p ⊑ RHS
```

```
=========================================
OBJ> apply -.*~elim2 with b = (b ∨ c) within term .
result Bool: (p ; RHS ◁ b ∨ c ▷ ¬(b ∨ c)⊤ ; (b ∨ c) * p ; skip) ◁ b ▷
             (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply .;-skip~Runit within term .
result Bool: (p ; RHS ◁ b ∨ c ▷ ¬(b ∨ c)⊤ ; (b ∨ c) * p) ◁ b ▷
             (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply .◁▷~void~b⊤~2 within term .
result Bool: (p ; RHS ◁ b ∨ c ▷ (b ∨ c) * p) ◁ b ▷ (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply .◁▷~cond~conj within term .
result Bool: p ; RHS ◁ (b ∨ c) ∧ b ▷ (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply red at (2) of (1) .
result Bool: p ; RHS ◁ b ▷ (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply -.subst1 within term .
result Bool: (p ; X ◁ b ▷ (b ∨ c) * p)[X <- RHS] ⊑ RHS
=========================================
OBJ> apply -.μ~lfp at term .
result Bool: μ X • p ; X ◁ b ▷ (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply -.*-μ~tail~rec within term .
result Bool: b * p ; (b ∨ c) * p ⊑ RHS
=========================================
OBJ> apply -.LHS~def within term .
result Bool: LHS ⊑ RHS
```

The above result is added to the system in the form of an "inequation"

```
[LHS~⊑~RHS ⊑] eq LHS = RHS .
```

which is then used in the other part of the proof.

```
***> Proof of (RHS ⊑ LHS)

OBJ> start (b * p) ; (b ∨ c) * p ⊑ LHS .
=========================================
OBJ> apply .*~fp at (1) of (1) .
result Bool: (p ; b * p ◁ b ▷ skip) ; (b ∨ c) * p ⊑ LHS
=========================================
OBJ> apply .;-◁▷~Ldist within term .
result Bool: p ; b * p ; (b ∨ c) * p ◁ b ▷ skip ; (b ∨ c) * p ⊑ LHS
=========================================
OBJ> apply .;-skip~Lunit within term .
result Bool: p ; b * p ; (b ∨ c) * p ◁ b ▷ (b ∨ c) * p ⊑ LHS
=========================================
OBJ> apply .*~fp at (3) of (1) .
result Bool: p ; b * p ; (b ∨ c) * p ◁ b ▷ (p ; (b ∨ c) * p ◁ b ∨ c ▷ skip)
             ⊑ LHS
=========================================
```

```
OBJ> apply -.LHS~def within term .
result Bool: p ; LHS ◁ b ▷ (p ; (b ∨ c) * p ◁ b ∨ c ▷ skip) ⊑ LHS
==========================================
OBJ> apply -.RHS~def within term .
result Bool: p ; LHS ◁ b ▷ (p ; RHS ◁ b ∨ c ▷ skip) ⊑ LHS
==========================================
OBJ> apply -.LHS~⊑~RHS within term .
result Bool: p ; LHS ◁ b ▷ (p ; LHS ◁ b ∨ c ▷ skip) ⊑ LHS
==========================================
OBJ> apply .◁▷~cond~disj within term .
result Bool: p ; LHS ◁ b ∨ b ∨ c ▷ skip ⊑ LHS
==========================================
OBJ> apply red at (2) of (1) .
result Bool: p ; LHS ◁ c ∨ b ▷ skip ⊑ LHS
==========================================
OBJ> apply -.*~lfp at term .
result Bool: (c ∨ b) * p ⊑ LHS
==========================================
OBJ> apply -.RHS~def within term .
result Bool: RHS ⊑ LHS
```

C.2 The Normal Form

The normal form definition and the related lemmas and theorems are described by the following theory. Note that this module is also independent of an expression language. The next sections illustrate the verification of some of the proofs.

```
th NORMAL-FORM [X :: EXP] is

  protecting REASONING-LANGUAGE[X] .

  op-as (_:[_, _->_, _]) : ListVar CondExp CondExp Prog CondExp  -> Prog
        for v :[a, b -> p, c] if pwd(b,c) [prec 60] .

