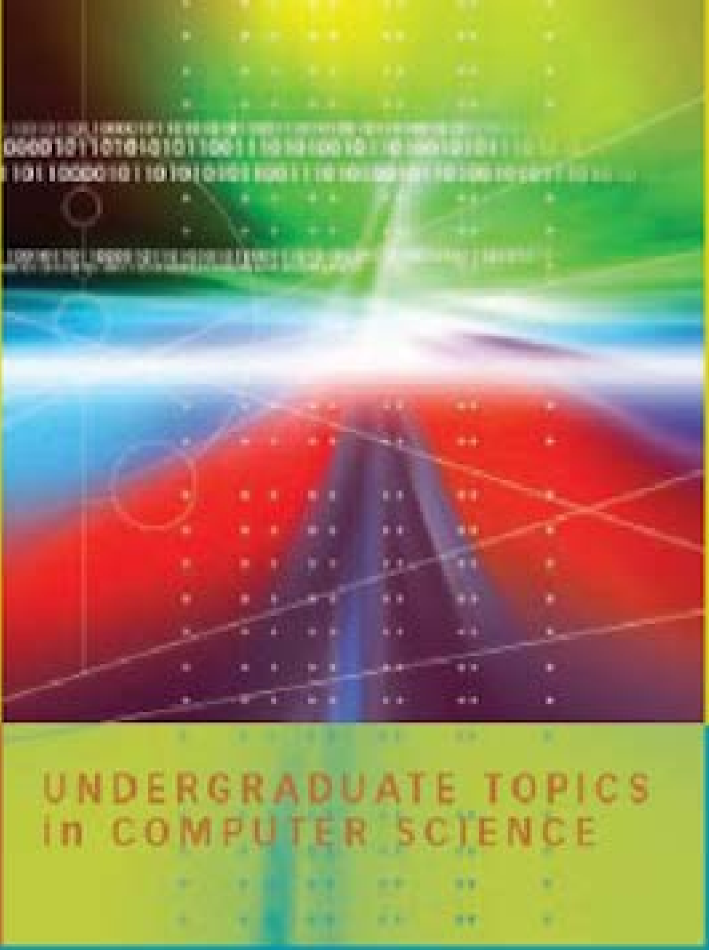


Hanne Riis Nielson
Flemming Nielson

Semantics with Applications

An Appetizer

Springer

UTiCS

Undergraduate Topics in Computer Science

Hanne Riis Nielson and Flemming Nielson

Semantics with Applications: An Appetizer

Springer

Hanne Riis Nielson, PhD
The Technical University of Denmark
Denmark

Flemming Nielson, PhD, DSc
The Technical University of Denmark
Denmark

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2006939147

Undergraduate Topics in Computer Science ISSN 1863-7310
ISBN-10: 1-84628-691-3 e-ISBN-10: 1-84628-692-1
ISBN-13: 978-1-84628-691-9 e-ISBN-13: 978-1-84628-692-6

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

Springer Science+Business Media
springer.com

Preface

This book is written out of a tradition that places special emphasis on the following three approaches to semantics:

– operational semantics,

– denotational semantics, and

– axiomatic semantics.

It is therefore beyond the scope of this introductory book to cover other approaches such as algebraic semantics, game semantics, and evolving algebras.

We strongly believe that semantics has an important role to play in the future development of software systems and domain-specific languages (and hence is not confined to the enormous task of specifying "real life" languages such as C++, Java or C#). We have therefore found the need for an introductory book that

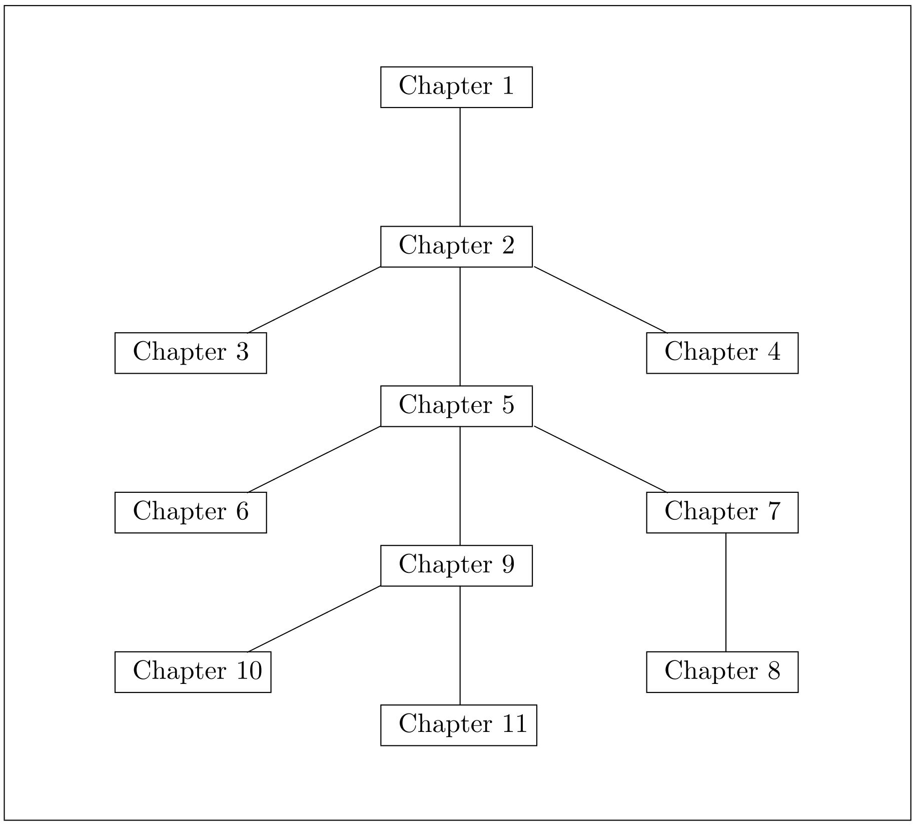
– presents the *fundamental ideas* behind these approaches,

– stresses their *relationship* by formulating and proving the relevant theorems, and

– illustrates the *applications* of semantics in computer science.

This is an ambitious goal for an introductory book, and to achieve it, the bulk of the technical development concentrates on a rather small core language of `while`-programs for which the three approaches are developed to roughly the same level of sophistication; this should enable students to get a better grasp of similarities and differences among the three approaches.

In our choice of applications, we have selected some of the historically important application areas as well as some of the more promising candidates for future applications:



- the use of semantics for validating prototype implementations of programming languages;
- the use of semantics for verifying program analyses that are part of more advanced implementations of programming languages;
- the use of semantics for verifying security analyses; and
- the use of semantics for verifying useful program properties, including information about execution time.

Clearly this only serves as an appetizer to the fascinating area of "Semantics with Applications"; some pointers for further reading are given in Chapter 11.

Overview. As is illustrated in the dependency diagram, Chapters 1, 2, 5, 9, and 11 form the core of the book. Chapter 1 introduces the example language **While** of `while`-programs that is used throughout the book. In Chapter 2 we cover two approaches to *operational semantics*, the natural semantics of

G. Kahn and the structural operational semantics of G. D. Plotkin. Chapter 5 develops the *denotational semantics* of D. Scott and C. Strachey, including simple fixed point theory. Chapter 9 introduces *program verification* based on operational and denotational semantics and goes on to present the *axiomatic approach* due to C. A. R. Hoare. Finally, Chapter 11 contains suggestions for further reading. Chapters 2, 5, and 9 are devoted to the language **While** and cover specification as well as theory; there is quite a bit of attention to the proof techniques needed for proving the relevant theorems.

Chapters 3, 6, and 10 consider extensions of the approach by incorporating new descriptive techniques or new language constructs; in the interest of breadth of coverage, the emphasis is on specification rather than theory. To be specific, Chapter 3 considers extensions with abortion, non-determinism, parallelism, block constructs, dynamic and static procedures, and non-recursive and recursive procedures. In Chapter 6 we consider static procedures that may or may not be recursive and we show how to handle exceptions; that is, certain kinds of jumps. Finally, in Section 10.1 we consider non-recursive and recursive procedures and show how to deal with total correctness properties.

Chapters 4, 7, 8, and 10 cover the applications of operational, denotational, and axiomatic semantics to the language **While** as developed in Chapters 2, 5, and 9. In Chapter 4 we show how to prove the correctness of a simple compiler using the operational semantics. In Chapter 7 we show how to specify and prove the correctness of a program analysis for "Detection of Signs" using the denotational semantics. Furthermore, in Chapter 8 we specify and prove the correctness of a security analysis once more using the denotational semantics. Finally, in Section 10.2 we extend the axiomatic approach so as to obtain information about execution time.

Appendix A reviews the mathematical notation on which this book is based. It is mostly standard notation, but some may find our use of $\hookrightarrow$ and $\diamond$ non-standard. We use $D \hookrightarrow E$ for the set of *partial* functions from D to E; this is because we find that the $D \rightharpoonup E$ notation is too easily overlooked. Also, we use $R \diamond S$ for the composition of binary relations R and S. When dealing with axiomatic semantics we use formulae $\{ \ P \ \} \ S \ \{ \ Q \ \}$ for partial correctness assertions but $\{ \ P \ \} \ S \ \{ \Downarrow Q \ \}$ for total correctness assertions, hoping that the explicit occurrence of $\Downarrow$ (for termination) may prevent the student from confusing the two systems.

Appendix B contains some fairly detailed results for calculating the number of iterations of a functional before it stabilises and produces the least fixed point. This applies to the functionals arising in the program analyses developed in Chapters 7 and 8.

Notes for the instructor. The reader should preferably be acquainted with the BNF style of specifying the syntax of programming languages and should be familiar with most of the mathematical concepts surveyed in Appendix A.

We provide two kinds of exercises. One kind helps the student in understanding the definitions, results, and techniques used in the text. In particular, there are exercises that ask the student to prove auxiliary results needed for the main results but then the proof techniques will be minor variations of those already explained in the text. We have marked those exercises whose results are needed later by "Essential". The other kind of exercises are more challenging in that they extend the development, for example by relating it to other approaches. We use a star to mark the more difficult of these exercises. Exercises marked by two stars are rather lengthy and may require insight not otherwise presented in the book. It will not be necessary for students to attempt all the exercises, but we do recommend that they read them and try to understand what the exercises are about. For a list of misprints and supplementary material, please consult the webpage `http://www.imm.dtu.dk/~riis/SWA/swa.html`.

Acknowledgments. This book grew out of our previous book *Semantics with Applications: A Formal Introduction* [18] that was published by Wiley in 1992 and a note, *Semantics with Applications: Model-Based Program Analysis*, written in 1996. Over the years, we have obtained many comments from colleagues and students, and since we are constantly reminded that the material is still in demand, we have taken this opportunity to rework the book. This includes using shorter chapters and a different choice of security-related analyses. The present version has benefitted from the comments of Henning Makholm.

Kongens Lyngby, Denmark, January 2007 Hanne Riis Nielson

Flemming Nielson

Contents

List of Tables

1 Introduction

The purpose of this book is to describe some of the main ideas and methods used in semantics, to illustrate these on interesting applications, and to investigate the relationship between the various methods.

Formal semantics is concerned with rigorously specifying the meaning, or behaviour, of programs, pieces of hardware, etc. The need for rigour arises because

- it can reveal ambiguities and subtle complexities in apparently crystal clear defining documents (for example, programming language manuals), and
- it can form the basis for implementation, analysis, and verification (in particular, proofs of correctness).

We will use informal set-theoretic notation (reviewed in Appendix A) to represent semantic concepts. This will suffice in this book but for other purposes greater notational precision (that is, formality) may be needed, for example when processing semantic descriptions by machine as in semantics-directed compiler-compilers or machine-assisted proof checkers.

1.1 Semantic Description Methods

It is customary to distinguish between the syntax and the semantics of a programming language. The *syntax* is concerned with the grammatical structure of programs. So a syntactic analysis of the program

$$\texttt{z:=x; x:=y; y:=z}$$

will tell us that it consists of three statements separated by the symbol ';'. Each of these statements has the form of a variable followed by the composite symbol ':=' and an expression that is just a variable.

The *semantics* is concerned with the meaning of grammatically correct programs. So it will express that the meaning of the program above is to exchange the values of the variables `x` and `y` (and setting `z` to the final value of `y`). If we were to explain this in more detail, we would look at the grammatical structure of the program and use explanations of the meanings of

– sequences of statements separated by ';' and

– a statement consisting of a variable followed by ':=' and an expression.

The actual explanations can be formalized in different ways. In this book, we shall consider three approaches. Very roughly, the ideas are as follows:

Operational semantics: The meaning of a construct is specified by the computation it induces when it is executed on a machine. In particular, it is of interest *how* the effect of a computation is produced.

Denotational semantics: Meanings are modelled by mathematical objects that represent the effect of executing the constructs. Thus *only* the effect is of interest, not how it is obtained.

Axiomatic semantics: Specific properties of the effect of executing the constructs are expressed as *assertions*. Thus there may be aspects of the executions that are ignored.

To get a feeling for their different natures, let us see how they express the meaning of the example program above.

Operational Semantics

An operational explanation of the meaning of a construct will tell how to *execute* it:

– To execute a sequence of statements separated by ';', we execute the individual statements one after the other and from left to right.

– To execute a statement consisting of a variable followed by ':=' and another variable, we determine the value of the second variable and assign it to the first variable.

We shall record the execution of the example program in a state where `x` has the value **5**, `y` the value **7**, and `z` the value **0** by the following derivation sequence:

$$\begin{array}{lrl} & \langle \texttt{z:=x; x:=y; y:=z}, & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}]\rangle \\ \Rightarrow & \langle \texttt{x:=y; y:=z}, & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}]\rangle \\ \Rightarrow & \langle \texttt{y:=z}, & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}]\rangle \\ \Rightarrow & & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{5}, \texttt{z}\mapsto\mathbf{5}] \end{array}$$

In the first step, we execute the statement z:=x and the value of z is changed to **5**, whereas those of x and y are unchanged. The remaining program is now x:=y; y:=z. After the second step, the value of x is **7** and we are left with the program y:=z. The third and final step of the computation will change the value of y to **5**. Therefore the initial values of x and y have been exchanged, using z as a temporary variable.

This explanation gives an *abstraction* of how the program is executed on a machine. It is important to observe that it is indeed an abstraction: we ignore details such as the use of registers and addresses for variables. So the operational semantics is rather independent of machine architectures and implementation strategies.

In Chapter 2 we shall formalize this kind of operational semantics, which is often called *structural operational semantics* (or small-step semantics). An alternative operational semantics is called *natural semantics* (or big-step semantics) and differs from the structural operational semantics by hiding even more execution details. In the natural semantics, the execution of the example program in the same state as before will be represented by the derivation tree

$$\frac{\dfrac{\langle \texttt{z:=x}, s_0\rangle \to s_1 \qquad \langle \texttt{x:=y}, s_1\rangle \to s_2}{\langle \texttt{z:=x; x:=y}, s_0\rangle \to s_2} \qquad \langle \texttt{y:=z}, s_2\rangle \to s_3}{\langle \texttt{z:=x; x:=y; y:=z}, s_0\rangle \to s_3}$$

where we have used the abbreviations

$$\begin{array}{rcl} s_0 & = & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}] \\ s_1 & = & [\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}] \\ s_2 & = & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{5}] \\ s_3 & = & [\texttt{x}\mapsto\mathbf{7}, \texttt{y}\mapsto\mathbf{5}, \texttt{z}\mapsto\mathbf{5}] \end{array}$$

This is to be read as follows. The execution of z:=x in the state s_0 will result in the state s_1 and the execution of x:=y in state s_1 will result in state s_2. Therefore the execution of z:=x; x:=y in state s_0 will give state s_2. Furthermore, execution of y:=z in state s_2 will give state s_3 so in total the execution of the program in state s_0 will give the resulting state s_3. This is expressed by

$$\langle \texttt{z:=x; x:=y; y:=z}, s_0\rangle \to s_3$$

but now we have hidden the explanation above of how it was actually obtained.

The operational approaches are introduced in Chapter 2 for the language **While** of `while`-programs and extended to other language constructs in Chapter 3. In Chapter 4 we shall use the natural semantics as the basis for proving the correctness of an implementation of **While**.

Denotational Semantics

In the denotational semantics, we concentrate on the *effect* of executing the programs and we shall model this by mathematical functions:

- The effect of a sequence of statements separated by ';' is the functional composition of the effects of the individual statements.
- The effect of a statement consisting of a variable followed by ':=' and another variable is the function that given a state will produce a new state: it is like the original one except that the value of the first variable of the statement is equal to that of the second variable.

For the example program, we obtain functions written $\mathcal{S}[\![\text{z:=x}]\!]$, $\mathcal{S}[\![\text{x:=y}]\!]$, and $\mathcal{S}[\![\text{y:=z}]\!]$ for each of the assignment statements, and for the overall program we get the function

$$\mathcal{S}[\![\text{z:=x; x:=y; y:=z}]\!] = \mathcal{S}[\![\text{y:=z}]\!] \circ \mathcal{S}[\![\text{x:=y}]\!] \circ \mathcal{S}[\![\text{z:=x}]\!]$$

Note that the *order* of the statements has changed because we use the usual notation for function composition, where $(f \circ g)\ s$ means $f\ (g\ s)$. If we want to determine the effect of executing the program on a particular state, then we can *apply* the function to that state and *calculate* the resulting state as follows:

$$\begin{aligned}
&\mathcal{S}[\![\text{z:=x; x:=y; y:=z}]\!]([\text{x}\mapsto\mathbf{5},\ \text{y}\mapsto\mathbf{7},\ \text{z}\mapsto\mathbf{0}])\\
&\qquad= (\mathcal{S}[\![\text{y:=z}]\!] \circ \mathcal{S}[\![\text{x:=y}]\!] \circ \mathcal{S}[\![\text{z:=x}]\!])([\text{x}\mapsto\mathbf{5},\ \text{y}\mapsto\mathbf{7},\ \text{z}\mapsto\mathbf{0}])\\
&\qquad= \mathcal{S}[\![\text{y:=z}]\!](\mathcal{S}[\![\text{x:=y}]\!](\mathcal{S}[\![\text{z:=x}]\!]([\text{x}\mapsto\mathbf{5},\ \text{y}\mapsto\mathbf{7},\ \text{z}\mapsto\mathbf{0}])))\\
&\qquad= \mathcal{S}[\![\text{y:=z}]\!](\mathcal{S}[\![\text{x:=y}]\!]([\text{x}\mapsto\mathbf{5},\ \text{y}\mapsto\mathbf{7},\ \text{z}\mapsto\mathbf{5}]))\\
&\qquad= \mathcal{S}[\![\text{y:=z}]\!]([\text{x}\mapsto\mathbf{7},\ \text{y}\mapsto\mathbf{7},\ \text{z}\mapsto\mathbf{5}])\\
&\qquad= [\text{x}\mapsto\mathbf{7},\ \text{y}\mapsto\mathbf{5},\ \text{z}\mapsto\mathbf{5}]
\end{aligned}$$

Note that we are only manipulating mathematical objects; we are not concerned with executing programs. The difference may seem small for a program with only assignment and sequencing statements, but for programs with more sophisticated constructs it is substantial. The benefits of the denotational approach are mainly due to the fact that it abstracts away from how programs are executed. Therefore it becomes easier to reason about programs, as it simply amounts to reasoning about mathematical objects. However, a prerequisite for doing so is to establish a firm mathematical basis for denotational semantics, and this task turns out not to be entirely trivial.

The denotational approach and its mathematical foundations are introduced in Chapter 5 for the language **While**; in Chapter 6 we extend it to other language constructs. The approach can easily be adapted to express other properties of programs besides their execution behaviours. Some examples are:

– Determine whether all variables are initialized before they are used — if not, a warning may be appropriate.

– Determine whether a certain expression in the program always evaluates to a constant — if so, one can replace the expression by the constant.

– Determine whether all parts of the program are reachable — if not, they could be removed or a warning might be appropriate.

In Chapters 7 and 8 we develop examples of this.

While we prefer the denotational approach when reasoning about programs, we may prefer an operational approach when implementing the language. It is therefore of interest whether a denotational definition is *equivalent* to an operational definition, and this is studied in Section 5.4.

Axiomatic Semantics

Often one is interested in *partial correctness properties* of programs: A program is partially correct, with respect to a precondition and a postcondition, if whenever the initial state fulfils the precondition and the program terminates, then the final state is guaranteed to fulfil the postcondition. For our example program, we have the partial correctness property

$$\{ \texttt{x}=\texttt{n} \wedge \texttt{y}=\texttt{m} \} \ \texttt{z:=x; x:=y; y:=z} \ \{ \texttt{y}=\texttt{n} \wedge \texttt{x}=\texttt{m} \}$$

where $\texttt{x}=\texttt{n} \wedge \texttt{y}=\texttt{m}$ is the precondition and $\texttt{y}=\texttt{n} \wedge \texttt{x}=\texttt{m}$ is the postcondition. The names `n` and `m` are used to "remember" the initial values of `x` and `y`, respectively. The state $[\texttt{x}\mapsto\mathbf{5}, \texttt{y}\mapsto\mathbf{7}, \texttt{z}\mapsto\mathbf{0}]$ satisfies the precondition by taking $\texttt{n}=\mathbf{5}$ and $\texttt{m}=\mathbf{7}$, and when we have *proved* the partial correctness property we can deduce that *if* the program terminates, *then* it will do so in a state where `y` is **5** and `x` is **7**. However, the partial correctness property does not ensure that the program *will* terminate, although this is clearly the case for the example program.