*** Variables for use in equations
  var p q : Prog .
  var a ao a1 a2 b b1 b2 c co c1 : CondExp .
  var x v : ListVar .
  var e f : ListExp .

[nf^def] eq  v :[a, b -> p, c]  =  dec v • v :∈ a ; b * p ; c⊥ .


[T:skip1 ⊑] cq  skip  =  v :[a, b -> p, a]   if pwd(a,b) .

[L:primitive^commands ⊑]
  cq  p  ;  v :[a, a -> (p ; v :∈ c), c]   if pwd(a,c) and v \ p .

[T:skip2 ⊑] cq  skip  =  v :[a, a -> v :∈ c, c]   if pwd(a,c) .

[T:assignment ⊑]
  cq  (x := e)  =  v :[a, a -> (x := e ; v :∈ c), c]   if pwd(a,c) .

[L:sequential^composition ⊑]
  cq  v :[a, b1 -> p, co]  ;  v :[co, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]  =
       v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]   if pwd(b1,b2) .


[L:eliminate^guarded^command ⊑]
  cq  v :[a, b1 -> p, c]  =  v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]
  if pwd(b1,b2,c) .

[T:sequential^composition ⊑]
  cq  v :[a, b1 -> p, co]  ;  v :[co, b2 -> q, c]  =
       v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]   if pwd(b1,b2,c) .

[L:conditional ⊑]
  cq v :[a1, (a ∨ b1) -> (a → (v :∈ a1 ◁ b ▷ v :∈ a2) □ b1 → p), c]  ◁ b ▷
     v :[a2, (a ∨ b1) -> (a → (v :∈ a1 ◁ b ▷ v :∈ a2) □ b1 → p), c]
   =
     v :[a, (a ∨ b1) -> (a → (v :∈ a1 ◁ b ▷ v :∈ a2) □ b1 → p), c]
   if pwd(a,b1) .

[T:conditional ⊑]
```

```
  cq v :[a1, b1 -> p, c1]  ◁ b ▷ v :[a2, b2 -> q, c]  =
     v :[a, (a ∨ b1 ∨ c1 ∨ b2) ->
            ( a → (v :∈ a1 ◁ b ▷ v :∈ a2) □ b1 → p
             □ c1 → v :∈ c □ b2 → q), c]
   if pwd(a,b1,b2,c1,c) .

[L:void~initial~state ⊑]
  cq v :[a, (co ∨ b) -> (co → v :∈ a □ b → p), c]  =
     v :[co, co -> (v :∈ a □ b → p), c]   if pwd(co,b) .

[T:iteration ⊑]
  cq b * v :[ao, b1 -> p, co]  =
     v :[a, (a ∨ b1 ∨ co) ->
              (a → (v :∈ ao ◁ b ▷ v :∈ c) □ b1 → p □ co → v :∈ a ), c]
   if pwd(a,b1,co,c)  .

endth
```

C.2.1 Proof of Lemma 4.2

Here we verify the inequation

```
[L:sequential~composition ⊑]
  cq  v :[a, b1 -> p, co]  ;  v :[co, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]  =
      v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]   if pwd(b1,b2) .
```

As usual, we assume the hypothesis by adding it in the form of an equation.

```
open NORMAL-FORM [EXP] .
[hyp1] eq pwd(b1,b2) = true .
```

This is used to discharge the disjointness conditions associated with some of the laws. It is possible to tell OBJ3 that we want conditions to be discharged automatically, rather than by applying rules step by step:

```
set reduce conditions on .
```

Then we proceed with the proof.