The axiomatic semantics provides a *logical system* for proving partial correctness properties of individual programs. A proof of the partial correctness property above, may be expressed by the proof tree

$$\dfrac{\dfrac{\{\ p_0\ \}\ \texttt{z:=x}\ \{\ p_1\ \} \qquad \{\ p_1\ \}\ \texttt{x:=y}\ \{\ p_2\ \}}{\{\ p_0\ \}\ \texttt{z:=x; x:=y}\ \{\ p_2\ \}} \qquad \{\ p_2\ \}\ \texttt{y:=z}\ \{\ p_3\ \}}{\{\ p_0\ \}\ \texttt{z:=x; x:=y; y:=z}\ \{\ p_3\ \}}$$

where we have used the abbreviations

$$\begin{aligned} p_0 &= \texttt{x=n} \wedge \texttt{y=m} \\ p_1 &= \texttt{z=n} \wedge \texttt{y=m} \\ p_2 &= \texttt{z=n} \wedge \texttt{x=m} \\ p_3 &= \texttt{y=n} \wedge \texttt{x=m} \end{aligned}$$

We may view the logical system as a specification of only certain aspects of the semantics. It usually does not capture all aspects for the simple reason that all the partial correctness properties listed below can be proved using the logical system, but certainly we would not regard the programs as behaving in the same way:

$\{\ \texttt{x=n} \wedge \texttt{y=m}\ \}\ \texttt{z:=x; x:=y; y:=z}\ \{\ \texttt{y=n} \wedge \texttt{x=m}\ \}$

$\{\ \texttt{x=n} \wedge \texttt{y=m}\ \}\ \texttt{if x=y then skip else (z:=x; x:=y; y:=z)}\ \{\ \texttt{y=n} \wedge \texttt{x=m}\ \}$

$\{\ \texttt{x=n} \wedge \texttt{y=m}\ \}\ \texttt{while true do skip}\ \{\ \texttt{y=n} \wedge \texttt{x=m}\ \}$

The benefits of the axiomatic approach are that the logical systems provide an easy way of proving properties of programs — and that to a large extent it has been possible to automate it. Of course this is only worthwhile if the axiomatic semantics is faithful to the "more general" (denotational or operational) semantics we have in mind.

An axiomatic approach is developed in Chapter 9 for the language **While**; it is extended to other language constructs in Chapter 10, where we also show how the approach can be modified to treat termination properties and properties about execution times.

The Complementary View

It is important to note that these kinds of semantics are *not* rival approaches but are different techniques appropriate for different purposes and — to some extent — for different programming languages. To stress this, the development in this book will address the following issues:

– It will develop each of the approaches for a simple language **While** of `while`-programs.

– It will illustrate the power and weakness of each of the approaches by extending **While** with other programming constructs.

– It will prove the relationship between the approaches for **While**.

– It will give examples of applications of the semantic descriptions in order to illustrate their merits.

1.2 The Example Language While

This book illustrates the various forms of semantics on a very simple imperative programming language called **While**. As a first step, we must specify its syntax.

The syntactic notation we use is based on BNF. First we list the various *syntactic categories* and give a meta-variable that will be used to range over *constructs* of each category. For our language, the meta-variables and categories are as follows:

n will range over numerals, **Num**,

x will range over variables, **Var**,

a will range over arithmetic expressions, **Aexp**,

b will range over boolean expressions, **Bexp**, and

S will range over statements, **Stm**.

The meta-variables can be primed or subscripted. So, for example, n, n', n_1, and n_2 all stand for numerals.

We assume that the structure of numerals and variables is given elsewhere; for example, numerals might be strings of digits, and variables might be strings of letters and digits starting with a letter. The structure of the other constructs is:

$$\begin{array}{lll}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S
\end{array}$$

Thus, a boolean expression b can only have one of six forms. It is called a *basis element* if it is `true` or `false` or has the form $a_1 = a_2$ or $a_1 \leq a_2$, where a_1 and a_2 are arithmetic expressions. It is called a *composite element* if it has the

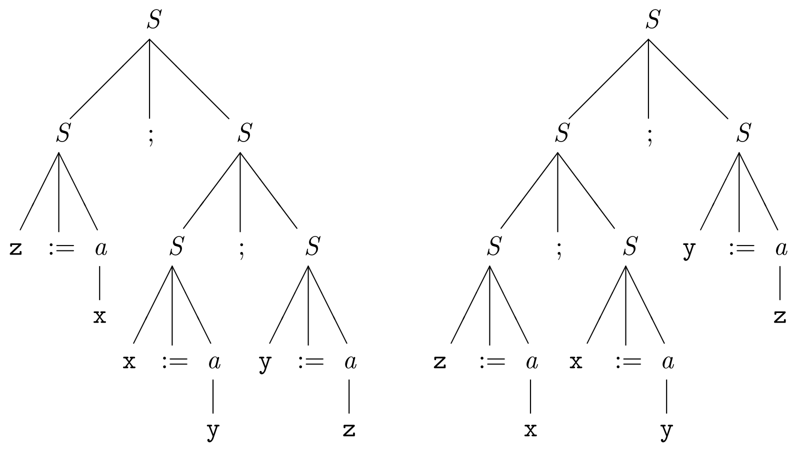


Figure 1.1 Abstract syntax trees for z:=x; x:=y; y:=z

form $\neg b$, where b is a boolean expression, or the form $b_1 \wedge b_2$, where b_1 and b_2 are boolean expressions. Similar remarks apply to arithmetic expressions and statements.

The specification above defines the *abstract syntax* of **While** in that it simply says how to build arithmetic expressions, boolean expressions, and statements in the language. One way to think of the abstract syntax is as specifying the parse trees of the language, and it will then be the purpose of the *concrete syntax* to provide sufficient information that enables unique parse trees to be constructed.

So given the string of characters

z:=x; x:=y; y:=z

the concrete syntax of the language must be able to resolve which of the two abstract syntax trees of Figure 1.1 it is intended to represent. In this book, we shall *not* be concerned with concrete syntax. Whenever we talk about syntactic entities such as arithmetic expressions, boolean expressions, or statements, we will always be talking about the abstract syntax so there is no ambiguity with respect to the form of the entity. In particular, the two trees above are different elements of the syntactic category **Stm**.

It is rather cumbersome to use the graphical representation of abstract syntax, and we shall therefore use a linear notation. So we shall write

z:=x; (x:=y; y:=z)

for the leftmost syntax tree and

$$(\texttt{z:=x; x:=y}); \texttt{y:=z}$$

for the rightmost one. For statements, one often writes the brackets as `begin` $\cdots$ `end`, but we shall feel free to use ($\cdots$) in this book. Similarly, we use brackets ($\cdots$) to resolve ambiguities for elements in the other syntactic categories. To cut down on the number of brackets needed, we shall allow use of the familiar relative binding powers (precedences) of $+$, $\star$, $-$, etc., and so write `1+x⋆2` for `1+(x⋆2)` but not for `(1+x)⋆2`.

Exercise 1.1

The following is a statement in **While**:

$$\texttt{y:=1; while } \neg(\texttt{x=1}) \texttt{ do } (\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1})$$

It computes the factorial of the initial value bound to `x` (provided that it is positive), and the result will be the final value of `y`. Draw a graphical representation of the abstract syntax tree. □

Exercise 1.2

Assume that the initial value of the variable `x` is n and that the initial value of `y` is m. Write a statement in **While** that assigns `z` the value of n to the power of m, that is

$$\underbrace{n \cdot \ldots \cdot n}_{m \text{ times}}$$

Give a linear as well as a graphical representation of the abstract syntax. □

The semantics of **While** is given by defining so-called *semantic functions* for each of the syntactic categories. The idea is that a semantic function takes a syntactic entity as argument and returns its meaning. The operational, denotational, and axiomatic approaches mentioned earlier will be used to specify semantic functions for the statements of **While**. For numerals, arithmetic expressions, and boolean expressions, the semantic functions are specified once and for all below.

1.3 Semantics of Expressions

Before embarking on specifying the semantics of the arithmetic and boolean expressions of **While**, let us have a brief look at the numerals; this will present the main ingredients of the approach in a very simple setting. So assume for the moment that the numerals are in the *binary* system. Their abstract syntax could then be specified by

$$n ::= \texttt{0} \mid \texttt{1} \mid n\ \texttt{0} \mid n\ \texttt{1}$$

In order to determine the number represented by a numeral, we shall define a function

$$\mathcal{N}\colon \mathbf{Num} \to \mathbf{Z}$$

This is called a *semantic function*, as it defines the semantics of the numerals. We want $\mathcal{N}$ to be a *total function* because we want to determine a unique number for each numeral of **Num**. If $n \in \mathbf{Num}$, then we write $\mathcal{N}[\![n]\!]$ for the application of $\mathcal{N}$ to n; that is, for the corresponding number. In general, the application of a semantic function to a syntactic entity will be written within the "syntactic" brackets '$[\![$' and '$]\!]$' rather than the more usual '(' and ')'. These brackets have no special meaning, but throughout this book we shall enclose syntactic arguments to semantic functions using the "syntactic" brackets, whereas we use ordinary brackets (or juxtapositioning) in all other cases.

The semantic function $\mathcal{N}$ is defined by the following *semantic clauses* (or *equations*):

$$\begin{aligned}
\mathcal{N}[\![\texttt{0}]\!] &= \mathbf{0}\\
\mathcal{N}[\![\texttt{1}]\!] &= \mathbf{1}\\
\mathcal{N}[\![n\ \texttt{0}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!]\\
\mathcal{N}[\![n\ \texttt{1}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![n]\!] + \mathbf{1}
\end{aligned}$$

Here **0** and **1** are mathematical numbers; that is, elements of **Z**. Furthermore, $\cdot$ and $+$ are the usual arithmetic operations on numbers. The definition above is an example of a *compositional* definition. This means that for each possible way of constructing a numeral, it tells how the corresponding number is obtained from the meanings of the *sub*constructs.

Example 1.3

We can calculate the number $\mathcal{N}[\![\texttt{101}]\!]$ corresponding to the numeral `101` as follows:

$$\begin{aligned}
\mathcal{N}[\![\texttt{101}]\!] &= \mathbf{2} \cdot \mathcal{N}[\![\texttt{10}]\!] + \mathbf{1}\\
&= \mathbf{2} \cdot (\mathbf{2} \cdot \mathcal{N}[\![\texttt{1}]\!]) + \mathbf{1}\\
&= \mathbf{2} \cdot (\mathbf{2} \cdot \mathbf{1}) + \mathbf{1}\\
&= \mathbf{5}
\end{aligned}$$

Note that the string `101` is decomposed in strict accordance with the syntax for numerals. □

Exercise 1.4

Suppose that the grammar for n had been

$$n ::= \texttt{0} \mid \texttt{1} \mid \texttt{0}\ n \mid \texttt{1}\ n$$

Can you define $\mathcal{N}$ correctly in this case? □

So far we have only *claimed* that the definition of $\mathcal{N}$ gives rise to a well-defined total function. We shall now present a *formal proof* showing that this is indeed the case.