```
OBJ> start v :[a,b1 -> p,co] ; v :[co,(b1 ∨ b2) -> (b1 → p □ b2 → q),c] .
==========================================
OBJ> apply .nf~def within term .
result Prog: (dec v • v :∈ a ; b1 * p ; co⊥) ;
             (dec v • v :∈ co ; (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥)
==========================================
OBJ> apply .dec-;~dist at term .
result Prog: dec v • v :∈ a ; b1 * p ; co⊥ ; v :∈ co ;
                        (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥
==========================================
OBJ> apply .:∈-b⊤ with b = co within term .
result Prog: dec v • v :∈ a ; b1 * p ; co⊥ ; co⊤ ;
```

```
                      (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥
=========================================
OBJ> apply .b⊥-b⊤-sim1 within term .
result Prog: dec v • v :∈ a ; b1 * p ; skip ; (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥
=========================================
OBJ> apply .;-skip-Runit within term .
result Prog: dec v • v :∈ a ; b1 * p ; (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥
=========================================
OBJ> apply -.*-□-elim with b = b2 within term .
result Prog: dec v • v :∈ a ; b1 * (b1 → p □ b2 → q) ;
                      (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥
=========================================
OBJ> apply .*-sequence within term .
result Prog: dec v • v :∈ a ; (b1 ∨ b2) * (b1 → p □ b2 → q) ; c⊥
=========================================
OBJ> apply -.nf-def at term .
result Prog: v :[a,b1 ∨ b2 -> b1 → p □ b2 → q,c]
```

C.2.2 Proof of Lemma 4.3

The proof of the lemma

```
[L:eliminate-guarded-command ⊑]
  cq  v :[a, b1 -> p, c]  =  v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]
  if pwd(b1,b2,c) .
```

follows directly from the lemma verified in the last section and the one of the reduction theorems for skip, as shown below.

```
open NORMAL-FORM [EXP] .
[hyp1] eq pwd(b1,b2,c) = true .

OBJ> start v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c] .
=========================================
OBJ> apply -.L:sequential-composition with co = c at term .
result Prog: v :[a,b1 -> p,c] ; v :[c,b1 ∨ b2 -> b1 → p □ b2 → q,c]
=========================================
OBJ> apply -.T:skip1 at (2) .
result Prog: v :[a,b1 -> p,c] ; skip
=========================================
OBJ> apply .;-skip-Runit at term .
result Prog: v :[a,b1 -> p,c]
```

C.2.3 Proof of Theorem 4.4

Using the above two lemmas, the proof of the reduction theorem of sequential composition

```
[T:sequential-composition ⊑]
  cq  v :[a, b1 -> p, co]  ;  v :[co, b2 -> q, c]  =
      v :[a, (b1 ∨ b2) -> (b1 → p □ b2 → q), c]   if pwd(b1,b2,c) .
```

is straightforward.

```
open NORMAL-FORM [EXP] .
[hyp1] eq pwd(b1,b2,c) = true .

OBJ> start v : [a , (b1 ∨ b2) -> (b1 → p □ b2 → q), c] .
==========================================
OBJ> apply -.L:sequential~composition at term .
result Prog: v :[a,b1 -> p,co] ; v :[co,b1 ∨ b2 -> b1 → p □ b2 → q,c]
==========================================
OBJ> apply -.L:eliminate~guarded~command with b1 = b2, p = q at (2) .
result Prog: v :[a,b1 -> p,co] ; v :[co,b2 -> q,c]
```

C.3 Simplification of Expressions

The following module groups the theorems related to the simplification of expressions. In order to prove them, we need to instantiate the module describing the reasoning language with the module SOURCE-EXP which describes the expression sub-language of the source language. The module SOURCE-EXP is omitted here; its relevant sorts and operators were given in Chapter 6. Each of the next two sections presents the verification of one theorem related to this phase of compilation.

```
th EXP-SIMPLIFICATION is

  protecting REASONING-LANGUAGE [SOURCE-EXP] .

*** Variables for use in equations ***
  var p q     : Prog .
  var b ė ḟ   : SourceExp .
  var ẋ t     : SourceVar .
  var x y u   : ListVar .

[Introduce^A] cq (ẋ := ė) = (dec A • A := ė ; ẋ := A)
              if (A \\ ẋ) and (A \\ ė) .

[simple^bop] cq (A := ė bop ḟ) = dec t • A := ḟ ; t := A ; A := ė ;A := A bop t
             if (A,t) \\ (ė bop ḟ) and is-not-var(ḟ) .

[simple^bop^optimisation] cq (A := ė bop ẋ) = A := ė ; A := A bop ẋ
                          if A \\ (ė bop ẋ) .

[simple^uop] cq (A := uop ė) = (A := ė ; A := uop A)   if A \\ ė .


[simple^cond1 ⊑] cq (dec x,A • p) ◁ b ▷ (dec x,A • q) =
                     dec x,A • A := b ; (p ◁ A ▷ q)
                 if (x,A) \\ b .

[simple^cond2 ⊑] cq b * (dec x,A • p) = (dec x,A • A := b ; A * (p ; A := b))
                  if (x,A) \\ b .

endth
```

C.3.1 Proof of Theorem 4.3

The proof of the theorem

```
[simple^bop] cq (A := ė bop ḟ) = dec t • A := ḟ ; t := A ; A := ė ;A := A bop t
             if (A,t) \\ (ė bop ḟ) .
```

follows from the basic laws of assignment and declaration. Using a simple derived law to commute two assignments significantly reduces the number of proof steps. Although the equations defining substitution are used in automatic (rather than step by step) reductions, we still need to tell

OBJ3 to do so. Having substitution as a built-in operator (as in the B-tool) would reduce the number of steps of this proof to a half.

A new feature to select subterms is used in this proof. A term involving only associative operators is viewed as a sequence numbered from 1; the form `[n .. m]` selects the subsequence delimited by (and including) the positions `n` and `m`. The hypothesis is encoded in the usual way, and is omitted here.

```
OBJ> start dec t • A := ḟ ; t := A ; A := ė ; A := A bop t .
=========================================
OBJ> apply .:=˜combination within term .
result Prog: dec t • A := ḟ ; t := A ; A := A bop t[A <- ė]
=========================================
OBJ> apply red at (2) of [3] of (2) .
result Prog: dec t • A := ḟ ; t := A ; A := ė bop t
=========================================
OBJ> apply .:=˜commute at [1 .. 2] of (2) .
result Prog: dec t • t := A[A <- ḟ] ; A := ḟ ; A := ė bop t
=========================================
OBJ> apply red at (2) of [1] of (2) .
result Prog: dec t • t := ḟ ; A := ḟ ; A := ė bop t
=========================================
OBJ> apply .:=˜combination within term .
result Prog: dec t • t := ḟ ; A := ė bop t[A <- ḟ]
=========================================
OBJ> apply red at (2) of [2] of (2) .
result Prog: dec t • t := ḟ ; A := ė bop t
=========================================
OBJ> apply .:=˜commute within term .
result Prog: dec t • A := ė bop t[t <- ḟ] ; t := ḟ
=========================================
OBJ> apply red at (2) of [1] of (2) .
result Prog: dec t • A := ė bop ḟ ; t := ḟ
=========================================
OBJ> apply -.dec-:=˜final at term .
result Prog: dec t • A := ė bop ḟ
=========================================
OBJ> apply .dec˜elim at term .
result Prog: A := ė bop ḟ
```

C.3.2 Proof of Theorem 4.5

Here we verify the proof of the theorem

```
[simple˜cond2 ⊑] cq b * (dec x,A • p) = (dec x,A • A := b ; A * (p ; A := b))
                 if (x,A) \\ b .
```

It gives one more example of the use of the (least) fixed point laws. The hypothesis is encoded in the usual way, and is omitted here.

```
OBJ> start dec x,A • A := b ; A * (p ; A := b) ⊑ RHS .
=========================================
```