Fact 1.5

The equations above for $\mathcal{N}$ define a total function $\mathcal{N}$: $\mathbf{Num} \to \mathbf{Z}$.

Proof: We have a total function $\mathcal{N}$ if for all arguments $n \in \mathbf{Num}$ it holds that

$$\text{there is exactly one number } \mathbf{n} \in \mathbf{Z} \text{ such that } \mathcal{N}[\![n]\!] = \mathbf{n} \qquad (*)$$

Given a numeral n, it can have one of four forms: it can be a basis element and then is equal to `0` or `1`, or it can be a composite element and then is equal to $n'\texttt{0}$ or $n'\texttt{1}$ for some other numeral n'. So, in order to prove (*), we have to consider all four possibilities.

The proof will be conducted by *induction* on the *structure* of the numeral n. In the *base case*, we prove (*) for the basis elements of **Num**; that is, for the cases where n is `0` or `1`. In the *induction step*, we consider the composite elements of **Num**; that is, the cases where n is $n'\texttt{0}$ or $n'\texttt{1}$. The induction hypothesis will then allow us to assume that (*) holds for the immediate constituent of n; that is, n'. We shall then prove that (*) holds for n. It then follows that (*) holds for all numerals n because any numeral n can be constructed in that way.

The case $n = \texttt{0}$: Only one of the semantic clauses defining $\mathcal{N}$ can be used, and it gives $\mathcal{N}[\![n]\!] = \mathbf{0}$. So clearly there is exactly one number $\mathbf{n}$ in $\mathbf{Z}$ (namely $\mathbf{0}$) such that $\mathcal{N}[\![n]\!] = \mathbf{n}$.

The case $n = \texttt{1}$ is similar and we omit the details.

The case $n = n'\texttt{0}$: Inspection of the clauses defining $\mathcal{N}$ shows that only one of the clauses is applicable and we have $\mathcal{N}[\![n]\!] = \mathbf{2} \cdot \mathcal{N}[\![n']\!]$. We can now apply the induction hypothesis to n' and get that there is exactly one number $\mathbf{n}'$

such that $\mathcal{N}[\![n']\!] = \mathbf{n}'$. But then it is clear that there is exactly one number $\mathbf{n}$ (namely $\mathbf{2} \cdot \mathbf{n}'$) such that $\mathcal{N}[\![n]\!] = \mathbf{n}$.

The case $n = n'\mathtt{1}$ is similar and we omit the details. □

The general technique that we have applied in the definition of the syntax and semantics of numerals can be summarized as follows:

Compositional Definitions

1: The syntactic category is specified by an abstract syntax giving the *basis elements* and the *composite elements*. The composite elements have a unique decomposition into their immediate constituents.

2: The semantics is defined by *compositional* definitions of a function: There is a *semantic clause* for each of the basis elements of the syntactic category and one for each of the methods for constructing composite elements. The clauses for composite elements are defined in terms of the semantics of the immediate constituents of the elements.

The proof technique we have applied is closely connected with the approach to defining semantic functions. It can be summarized as follows:

Structural Induction

1: Prove that the property holds for all the *basis* elements of the syntactic category.

2: Prove that the property holds for all the *composite* elements of the syntactic category: Assume that the property holds for all the immediate constituents of the element (this is called the *induction hypothesis*) and prove that it also holds for the element itself.

In the remainder of this book, we shall assume that numerals are in decimal notation and have their normal meanings (so, for example, $\mathcal{N}[\![\mathtt{137}]\!] = \mathbf{137} \in \mathbf{Z}$). It is important to understand, however, that there is a distinction between numerals (which are syntactic) and numbers (which are semantic), even in decimal notation.

Semantic Functions

The meaning of an expression depends on the values bound to the variables that occur in it. For example, if `x` is bound to **3**, then the arithmetic expression `x+1` evaluates to **4**, but if `x` is bound to **2**, then the expression evaluates to **3**. We shall therefore introduce the concept of a *state*: to each variable the state will associate its current value. We shall represent a state as a function from variables to values; that is, an element of the set

$$\mathbf{State} = \mathbf{Var} \to \mathbf{Z}$$

Each state s specifies a value, written $s\ x$, for each variable x of **Var**. Thus, if s `x` $= \mathbf{3}$, then the value of `x+1` in state s is **4**.

Actually, this is just one of several representations of the state. Some other possibilities are to use a table

x	**5**
y	**7**
z	**0**

or a "list" of the form

$$[\mathtt{x}\mapsto\mathbf{5},\ \mathtt{y}\mapsto\mathbf{7},\ \mathtt{z}\mapsto\mathbf{0}]$$

(as in Section 1.1). In all cases, we must ensure that exactly one value is associated with each variable. By requiring a state to be a function, this is trivially fulfilled, whereas for the alternative representations above extra, restrictions have to be enforced.

Given an arithmetic expression a and a state s, we can determine the value of the expression. Therefore we shall define the meaning of arithmetic expressions as a total function $\mathcal{A}$ that takes two arguments: the syntactic construct *and* the state. The functionality of $\mathcal{A}$ is

$$\mathcal{A}\colon \mathbf{Aexp} \to (\mathbf{State} \to \mathbf{Z})$$

This means that $\mathcal{A}$ takes its parameters *one at a time*. So we may supply $\mathcal{A}$ with its first parameter, say `x+1`, and study the function $\mathcal{A}[\![\mathtt{x+1}]\!]$. It has functionality $\mathbf{State} \to \mathbf{Z}$, and only when we supply it with a state (which happens to be a function, but that does not matter) do we obtain the value of the expression `x+1`.

Assuming the existence of the function $\mathcal{N}$ defining the meaning of numerals, we can define the function $\mathcal{A}$ by defining its value $\mathcal{A}[\![a]\!]s$ on each arithmetic expression a and state s. The definition of $\mathcal{A}$ is given in Table 1.1. The clause for n reflects that the value of n in any state is $\mathcal{N}[\![n]\!]$. The value of a variable x

$$
\begin{array}{rcl}
\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![a_1\ +\ a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1\ \star\ a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1\ -\ a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{array}
$$

Table 1.1 The semantics of arithmetic expressions

in state s is the value bound to x in s; that is, $s\ x$. The value of the composite expression $a_1{+}a_2$ in s is the sum of the values of a_1 and a_2 in s. Similarly, the value of $a_1 \star a_2$ in s is the product of the values of a_1 and a_2 in s, and the value of $a_1 - a_2$ in s is the difference between the values of a_1 and a_2 in s. Note that $+$ and $-$ occurring on the right of these equations are the usual arithmetic operations, while on the left they are just pieces of syntax; this is analogous to the distinction between numerals and numbers, but we shall not bother to use different symbols.

Example 1.6

Suppose that $s\ \mathtt{x} = \mathbf{3}$. Then we may calculate:

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathtt{x{+}1}]\!]s & = & \mathcal{A}[\![\mathtt{x}]\!]s + \mathcal{A}[\![\mathtt{1}]\!]s \\
& = & (s\ \mathtt{x}) + \mathcal{N}[\![\mathtt{1}]\!] \\
& = & \mathbf{3} + \mathbf{1} \\
& = & \mathbf{4}
\end{array}
$$

Note that here $\mathtt{1}$ is a numeral (enclosed in the brackets '$[\![$' and '$]\!]$'), whereas $\mathbf{1}$ is a number. □

Example 1.7

Suppose we add the arithmetic expression $-\ a$ to our language. An acceptable semantic clause for this construct would be

$$\mathcal{A}[\![-\ a]\!]s = \mathbf{0} - \mathcal{A}[\![a]\!]s$$

whereas the alternative clause $\mathcal{A}[\![-\ a]\!]s = \mathcal{A}[\![\mathtt{0}\ -\ a]\!]s$ would contradict the compositionality requirement. □

$$
\begin{array}{rcl}
\mathcal{B}[\![\texttt{true}]\!]s & = & \mathbf{tt} \\[1ex]
\mathcal{B}[\![\texttt{false}]\!]s & = & \mathbf{ff} \\[1ex]
\mathcal{B}[\![a_1 = a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\[2ex]
\mathcal{B}[\![a_1 \le a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \le \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\[2ex]
\mathcal{B}[\![\neg\, b]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\[2ex]
\mathcal{B}[\![b_1 \wedge b_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{array}
$$

Table 1.2 The semantics of boolean expressions

Exercise 1.8

Prove that the equations of Table 1.1 define a total function $\mathcal{A}$ in **Aexp** $\rightarrow$ (**State** $\rightarrow$ **Z**): First argue that it is sufficient to prove that for each $a \in$ **Aexp** and each $s \in$ **State** there is exactly one value $\mathbf{v} \in \mathbf{Z}$ such that $\mathcal{A}[\![a]\!]s = \mathbf{v}$. Next use structural induction on the arithmetic expressions to prove that this is indeed the case. □

The values of boolean expressions are truth values, so in a similar way we shall define their meanings by a (total) function from **State** to **T**:

$$\mathcal{B}\colon \mathbf{Bexp} \rightarrow (\mathbf{State} \rightarrow \mathbf{T})$$

Here **T** consists of the truth values **tt** (for true) and **ff** (for false).

Using $\mathcal{A}$, we can define $\mathcal{B}$ by the semantic clauses of Table 1.2. Again we have the distinction between syntax (e.g., $\le$ on the left-hand side) and semantics (e.g., $\le$ on the right-hand side).

Exercise 1.9

Assume that $s\ \texttt{x} = \mathbf{3}$, and determine $\mathcal{B}[\![\neg(\texttt{x} = \texttt{1})]\!]s$. □

Exercise 1.10

Prove that Table 1.2 defines a total function $\mathcal{B}$ in **Bexp** $\rightarrow$ (**State** $\rightarrow$ **T**). □

Exercise 1.11

The syntactic category **Bexp′** is defined as the following extension of **Bexp**:

$$\begin{array}{rcl} b & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \\ & \mid & a_1 < a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\ & \mid & b_1 \Rightarrow b_2 \mid b_1 \Leftrightarrow b_2 \end{array}$$

Give a *compositional* extension of the semantic function $\mathcal{B}$ of Table 1.2.

Two boolean expressions b_1 and b_2 are *equivalent* if for all states s:

$$\mathcal{B}[\![b_1]\!]s = \mathcal{B}[\![b_2]\!]s$$

Show that for each b' of **Bexp′** there exists a boolean expression b of **Bexp** such that b' and b are equivalent. □

1.4 Properties of the Semantics

Later in the book, we shall be interested in two kinds of properties for expressions. One is that their values do not depend on values of variables that do not occur in them. The other is that if we replace a variable with an expression, then we could as well have made a similar change in the state. We shall formalize these properties below and prove that they do hold.

Free Variables

The *free variables* of an arithmetic expression a are defined to be the set of variables occurring in it. Formally, we may give a compositional definition of the subset FV(a) of **Var**:

$$\begin{array}{rcl} \mathrm{FV}(n) & = & \emptyset \\ \mathrm{FV}(x) & = & \{\, x \,\} \\ \mathrm{FV}(a_1 + a_2) & = & \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\ \mathrm{FV}(a_1 \star a_2) & = & \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\ \mathrm{FV}(a_1 - a_2) & = & \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \end{array}$$

As an example, FV(x+1) = { x } and FV(x+y⋆x) = { x, y }. It should be obvious that only the variables in FV(a) may influence the value of a. This is formally expressed by the following lemma.

Lemma 1.12

Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(a)$. Then $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Proof: We shall give a fairly detailed proof of the lemma using structural induction on the arithmetic expressions. We shall first consider the basis elements of **Aexp**.

The case n: From Table 1.1 we have $\mathcal{A}[\![n]\!]s = \mathcal{N}[\![n]\!]$ as well as $\mathcal{A}[\![n]\!]s' = \mathcal{N}[\![n]\!]$. So $\mathcal{A}[\![n]\!]s = \mathcal{A}[\![n]\!]s'$ and clearly the lemma holds in this case.

The case x: From Table 1.1, we have $\mathcal{A}[\![x]\!]s = s\ x$ as well as $\mathcal{A}[\![x]\!]s' = s'\ x$. From the assumptions of the lemma, we get $s\ x = s'\ x$ because $x \in \mathrm{FV}(x)$, so clearly the lemma holds in this case.

Next we turn to the composite elements of **Aexp**.

The case $a_1 + a_2$: From Table 1.1, we have $\mathcal{A}[\![a_1 + a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$ and similarly $\mathcal{A}[\![a_1 + a_2]\!]s' = \mathcal{A}[\![a_1]\!]s' + \mathcal{A}[\![a_2]\!]s'$. Since a_i (for i = 1, 2) is an immediate subexpression of $a_1 + a_2$ and $\mathrm{FV}(a_\mathrm{i}) \subseteq \mathrm{FV}(a_1 + a_2)$, we can apply the induction hypothesis (that is, the lemma) to a_i and get $\mathcal{A}[\![a_\mathrm{i}]\!]s = \mathcal{A}[\![a_\mathrm{i}]\!]s'$. It is now easy to see that the lemma holds for $a_1 + a_2$ as well.

The cases $a_1 - a_2$ and $a_1 \star a_2$ follow the same pattern and are omitted. This completes the proof. $\square$

In a similar way, we may define the set $\mathrm{FV}(b)$ of free variables in a boolean expression b as follows:

$$\begin{array}{rcl}
\mathrm{FV}(\texttt{true}) & = & \emptyset \\
\mathrm{FV}(\texttt{false}) & = & \emptyset \\
\mathrm{FV}(a_1 = a_2) & = & \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \leq a_2) & = & \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(\neg b) & = & \mathrm{FV}(b) \\
\mathrm{FV}(b_1 \wedge b_2) & = & \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)
\end{array}$$

Exercise 1.13 (Essential)

Let s and s' be two states satisfying that $s\ x = s'\ x$ for all x in $\mathrm{FV}(b)$. Prove that $\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$. $\square$

Substitutions

We shall later be interested in replacing each occurrence of a variable y in an arithmetic expression a with another arithmetic expression a_0. This is called *substitution*, and we write $a[y{\mapsto}a_0]$ for the arithmetic expression so obtained. The formal definition is as follows:

$$\begin{aligned}
n[y{\mapsto}a_0] &= n \\
x[y{\mapsto}a_0] &= \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y{\mapsto}a_0] &= (a_1[y{\mapsto}a_0]) + (a_2[y{\mapsto}a_0]) \\
(a_1 \star a_2)[y{\mapsto}a_0] &= (a_1[y{\mapsto}a_0]) \star (a_2[y{\mapsto}a_0]) \\
(a_1 - a_2)[y{\mapsto}a_0] &= (a_1[y{\mapsto}a_0]) - (a_2[y{\mapsto}a_0])
\end{aligned}$$

As an example, $(\mathtt{x}+\mathtt{1})[\mathtt{x}{\mapsto}\mathtt{3}] = \mathtt{3}+\mathtt{1}$ and $(\mathtt{x}+\mathtt{y}\star\mathtt{x})[\mathtt{x}{\mapsto}\mathtt{y}-\mathtt{5}] = (\mathtt{y}-\mathtt{5})+\mathtt{y}\star(\mathtt{y}-\mathtt{5})$.

We also have a notion of substitution (or updating) for states. We define $s[y{\mapsto}v]$ to be the state that is like s except that the value bound to y is v:

$$(s[y{\mapsto}v])\ x = \begin{cases} v & \text{if } x = y \\ s\ x & \text{if } x \neq y \end{cases}$$

The relationship between the two concepts is shown in the following exercise.

Exercise 1.14 (Essential)

Prove that $\mathcal{A}[\![a[y{\mapsto}a_0]]\!]s = \mathcal{A}[\![a]\!](s[y{\mapsto}\mathcal{A}[\![a_0]\!]s])$ for all states s. □

Exercise 1.15 (Essential)

Define substitution for boolean expressions: $b[y{\mapsto}a_0]$ is to be the boolean expression that is like b except that all occurrences of the variable y are replaced by the arithmetic expression a_0. Prove that your definition satisfies

$$\mathcal{B}[\![b[y{\mapsto}a_0]]\!]s = \mathcal{B}[\![b]\!](s[y{\mapsto}\mathcal{A}[\![a_0]\!]s])$$

for all states s. □

2 Operational Semantics

The role of a statement in **While** is to change the state. For example, if `x` is bound to **3** in s and we execute the statement `x := x + 1`, then we get a new state where `x` is bound to **4**. So while the semantics of arithmetic and boolean expressions only *inspect* the state in order to determine the value of the expression, the semantics of statements will *modify* the state as well.

In an operational semantics, we are concerned with *how* to execute programs and not merely what the results of execution are. More precisely, we are interested in how the states are modified during the execution of the statement. We shall consider two different approaches to operational semantics:

– *Natural semantics*: Its purpose is to describe how the *overall* results of executions are obtained; sometimes it is called a *big-step* operational semantics.

– *Structural operational semantics*: Its purpose is to describe how the *individual steps* of the computations take place; sometimes it is called a *small-step* operational semantics.

We shall see that for the language **While** we can easily specify both kinds of semantics and that they will be "equivalent" in a sense to be made clear later. However, in the next chapter we shall also give examples of programming constructs where one of the approaches is superior to the other.

For both kinds of operational semantics, the meaning of statements will be specified by a *transition system*. It will have two types of configurations:

[ass$_{\rm ns}$]	$\langle x := a,\ s\rangle \to s[x\mapsto\mathcal{A}[\![a]\!]s]$
[skip$_{\rm ns}$]	$\langle \texttt{skip},\ s\rangle \to s$
[comp$_{\rm ns}$]	$\dfrac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$
[if$^{\rm tt}_{\rm ns}$]	$\dfrac{\langle S_1,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
[if$^{\rm ff}_{\rm ns}$]	$\dfrac{\langle S_2,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
[while$^{\rm tt}_{\rm ns}$]	$\dfrac{\langle S,\ s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
[while$^{\rm ff}_{\rm ns}$]	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Table 2.1 Natural semantics for **While**

$\langle S,\ s\rangle$	representing that the statement S is to be executed from the state s and
s	representing a terminal (that is final) state.

The *terminal configurations* will be those of the latter form. The *transition relation* will then describe how the execution takes place. The difference between the two approaches to operational semantics amounts to different ways of specifying the transition relation.

2.1 Natural Semantics

In a natural semantics we are concerned with the relationship between the *initial* and the *final* state of an execution. Therefore the transition relation will specify the relationship between the initial state and the final state for each statement. We shall write a transition as

$$\langle S,\ s\rangle \to s'$$

Intuitively this means that the execution of S from s will terminate and the resulting state will be s'.

The definition of $\to$ is given by the rules of Table 2.1. A *rule* has the general form

$$\frac{\langle S_1,\, s_1\rangle \to s_1',\, \cdots,\, \langle S_\mathrm{n},\, s_\mathrm{n}\rangle \to s_\mathrm{n}'}{\langle S,\, s\rangle \to s'} \quad \text{if } \cdots$$

where $S_1, \cdots, S_\mathrm{n}$ are *immediate constituents* of S or are statements *constructed from* the immediate constituents of S. A rule has a number of *premises* (written above the solid line) and one *conclusion* (written below the solid line). A rule may also have a number of *conditions* (written to the right of the solid line) that have to be fulfilled whenever the rule is applied. Rules with an empty set of premises are called *axioms* and the solid line is then omitted.

Intuitively, the axiom [ass$_\mathrm{ns}$] says that in a state s, $x := a$ is executed to yield a final state $s[x{\mapsto}\mathcal{A}[\![a]\!]s]$, which is like s except that x has the value $\mathcal{A}[\![a]\!]s$. This is really an *axiom schema* because x, a, and s are meta-variables standing for arbitrary variables, arithmetic expressions, and states but we shall simply use the term *axiom* for this. We obtain an *instance* of the axiom by selecting particular variables, arithmetic expressions, and states. As an example, if s_0 is the state that assigns the value **0** to all variables, then

$$\langle \texttt{x := x+1},\, s_0\rangle \to s_0[\texttt{x}{\mapsto}\mathbf{1}]$$

is an instance of [ass$_\mathrm{ns}$] because x is instantiated to `x`, a to `x+1`, and s to s_0, and the value $\mathcal{A}[\![\texttt{x+1}]\!]s_0$ is determined to be **1**.

Similarly, [skip$_\mathrm{ns}$] is an axiom and, intuitively, it says that `skip` does not change the state. Letting s_0 be as above, we obtain

$$\langle \texttt{skip},\, s_0\rangle \to s_0$$

as an instance of the axiom [skip$_\mathrm{ns}$].