```
OBJ> apply .•~fp within term .
result Bool: dec x,A • A := b ; (p ; A := b ; A • (p ; A := b) ◁ A ▷ skip)
             ⊑ RHS
==========================================
OBJ> apply .:=-◁▷~Rdist within term .
result Bool: dec x,A • A := b ; p ; A := b ; A • (p ; A := b) ◁ A[A <- b] ▷
             A := b ; skip ⊑ RHS
==========================================
OBJ> apply red at (2) of (2) of (1) .
result Bool: dec x,A • A := b ; p ; A := b ; A • (p ; A := b) ◁ b ▷
             A := b ; skip ⊑ RHS
==========================================
OBJ> apply -.dec-◁▷~dist within term .
result Bool: (dec x,A • A := b ; p ; A := b ; A • (p ; A := b)) ◁ b ▷
             (dec x,A • A := b ; skip) ⊑ RHS
==========================================
OBJ> apply -.dec~assoc within term .
result Bool: (dec x • (dec A • A := b ; p ; A := b ; A • (p ; A := b))) ◁ b ▷
             (dec x • (dec A • A := b ; skip)) ⊑ RHS
==========================================
OBJ> apply -.dec-:=~init within term .
result Bool: (dec x • (dec A • p ; A := b ; A • (p ; A := b))) ◁ b ▷
             (dec x • (dec A • skip)) ⊑ RHS
==========================================
OBJ> apply .dec~assoc within term .
result Bool: (dec x,A • p ; A := b ; A • (p ; A := b)) ◁ b ▷
             (dec x,A • skip) ⊑ RHS
==========================================
OBJ> apply .dec~elim within term .
result Bool: (dec x,A • p ; A := b ; A • (p ; A := b)) ◁ b ▷ skip ⊑ RHS
==========================================
OBJ> apply -.dec-;~dist with p = p within term .
result Bool: (dec x,A • p) ; (dec x,A • A := b ; A • (p ; A := b)) ◁ b ▷
             skip ⊑ RHS
==========================================
OBJ> apply -.RHS~def within term .
result Bool: (dec x,A • p) ; RHS ◁ b ▷ skip ⊑ RHS
==========================================
OBJ> apply -.•~lfp at term .
result Bool: b • (dec x,A • p) ⊑ RHS
==========================================
OBJ> apply -.LHS~def within term .
result Bool: LHS ⊑ RHS
```

C.4 Control Elimination

The following theory illustrates an instantiation of the normal form theorems to deal with our particular target machine. The module NORMAL-FORM is instantiated with the *union* of the modules SOURCE-EXP (describing the expressions of the source language, as discussed above) and ROM-ADDR-EXP which includes sorts and operators to model addresses in ROM, as discussed in Chapter 6. The instantiation of some of the theorems requires some additional transformations, but they are very simple and do not illustrate any interesting point.

```
th CONTROL-ELIMINATION is

  protecting NORMAL-FORM [SOURCE-EXP + ROM-ADDR-EXP] .

  var p q : Prog .
  var s so s1 f fo f1 : Nat .
  var ẋ : Var .
  var ė : Exp .
  var b1 b2 : CondExp .
  var x : ListVar .

[skip~theo ⊑] eq skip = P :[P = s, false -> skip, P = s] .

[:=~theo ⊑] cq (ẋ := ė) = P :[P = s, (P = s) -> (ẋ,P := ė,P + 1), P = s + 1]
            if P \\ ė .

[;~theo ⊑] cq (P :[P = s, b1 -> p, P = fo]) ; (P :[P = fo, b2 -> q, P = f]) =
                P :[P = s, (b1 ∨ b2) -> (b1 → p □ b2 → q),P = f]
            if pwd(b1,b2,(P = f)) .

[if~theo ⊑]
  cq (P :[P = s + 1,  b1 -> p, P = fo]) ◁ A ▷ (P :[P = s1, b2 -> q, P = f]) =
      P :[P = s,((P = s) ∨ b1 ∨ (P = fo) ∨ b2) ->
          (  (P = s) → (P := P + 1 ◁ A ▷ P := s1)
           □ b1 → p □ (P = fo) → (P := f) □ b2 → q), P = f]
  if pwd((P = s),b1,b2,(P = fo),(P = f)) .

[iteration~theo ⊑]
  cq A * (P :[P = s + 1, b1 -> p, P = fo])  =
     P :[P = s,((P = s) ∨ b1 ∨ (P = fo)) ->
             (  (P = s) → (P := P + 1 ◁ A ▷ P := fo + 1)
              □ b1 → p □ (P = fo) → (P := s)), P = fo + 1]
  if pwd((P = s),b1,(P = fo),(P = fo + 1)) .

endth
```

C.5 Data Refinement

The instantiation of the module describing the reasoning language shows the many kinds of expressions used to reason about this phase of compilation. The modules **SYMTAB** and **RAM** defines the symbol table and the memory RAM as instantiations of a generic module describing maps with the usual operations.

Here we omit the declaration of the operators related to this phase of compilation. Only their definition and some of the theorems are described.

```
th DATA-REFINEMENT is

  protecting REASONING-LANGUAGE [SOURCE-EXP + ROM-ADDR-EXP + SYMTAB + RAM] .

*** Definitions
[Ψ̂~simulation] eq Ψ̂_w = var w ; w := M[Ψ[w]] ; end M .

[Ψ̂^-1~cosimulation] eq Ψ̂_w^-1 = var M ; M := M ⊕ {Ψ[w] ↦ w } ; end w .

[Ψ̂~simulation~function] eq Ψ̂_w(p) = Ψ̂_w ; p ; Ψ̂_w^-1 .

[Ψ̂~simulation~as~substitution] eq Ψ̂_w(ė) = ė [w <- M[Ψ[w]]] .


*** Theorems
[Ψ̂-Ψ̂^-1~simulation1 ⊑] eq Ψ̂_w ; Ψ̂_w^-1 = skip .

[Ψ̂-Ψ̂^-1~simulation2] eq Ψ̂_w^-1 ; Ψ̂_w = skip .

*** Piecewise~data~refinement
[Ψ̂-skip~dist ⊑] eq Ψ̂_w(skip) = skip .

[Ψ̂-:=~dist1 ⊑] eq Ψ̂_(ẋ,w)(ẋ := ė) = (M := M ⊕ {Ψ[ẋ] ↦ Ψ̂_(ẋ,w)(ė)}) .

[Ψ̂-:=~dist2 ⊑] cq Ψ̂_w(ẋ := ė) = (ẋ := Ψ̂_w(ė)) if ẋ \\ w .

[Ψ̂-;~dist ⊑] eq Ψ̂_w(p ; q) = Ψ̂_w(p) ; Ψ̂_w(q) .

[Ψ̂-◁ ▷~dist ⊑] eq Ψ̂_w(p ◁ b ▷ q) = Ψ̂_w(p) ◁ Ψ̂_w(b) ▷ Ψ̂_w(q) .

[Ψ̂-*~dist ⊑] eq Ψ̂_w(b * p) = Ψ̂_w(b) * Ψ̂_w(p) .

[introducing~machine~state ⊑]
  cq Ψ̂_w(dec v,P,A • p) = dec P,A • (Ψ ∪ {v ↦ n })_(w,v)(p)
  if disj(v,w) and disj(n,Ψ[w]) .

endth
```

C.5.1 Proof of Theorem 4.10

Below we verify the proof that $(\hat{\Psi}_w, \hat{\Psi}_w^{-1})$ is a simulation.

```
***> Ψ̂_v ; Ψ̂_v^-1 ⊑ skip .

OBJ> start Ψ̂_w ; Ψ̂_w^-1 .
==========================================
OBJ> apply .Ψ̂~simulation within term .
result Prog: var w ; w := M[Ψ[w]] ; end M ; Ψ̂_w^-1
==========================================
OBJ> apply .Ψ̂^-1~cosimulation within term .
result Prog: var w ; w := M[Ψ[w]] ; end M ;
             var M ; M := M ⊕ {Ψ[w] ↦ w} ; end w
==========================================
OBJ> apply .end-var~sim2 within term .
result Prog: var w ; w := M[Ψ[w]] ; skip ; M := M ⊕ {Ψ[w] ↦ w} ; end w
==========================================
OBJ> apply .;-skip~Lunit within term .
result Prog: var w ; w := M[Ψ[w]] ; M := M ⊕ {Ψ[w] ↦ w} ; end w
==========================================
OBJ> apply .:=~combination2 within term .
result Prog: var w ; w,M := M[Ψ[w]],(M ⊕ {Ψ[w] ↦ w}[w <- M[Ψ[w]]]) ; end w
==========================================
OBJ> apply red at (2) of [2] .
result Prog: var w ; w,M := M[Ψ[w]],M ; end w
==========================================
OBJ> apply .:=~identity within term .
result Prog: var w ; w := M[Ψ[w]] ; end w
==========================================
OBJ> apply -.end-:=~final within term .
result Prog: var w ; end w
==========================================
OBJ> apply .end-var~sim1 within term .
result Prog: skip