Intuitively, the rule [comp$_\mathrm{ns}$] says that to execute S_1;S_2 from state s we must first execute S_1 from s. Assuming that this yields a final state s', we shall then execute S_2 from s'. The premises of the rule are concerned with the two statements S_1 and S_2, whereas the conclusion expresses a property of the composite statement itself. The following is an *instance* of the rule:

$$\frac{\langle \texttt{skip},\, s_0\rangle \to s_0,\ \langle \texttt{x := x+1},\, s_0\rangle \to s_0[\texttt{x}{\mapsto}\mathbf{1}]}{\langle \texttt{skip; x := x+1},\, s_0\rangle \to s_0[\texttt{x}{\mapsto}\mathbf{1}]}$$

Here S_1 is instantiated to `skip`, S_2 to `x := x + 1`, s and s' are both instantiated to s_0, and s'' is instantiated to $s_0[\texttt{x}{\mapsto}\mathbf{1}]$. Similarly

$$\frac{\langle \texttt{skip},\, s_0\rangle \to s_0[\texttt{x}{\mapsto}\mathbf{5}],\ \langle \texttt{x := x+1},\, s_0[\texttt{x}{\mapsto}\mathbf{5}]\rangle \to s_0}{\langle \texttt{skip; x := x+1},\, s_0\rangle \to s_0}$$

is an instance of [comp$_\mathrm{ns}$], although it is less interesting because its premises can never be derived from the axioms and rules of Table 2.1.

For the `if`-construct, we have two rules. The first one, [if$^\mathrm{tt}_\mathrm{ns}$], says that to execute `if` b `then` S_1 `else` S_2 we simply execute S_1 provided that b evaluates to $\mathbf{tt}$ in the state. The other rule, $[\text{if}_{\text{ns}}^{\text{ff}}]$, says that if b evaluates to **ff**, then to execute `if` b `then` S_1 `else` S_2 we just execute S_2. Taking s_0 `x` $= \mathbf{0}$,

$$\frac{\langle \texttt{skip},\ s_0\rangle \rightarrow s_0}{\langle \texttt{if x = 0 then skip else x := x+1},\ s_0\rangle \rightarrow s_0}$$

is an instance of the rule $[\text{if}_{\text{ns}}^{\text{tt}}]$ because $\mathcal{B}[\![\texttt{x = 0}]\!]s_0 = \mathbf{tt}$. However, had it been the case that s_0 `x` $\neq \mathbf{0}$, then it would not be an instance of the rule $[\text{if}_{\text{ns}}^{\text{tt}}]$ because then $\mathcal{B}[\![\texttt{x = 0}]\!]s_0$ would amount to **ff**. Furthermore, it would not be an instance of the rule $[\text{if}_{\text{ns}}^{\text{ff}}]$ because the premise would contain the wrong statement.

Finally, we have one rule and one axiom expressing how to execute the `while`-construct. Intuitively, the meaning of the construct `while` b `do` S in the state s can be explained as follows:

– If the test b evaluates to true in the state s, then we first execute the body of the loop and then continue with the loop itself from the state so obtained.

– If the test b evaluates to false in the state s, then the execution of the loop terminates.

The rule $[\text{while}_{\text{ns}}^{\text{tt}}]$ formalizes the first case where b evaluates to $\mathbf{tt}$ and it says that then we have to execute S followed by `while` b `do` S again. The axiom $[\text{while}_{\text{ns}}^{\text{ff}}]$ formalizes the second possibility and states that if b evaluates to **ff**, then we terminate the execution of the `while`-construct, leaving the state unchanged. Note that the rule $[\text{while}_{\text{ns}}^{\text{tt}}]$ specifies the meaning of the `while`-construct in terms of the meaning of the very same construct, so we do *not* have a compositional definition of the semantics of statements.

When we use the axioms and rules to derive a transition $\langle S,\ s\rangle \rightarrow s'$, we obtain a *derivation tree*. The *root* of the derivation tree is $\langle S,\ s\rangle \rightarrow s'$ and the *leaves* are instances of axioms. The *internal nodes* are conclusions of instantiated rules, and they have the corresponding premises as their immediate sons. We request that all the instantiated conditions of axioms and rules be satisfied. When displaying a derivation tree, it is common to have the root at the bottom rather than at the top; hence the son is *above* its father. A derivation tree is called *simple* if it is an instance of an axiom; otherwise it is called *composite*.

Example 2.1

Let us first consider the statement of Chapter 1:

$$\texttt{(z:=x; x:=y); y:=z}$$

Let s_0 be the state that maps all variables except `x` and `y` to $\mathbf{0}$ and has s_0 `x` $= \mathbf{5}$ and s_0 `y` $= \mathbf{7}$. Then an example of a derivation tree is

$$\dfrac{\dfrac{\langle \mathtt{z:=x},\ s_0\rangle \to s_1 \qquad \langle \mathtt{x:=y},\ s_1\rangle \to s_2}{\langle \mathtt{z:=x;\ x:=y},\ s_0\rangle \to s_2} \qquad \langle \mathtt{y:=z},\ s_2\rangle \to s_3}{\langle (\mathtt{z:=x;\ x:=y});\ \mathtt{y:=z},\ s_0\rangle \to s_3}$$

where we have used the abbreviations:

$$\begin{array}{rcl} s_1 & = & s_0[\mathtt{z}\mapsto\mathbf{5}] \\ s_2 & = & s_1[\mathtt{x}\mapsto\mathbf{7}] \\ s_3 & = & s_2[\mathtt{y}\mapsto\mathbf{5}] \end{array}$$

The derivation tree has three leaves, denoted $\langle \mathtt{z:=x},\ s_0\rangle \to s_1$, $\langle \mathtt{x:=y},\ s_1\rangle \to s_2$, and $\langle \mathtt{y:=z},\ s_2\rangle \to s_3$, corresponding to three applications of the axiom [ass_{ns}]. The rule [comp_{ns}] has been applied twice. One instance is

$$\dfrac{\langle \mathtt{z:=x},\ s_0\rangle \to s_1,\ \langle \mathtt{x:=y},\ s_1\rangle \to s_2}{\langle \mathtt{z:=x;\ x:=y},\ s_0\rangle \to s_2}$$

which has been used to combine the leaves $\langle \mathtt{z:=x},\ s_0\rangle \to s_1$ and $\langle \mathtt{x:=y},\ s_1\rangle \to s_2$ with the internal node labelled $\langle \mathtt{z:=x;\ x:=y},\ s_0\rangle \to s_2$. The other instance is

$$\dfrac{\langle \mathtt{z:=x;\ x:=y},\ s_0\rangle \to s_2,\ \langle \mathtt{y:=z},\ s_2\rangle \to s_3}{\langle (\mathtt{z:=x;\ x:=y});\ \mathtt{y:=z},\ s_0\rangle \to s_3}$$

which has been used to combine the internal node $\langle \mathtt{z:=x;\ x:=y},\ s_0\rangle \to s_2$ and the leaf $\langle \mathtt{y:=z},\ s_2\rangle \to s_3$ with the root $\langle (\mathtt{z:=x;\ x:=y});\ \mathtt{y:=z},\ s_0\rangle \to s_3$. □

Consider now the problem of constructing a derivation tree for a given statement S and state s. The best way to approach this is to try to construct the tree from the root upwards. So we will start by finding an axiom or rule with a conclusion where the left-hand side matches the configuration $\langle S,\ s\rangle$. There are two cases:

– If it is an *axiom* and if the conditions of the axiom are satisfied, then we can determine the final state and the construction of the derivation tree is completed.

– If it is a *rule*, then the next step is to try to construct derivation trees for the premises of the rule. When this has been done, it must be checked that the conditions of the rule are fulfilled, and only then can we determine the final state corresponding to $\langle S,\ s\rangle$.

Often there will be more than one axiom or rule that matches a given configuration, and then the various possibilities have to be inspected in order to find a derivation tree. We shall see later that for **While** there will be at most one derivation tree for each transition $\langle S,\ s\rangle \to s'$ but that this need not hold in extensions of **While**.

Example 2.2

Consider the factorial statement

$$\texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y} \star \texttt{x; x:=x}-\texttt{1)}$$

and let s be a state with $s\ \texttt{x} = \mathbf{3}$. In this example, we shall show that

$$\langle\texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y} \star \texttt{x; x:=x}-\texttt{1)},\ s\rangle \to s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{1}] \qquad (*)$$

To do so, we shall show that (*) can be obtained from the transition system of Table 2.1. This is done by constructing a derivation tree with the transition (*) as its root.

Rather than presenting the complete derivation tree T in one go, we shall build it in an upwards manner. Initially, we only know that the root of T is of the form (where we use an auxiliary state s_{61} to be defined later)

$$\langle\texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y} \star \texttt{x; x:=x}-\texttt{1)},\ s\rangle \to s_{61}$$

However, the statement

$$\texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y} \star \texttt{x; x:=x}-\texttt{1)}$$

is of the form S_1; S_2, so the only rule that could have been used to produce the root of T is [comp$_{\rm ns}$]. Therefore T must have the form

$$\frac{\langle\texttt{y:=1},\ s\rangle\to s_{13} \qquad\qquad T_1}{\langle\texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s\rangle\to s_{61}}$$

for some state s_{13} and some derivation tree T_1 that has root

$$\langle\texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s_{13}\rangle\to s_{61} \qquad (**)$$

Since $\langle\texttt{y:=1},\ s\rangle \to s_{13}$ has to be an instance of the axiom [ass$_{\rm ns}$], we get that $s_{13} = s[\texttt{y}\mapsto\mathbf{1}]$.

The missing part T_1 of T is a derivation tree with root (**). Since the statement of (**) has the form `while` b `do` S, the derivation tree T_1 must have been constructed by applying either the rule [while$^{\rm tt}_{\rm ns}$] or the axiom [while$^{\rm ff}_{\rm ns}$]. Since $\mathcal{B}[\![\neg(\texttt{x=1})]\!]s_{13} = \mathbf{tt}$, we see that only the rule [while$^{\rm tt}_{\rm ns}$] could have been applied so T_1 will have the form

$$\frac{T_2 \qquad\qquad T_3}{\langle\texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s_{13}\rangle\to s_{61}}$$

where T_2 is a derivation tree with root

$$\langle \texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1},\ s_{13}\rangle \to s_{32}$$

and T_3 is a derivation tree with root

$$\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}),\ s_{32}\rangle \to s_{61} \qquad (***)$$

for some state s_{32}.

Using that the form of the statement y:=y⋆x; x:=x−1 is S_1;S_2, it is now easy to see that the derivation tree T_2 is

$$\frac{\langle \texttt{y:=y}\star\texttt{x},\ s_{13}\rangle \to s_{33} \qquad \langle \texttt{x:=x}-\texttt{1},\ s_{33}\rangle \to s_{32}}{\langle \texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1},\ s_{13}\rangle \to s_{32}}$$

where $s_{33} = s[\texttt{y}\mapsto\mathbf{3}]$ and $s_{32} = s[\texttt{y}\mapsto\mathbf{3}][\texttt{x}\mapsto\mathbf{2}]$. The leaves of T_2 are instances of [ass$_{\text{ns}}$] and are combined using [comp$_{\text{ns}}$]. So now T_2 is fully constructed.