***> Ψ̂_w^-1 ; Ψ̂_w = skip .

OBJ> start Ψ̂_w^-1 ; Ψ̂_w .
==========================================
OBJ> apply .Ψ̂^-1~cosimulation within term .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w } ; end w ; Ψ̂_w
==========================================
OBJ> apply .Ψ̂~simulation within term .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w } ; end w ;
             var w ; w := M[Ψ[w]] ; end M
==========================================
OBJ> apply .end-var~skip within term .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w } ; w := M[Ψ[w]] ; end M
==========================================
OBJ> apply .:=~combination2 within term .
result Prog: var M ; M,w := (M ⊕ {Ψ[w] ↦ w}),(M[Ψ[w]][M<-M ⊕ {Ψ[w] ↦ w}]);
             end M
==========================================
OBJ> apply red at (2) of [2] .
result Prog: var M ; M,w := (M ⊕ {Ψ[w] ↦ w}),w ; end M
```

```
=========================================
OBJ> apply .:=~identity within term .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w} ; end M
=========================================
OBJ> apply -.end-:=~final within term .
result Prog: var M ; end M
=========================================
OBJ> apply .end-var~sim1 within term .
result Prog: skip
```

C.5.2 Proof of Theorem 4.14

Here we verify the inequation

```
[introducing~machine~state ⊑]
  cq Ψ̂_w(dec v,P,A • p) = dec P,A • Φ̂_(w,v)(p)
  if disj(v,w) and disj(n,Ψ[w]) .
```

where $\Phi = \Psi \cup \{v \mapsto n\}$. The proof uses two lemmas which are also verified below.

```
***> lemma1: Ψ̂_w ; var v ⊑ Φ̂_v,w

OBJ> start Ψ̂_w ; var v .
=========================================
OBJ> apply .Ψ̂~simulation within term .
result Prog: var w ; w := M[Ψ[w]] ; end M ; var v
=========================================
OBJ> apply .var~change~scope at [2 .. 4] .
result Prog: var w ; var v ; w := M[Ψ[w]] ; end M
=========================================
OBJ> apply .var-:=~init with e = M[n] at [2] .
result Prog: var w ; var v ; v := M[n] ; w := M[Ψ[w]] ; end M
=========================================
OBJ> apply .var~change~scope at [ 1 .. 2 ] .
result Prog: var v ; var w ; v := M[n] ; w := M[Ψ[w]] ; end M
=========================================
OBJ> apply .var~assoc within term .
result Prog: var v,w ; v := M[n] ; w := M[Ψ[w]] ; end M
=========================================
OBJ> apply .:=~combination2 within term .
result Prog: var v,w ; v,w := M[n],(M[Ψ[w]][v <- M[n]]) ; end M
=========================================
OBJ> apply red at (2) of [2] .
result Prog: var v,w ; v,w := M[n],M[Ψ[w]] ; end M
=========================================
OBJ> apply -.list~application within term .
result Prog: var v,w ; v,w := M[n,Ψ[w]] ; end M
=========================================
OBJ> apply -.map~lemma1 with x1 = v within term .
result Prog: var v,w ; v,w := M[(Ψ ∪ {v ↦ n })[v,w]] ; end M
=========================================
OBJ> apply -.Φ~def within term .
```

```
result Prog: var v,w ; v,w := M[Φ[v,w]] ; end M
================================================
OBJ> apply -.Ψ̂~simulation within term .
result Prog: Φ̂v,w


***> lemma2: end v ; Ψ̂w^-1 ⊑ Φ̂v,w^-1

OBJ> start end v ; Ψ̂w^-1 .
================================================
OBJ> apply .Ψ̂^-1~cosimulation within term .
result Prog: end v ; var M ; M := M ⊕ {Ψ[w] ↦ w } ; end w
================================================
OBJ> apply .end~change~scope at [ 1 .. 3 ] .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w } ; end v ; end w
================================================
OBJ> apply .end~assoc within term .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w } ; end v,w
================================================
OBJ> apply .var~:=~init with e = M ⊕  n ↦ v   within term .
result Prog: var M ; M := M ⊕ {n ↦ v } ; M := M ⊕ {Ψ[w] ↦ w } ; end v,w
================================================
OBJ> apply .:=~combination within term .
result Prog: var M ; M := M ⊕ {Ψ[w] ↦ w }[M <- M ⊕ {n ↦ v }] ; end v,w
================================================
OBJ> apply red at (2) of [2] .
result Prog: var M ; M := M ⊕ {n,Ψ[w] ↦ v,w } ; end v,w
================================================
OBJ> apply -.map~lemma1 with x1 = v within term .
result Prog: var M ; M := M ⊕ {(Ψ ∪ {v ↦ n })[v,w] ↦ v,w } ; end v,w
================================================
OBJ> apply -.Φ~def within term .
result Prog: var M ; (M := M ⊕ {Φ[v,w] ↦ v,w } ; end v,w)
================================================
OBJ> apply -.Ψ̂^-1~cosimulation within term .
result Prog: Φ̂v,w^-1


***> Therefore we can add inequations representing the two lemmas
[lemma1 ⊑] eq Ψ̂w ; var v = Φ̂v,w .
[lemma2 ⊑] eq end v ; Ψ̂w^-1 = Φ̂v,w^-1 .


***> Proof of the theorem

OBJ> start Ψ̂v(dec v,P,A • p) .
================================================
OBJ> apply .Ψ̂~simulation~function at term .
result Prog: Ψ̂w ; (dec v,P,A • p) ; Ψ̂w^-1
================================================
OBJ> apply -.dec~assoc with x = v within term .
result Prog: Ψ̂w ; (dec v •(dec P,A • p)) ; Ψ̂w^-1
================================================
```

```
OBJ> apply .dec-var~end within term .
result Prog: Ψ̂w ; var v ; (dec P,A • p) ; end v ; Ψ̂w⁻¹
==========================================
OBJ> apply .lemma1 within term .
result Prog: Φ̂v,w ; (dec P,A • p) ; end v ; Ψ̂w⁻¹
==========================================
OBJ> apply .lemma2 within term .
result Prog: Φ̂v,w ; (dec P,A • p) ; Φ̂v,w⁻¹
==========================================
OBJ> apply .;-dec~Rdist within term .
result Prog: (dec P,A • Φ̂v,w ; p) ; Φ̂v,w⁻¹
==========================================
OBJ> apply .;-dec~Ldist within term .
result Prog: dec P,A • Φ̂v,w ; p ; Φ̂v,w⁻¹
==========================================
OBJ> apply -.Ψ̂~simulation~function within term .
result Prog: dec P,A • Φ̂v,w(p)
```

C.6 Machine Instructions

The machine instructions are defined as assignments that update the machine state. Therefore, the instructions should also be regarded as elements of the sort Prog. However, to make it clear that we are introducing a new concept, we declare a subsort of Prog whose elements are the machine instructions.

```
th CODE is

  protecting REASONING-LANGUAGE [SOURCE-EXP + ROM-ADDR-EXP + RAM] .

  sort Instruction .
  subsort Instruction < Prog .

  op load   : RamAddr -> Instruction .
  op store  : RamAddr -> Instruction .
  op bop-A  : RamAddr -> Instruction .
  op uop-A  : -> Instruction .

  op jump   : RomAddr -> Instruction .
  op cjump  : RomAddr -> Instruction .

  var n : RamAddr .
  var j : RomAddr .

  eq (A,P := M[n],P + 1) = load(n) .
  eq (M,P := (M ⊕ {n ↦ A }),P + 1) = store(n) .
  eq (A,P := A bop M[n],P + 1) = bop-A(n) .
  eq (A,P := uop A,P + 1) = uop-A .

  eq (P := j) = jump(j) .
  eq (P := P + 1  ◁ A ▷ jump(j)) = cjump(j) .

endth
```

The reason to order the equations in this way is that they are used as rewrite rules during the compilation process. Therefore, when the assignment statements (used as patterns to define the instructions) are generated, they are automatically replaced by the corresponding instructions names.