In a similar way, we can construct the derivation tree T_3 with root (***) and we get

$$\frac{\dfrac{\langle \texttt{y:=y}\star\texttt{x},\ s_{32}\rangle \to s_{62} \qquad \langle \texttt{x:=x}-\texttt{1},\ s_{62}\rangle \to s_{61}}{\langle \texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1},\ s_{32}\rangle \to s_{61}} \qquad T_4}{\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}),\ s_{32}\rangle \to s_{61}}$$

where $s_{62} = s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{2}]$, $s_{61} = s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{1}]$, and T_4 is a derivation tree with root

$$\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}),\ s_{61}\rangle \to s_{61}$$

Finally, we see that the derivation tree T_4 is an instance of the axiom [while$^{\text{ff}}_{\text{ns}}$] because $\mathcal{B}[\![\neg(\texttt{x=1})]\!]s_{61} = \mathbf{ff}$. This completes the construction of the derivation tree T for (*). □

Exercise 2.3

Consider the statement

$$\texttt{z:=0; while y}\leq\texttt{x do (z:=z+1; x:=x}-\texttt{y)}$$

Construct a derivation tree for this statement when executed in a state where x has the value **17** and y has the value **5**. □

We shall introduce the following terminology. The execution of a statement S on a state s

– *terminates* if and only if there is a state s' such that $\langle S,\ s\rangle \to s'$ and

– *loops* if and only if there is *no* state s' such that $\langle S,\ s\rangle \to s'$.

(For the latter definition, note that no run-time errors are possible.) We shall say that a statement S *always terminates* if its execution on a state s terminates for all choices of s, and *always loops* if its execution on a state s loops for all choices of s.

Exercise 2.4

Consider the following statements

– `while` $\neg$(x=1) `do` (y:=y$\star$x; x:=x$-$1)

– `while` 1$\leq$x `do` (y:=y$\star$x; x:=x$-$1)

– `while true do skip`

For each statement determine whether or not it always terminates and whether or not it always loops. Try to argue for your answers using the axioms and rules of Table 2.1. □

Properties of the Semantics

The transition system gives us a way of arguing about statements and their properties. As an example, we may be interested in whether two statements S_1 and S_2 are *semantically equivalent*; this means that for all states s and s'

$$\langle S_1,\, s\rangle \to s' \quad \text{if and only if} \quad \langle S_2,\, s\rangle \to s'$$

Lemma 2.5

The statement

$$\texttt{while } b \texttt{ do } S$$

is semantically equivalent to

$$\texttt{if } b \texttt{ then } (S;\, \texttt{while } b \texttt{ do } S) \texttt{ else skip}$$

Proof: The proof is in two parts. We shall first prove that if

$$\langle \texttt{while } b \texttt{ do } S,\, s\rangle \to s'' \tag{*}$$

then

$$\langle \texttt{if } b \texttt{ then } (S;\, \texttt{while } b \texttt{ do } S) \texttt{ else skip},\, s\rangle \to s'' \tag{**}$$

Thus, if the execution of the loop terminates, then so does its one-level unfolding. Later we shall show that if the unfolded loop terminates, then so will the loop itself; the conjunction of these results then proves the lemma.

Because (*) holds, we know that we have a derivation tree T for it. It can have one of two forms depending on whether it has been constructed using the rule [while$^{\rm tt}_{\rm ns}$] or the axiom [while$^{\rm ff}_{\rm ns}$]. In the first case, the derivation tree T has the form

$$\frac{T_1 \qquad\qquad T_2}{\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s''}$$

where T_1 is a derivation tree with root $\langle S,\ s\rangle{\to}s'$ and T_2 is a derivation tree with root $\langle \texttt{while } b \texttt{ do } S,\ s'\rangle{\to}s''$. Furthermore, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Using the derivation trees T_1 and T_2 as the premises for the rules [comp$_{\rm ns}$], we can construct the derivation tree

$$\frac{T_1 \qquad\qquad T_2}{\langle S;\ \texttt{while } b \texttt{ do } S,\ s\rangle \to s''}$$

Using that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, we can use the rule [if$^{\rm tt}_{\rm ns}$] to construct the derivation tree

$$\dfrac{\dfrac{T_1 \qquad\qquad\qquad\qquad\qquad\qquad T_2}{\langle S;\ \texttt{while } b \texttt{ do } S,\ s\rangle \to s''}}{\langle \texttt{if } b \texttt{ then } (S;\ \texttt{while } b \texttt{ do } S) \texttt{ else skip},\ s\rangle \to s''}$$

thereby showing that (**) holds.

Alternatively, the derivation tree T is an instance of [while$^{\rm ff}_{\rm ns}$]. Then $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and we must have that $s''{=}s$. So T simply is

$$\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s$$

Using the axiom [skip$_{\rm ns}$], we get a derivation tree

$$\langle \texttt{skip},\ s\rangle{\to}s''$$

and we can now apply the rule [if$^{\rm ff}_{\rm ns}$] to construct a derivation tree for (**):

$$\frac{\langle \texttt{skip},\ s\rangle \to s''}{\langle \texttt{if } b \texttt{ then } (S;\ \texttt{while } b \texttt{ do } S) \texttt{ else skip},\ s\rangle \to s''}$$

This completes the first part of the proof.

For the second part of the proof, we assume that (**) holds and shall prove that (*) holds. So we have a derivation tree T for (**) and must construct one for (*). Only two rules could give rise to the derivation tree T for (**), namely [if$^{\rm tt}_{\rm ns}$] or [if$^{\rm ff}_{\rm ns}$]. In the first case, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and we have a derivation tree T_1 with root

$$\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$$

The statement has the general form $S_1;\ S_2$, and the only rule that could give this is [comp$_{\text{ns}}$]. Therefore there are derivation trees T_2 and T_3 for

$$\langle S,\ s\rangle \to s'$$

and

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$$

for some state s'. It is now straightforward to use the rule [while$^{\text{tt}}_{\text{ns}}$] to combine T_2 and T_3 into a derivation tree for (*).

In the second case, $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and T is constructed using the rule [if$^{\text{ff}}_{\text{ns}}$]. This means that we have a derivation tree for

$$\langle \texttt{skip},\ s\rangle \to s''$$

and according to axiom [skip$_{\text{ns}}$] it must be the case that s=s''. But then we can use the axiom [while$^{\text{ff}}_{\text{ns}}$] to construct a derivation tree for (*). This completes the proof. $\square$

Exercise 2.6

Prove that the two statements S_1;(S_2;S_3) and (S_1;S_2);S_3 are semantically equivalent. Construct a statement showing that S_1;S_2 is not, in general, semantically equivalent to S_2;S_1. $\square$

Exercise 2.7

Extend the language **While** with the statement

$$\texttt{repeat}\ S\ \texttt{until}\ b$$

and define the relation $\to$ for it. (The semantics of the `repeat`-construct is not allowed to rely on the existence of a `while`-construct in the language.) Prove that `repeat` S `until` b and S; `if` b `then skip else` (`repeat` S `until` b) are semantically equivalent. $\square$

Exercise 2.8

Another iterative construct is

$$\texttt{for}\ x := a_1\ \texttt{to}\ a_2\ \texttt{do}\ S$$

Extend the language **While** with this statement and define the relation $\to$ for it. Evaluate the statement

$$\texttt{y:=1; for z:=1 to x do (y:=y} \star \texttt{ x; x:=x}-\texttt{1)}$$

from a state where `x` has the value 5. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$, so that there is a numeral for each number that may arise during the computation. (The semantics of the `for`-construct is not allowed to rely on the existence of a `while`-construct in the language.) □

In the proof above Table 2.1 was used to inspect the structure of the derivation tree for a certain transition known to hold. In the proof of the next result, we shall combine this with an *induction on the shape of the derivation tree*. The idea can be summarized as follows:

Induction on the Shape of Derivation Trees

1: Prove that the property holds for all the simple derivation trees by showing that it holds for the *axioms* of the transition system.

2: Prove that the property holds for all composite derivation trees: For each *rule* assume that the property holds for its premises (this is called the *induction hypothesis*) and prove that it also holds for the conclusion of the rule provided that the conditions of the rule are satisfied.

We shall say that the semantics of Table 2.1 is *deterministic* if for all choices of S, s, s', and s'' we have that

$$\langle S,\ s\rangle \to s' \text{ and } \langle S,\ s\rangle \to s'' \quad \text{imply} \quad s' = s''$$

This means that for every statement S and initial state s we can uniquely determine a final state s' if (and only if) the execution of S terminates.

Theorem 2.9

The natural semantics of Table 2.1 is deterministic.

Proof: We assume that $\langle S,\ s\rangle{\to}s'$ and shall prove that

$$\text{if } \langle S,\ s\rangle{\to}s'' \text{ then } s' = s''.$$

We shall proceed by induction on the shape of the derivation tree for $\langle S,\ s\rangle{\to}s'$.

The case $[\text{ass}_{\text{ns}}]$: Then S is $x{:=}a$ and s' is $s[x{\mapsto}\mathcal{A}[\![a]\!]s]$. The only axiom or rule that could be used to give $\langle x{:=}a,\ s\rangle{\to}s''$ is $[\text{ass}_{\text{ns}}]$, so it follows that s'' must be $s[x{\mapsto}\mathcal{A}[\![a]\!]s]$ and thereby $s' = s''$.

The case [skip$_{\rm ns}$]: Analogous.

The case [comp$_{\rm ns}$]: Assume that

$$\langle S_1;S_2,\ s\rangle\to s'$$

holds because

$$\langle S_1,\ s\rangle\to s_0 \text{ and } \langle S_2,\ s_0\rangle\to s'$$

for some s_0. The only rule that could be applied to give $\langle S_1;S_2,\ s\rangle\to s''$ is [comp$_{\rm ns}$], so there is a state s_1 such that

$$\langle S_1,\ s\rangle\to s_1 \text{ and } \langle S_2,\ s_1\rangle\to s''$$

The induction hypothesis can be applied to the premise $\langle S_1,\ s\rangle\to s_0$ and from $\langle S_1,\ s\rangle\to s_1$ we get $s_0 = s_1$. Similarly, the induction hypothesis can be applied to the premise $\langle S_2,\ s_0\rangle\to s'$ and from $\langle S_2,\ s_0\rangle\to s''$ we get $s' = s''$ as required.

The case [if$^{\rm tt}_{\rm ns}$]: Assume that

$$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\ s\rangle \to s'$$

holds because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1,\ s\rangle\to s'$$

From $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ we get that the only rule that could be applied to give the alternative $\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\ s\rangle \to s''$ is [if$^{\rm tt}_{\rm ns}$]. So it must be the case that

$$\langle S_1,\ s\rangle \to s''$$

But then the induction hypothesis can be applied to the premise $\langle S_1,\ s\rangle \to s'$ and from $\langle S_1,\ s\rangle \to s''$ we get $s' = s''$.

The case [if$^{\rm ff}_{\rm ns}$]: Analogous.

The case [while$^{\rm tt}_{\rm ns}$]: Assume that

$$\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s'$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt},\ \langle S,\ s\rangle\to s_0 \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s_0\rangle\to s'$$

The only rule that could be applied to give $\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s''$ is [while$^{\rm tt}_{\rm ns}$] because $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, and this means that

$$\langle S,\ s\rangle\to s_1 \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s_1\rangle \to s''$$

must hold for some s_1. Again the induction hypothesis can be applied to the premise $\langle S,\ s\rangle\to s_0$, and from $\langle S,\ s\rangle\to s_1$ we get $s_0 = s_1$. Thus we have

$$\langle \texttt{while } b \texttt{ do } S,\ s_0\rangle\to s' \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s_0\rangle\to s''$$

Since $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s'$ is a premise of (the instance of) [while$^{\text{tt}}_{\text{ns}}$], we can apply the induction hypothesis to it. From $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s_0\rangle \to s''$ we therefore get $s' = s''$ as required.

The case [while$^{\text{ff}}_{\text{ns}}$]: Straightforward. □

Exercise 2.10 (*)

Prove that $\texttt{repeat}\ S\ \texttt{until}\ b$ (as defined in Exercise 2.7) is semantically equivalent to S; $\texttt{while}\ \neg b\ \texttt{do}\ S$. Argue that this means that the extended semantics is deterministic. □

It is worth observing that we could not prove Theorem 2.9 using structural induction on the statement S. The reason is that the rule [while$^{\text{tt}}_{\text{ns}}$] defines the semantics of $\texttt{while}\ b\ \texttt{do}\ S$ in terms of itself. Structural induction works fine when the semantics is defined *compositionally* (as, e.g., $\mathcal{A}$ and $\mathcal{B}$ in Chapter 1). But the natural semantics of Table 2.1 is *not* defined compositionally because of the rule [while$^{\text{tt}}_{\text{ns}}$].

Basically, induction on the shape of derivation trees is a kind of structural induction on the derivation trees: In the *base case*, we show that the property holds for the simple derivation trees. In the *induction step*, we assume that the property holds for the immediate constituents of a derivation tree and show that it also holds for the composite derivation tree.

The Semantic Function $\mathcal{S}_{\text{ns}}$

The *meaning* of statements can now be summarized as a (partial) function from **State** to **State**. We define

$$\mathcal{S}_{\text{ns}}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

and this means that for every statement S we have a partial function

$$\mathcal{S}_{\text{ns}}[\![S]\!] \in \mathbf{State} \hookrightarrow \mathbf{State}.$$

It is given by

$$\mathcal{S}_{\text{ns}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S,\ s\rangle \to s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

Note that $\mathcal{S}_{\text{ns}}$ is a well-defined partial function because of Theorem 2.9. The need for partiality is demonstrated by the statement `while true do skip` that always loops (see Exercise 2.4); we then have

$$\mathcal{S}_{\mathrm{ns}}[\![\texttt{while true do skip}]\!]\; s = \underline{\mathrm{undef}}$$

for all states s.

Exercise 2.11

The semantics of arithmetic expressions is given by the function $\mathcal{A}$. We can also use an operational approach and define a natural semantics for the arithmetic expressions. It will have two kinds of configurations:

$\langle a,\, s\rangle$	denoting that a has to be evaluated in state s, and
z	denoting the final value (an element of $\mathbf{Z}$).

The transition relation $\to_{\mathrm{Aexp}}$ has the form

$$\langle a,\, s\rangle \to_{\mathrm{Aexp}} z$$

where the idea is that a evaluates to z in state s. Some example axioms and rules are

$$\langle n,\, s\rangle \to_{\mathrm{Aexp}} \mathcal{N}[\![n]\!]$$

$$\langle x,\, s\rangle \to_{\mathrm{Aexp}} s\ x$$

$$\frac{\langle a_1,\, s\rangle \to_{\mathrm{Aexp}} z_1,\ \langle a_2,\, s\rangle \to_{\mathrm{Aexp}} z_2}{\langle a_1 + a_2,\, s\rangle \to_{\mathrm{Aexp}} z} \quad \text{where } z = z_1 + z_2$$

Complete the specification of the transition system. Use structural induction on **Aexp** to prove that the meaning of a defined by this relation is the same as that defined by $\mathcal{A}$. □

Exercise 2.12

In a similar, way we can specify a natural semantics for the boolean expressions. The transitions will have the form

$$\langle b,\, s\rangle \to_{\mathrm{Bexp}} t$$

where $t \in \mathbf{T}$. Specify the transition system and prove that the meaning of b defined in this way is the same as that defined by $\mathcal{B}$. □

Exercise 2.13

Determine whether or not semantic equivalence of S_1 and S_2 amounts to $\mathcal{S}_{\mathrm{ns}}[\![S_1]\!] = \mathcal{S}_{\mathrm{ns}}[\![S_2]\!]$. □

[ass$_{\text{sos}}$]	$\langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
[skip$_{\text{sos}}$]	$\langle \texttt{skip},\ s\rangle \Rightarrow s$
[comp$^1_{\text{sos}}$]	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$
[comp$^2_{\text{sos}}$]	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
[if$^{\text{tt}}_{\text{sos}}$]	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
[if$^{\text{ff}}_{\text{sos}}$]	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
[while$_{\text{sos}}$]	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow$ $\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle$

Table 2.2 Structural operational semantics for **While**

2.2 Structural Operational Semantics

In structural operational semantics, the emphasis is on the *individual steps* of the execution; that is, the execution of assignments and tests. The transition relation has the form

$$\langle S,\ s\rangle \Rightarrow \gamma$$

where γ either is of the form $\langle S',\ s'\rangle$ or of the form s'. The transition expresses the *first* step of the execution of S from state s. There are two possible outcomes:

– If γ is of the form $\langle S',\ s'\rangle$, then the execution of S from s is *not* completed and the remaining computation is expressed by the intermediate configuration $\langle S',\ s'\rangle$.

– If γ is of the form s', then the execution of S from s *has* terminated and the final state is s'.

We shall say that $\langle S,\ s\rangle$ is *stuck* if there is no γ such that $\langle S,\ s\rangle \Rightarrow \gamma$.

The definition of $\Rightarrow$ is given by the axioms and rules of Table 2.2, and the general form of these is as in the previous section. Axioms [ass$_{\text{sos}}$] and [skip$_{\text{sos}}$] have not changed at all because the assignment and `skip` statements are fully executed in one step.

The rules [comp$^1_{\text{sos}}$] and [comp$^2_{\text{sos}}$] express that to execute $S_1;S_2$ in state s we first execute S_1 one step from s. Then there are two possible outcomes:

– If the execution of S_1 has not been completed, we have to complete it before embarking on the execution of S_2.

– If the execution of S_1 has been completed, we can start on the execution of S_2.

The first case is captured by the rule $[\mathrm{comp}^1_{\mathrm{sos}}]$: if the result of executing the first step of $\langle S,\ s\rangle$ is an intermediate configuration $\langle S'_1,\ s'\rangle$, then the next configuration is $\langle S'_1;S_2,\ s'\rangle$, showing that we have to complete the execution of S_1 before we can start on S_2. The second case above is captured by the rule $[\mathrm{comp}^2_{\mathrm{sos}}]$: if the result of executing S_1 from s is a final state s', then the next configuration is $\langle S_2,\ s'\rangle$, so that we can now start on S_2.

From the axioms $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}]$ and $[\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}]$ we see that the first step in executing a conditional is to perform the test and to select the appropriate branch. Finally, the axiom $[\mathrm{while}_{\mathrm{sos}}]$ shows that the first step in the execution of the `while`-construct is to unfold it one level; that is, to rewrite it as a conditional. The test will therefore be performed in the second step of the execution (where one of the axioms for the `if`-construct is applied). We shall see an example of this shortly.

A *derivation sequence* of a statement S starting in state s is either

1. a *finite* sequence

   $$\gamma_0,\ \gamma_1,\ \gamma_2,\ \cdots,\ \gamma_{\mathrm{k}}$$

   which is sometimes written

   $$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{\mathrm{k}}$$

   consisting of configurations satisfying $\gamma_0 = \langle S,\ s\rangle$, $\gamma_{\mathrm{i}} \Rightarrow \gamma_{\mathrm{i+1}}$ for $0 \le \mathrm{i} < \mathrm{k}$, where $\mathrm{k} \ge 0$ and where γ_{k} is either a terminal configuration or a stuck configuration, or it is

2. an *infinite* sequence

   $$\gamma_0,\ \gamma_1,\ \gamma_2,\ \cdots$$

   which is sometimes written

   $$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots$$

   consisting of configurations satisfying $\gamma_0 = \langle S,\ s\rangle$ and $\gamma_{\mathrm{i}} \Rightarrow \gamma_{\mathrm{i+1}}$ for $0 \le \mathrm{i}$.

We shall write $\gamma_0 \Rightarrow^{\mathrm{i}} \gamma_{\mathrm{i}}$ to indicate that there are i steps in the execution from γ_0 to γ_{i}, and we write $\gamma_0 \Rightarrow^* \gamma_{\mathrm{i}}$ to indicate that there are a finite number of steps. Note that $\gamma_0 \Rightarrow^{\mathrm{i}} \gamma_{\mathrm{i}}$ and $\gamma_0 \Rightarrow^{*} \gamma_{\mathrm{i}}$ need *not* be derivation sequences: they will be so if and only if γ_{i} is either a terminal configuration or a stuck configuration.

Example 2.14

Consider the statement

$$(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z}$$

of Chapter 1, and let s_0 be the state that maps all variables except `x` and `y` to $\mathbf{0}$ and that has $s_0\ \mathtt{x} = \mathbf{5}$ and $s_0\ \mathtt{y} = \mathbf{7}$. We then have the derivation sequence

$$\begin{aligned}
&\langle(\mathtt{z} := \mathtt{x};\ {}^{-}\mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0\rangle \\
&\Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle \\
&\Rightarrow \langle \mathtt{y} := \mathtt{z},\ (s_0[\mathtt{z}\mapsto\mathbf{5}])[\mathtt{x}\mapsto\mathbf{7}]\rangle \\
&\Rightarrow ((s_0[\mathtt{z}\mapsto\mathbf{5}])[\mathtt{x}\mapsto\mathbf{7}])[\mathtt{y}\mapsto\mathbf{5}]
\end{aligned}$$

Corresponding to *each* of these steps, we have *derivation trees* explaining why they take place. For the first step

$$\langle(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0\rangle \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle$$

the derivation tree is

$$\dfrac{\dfrac{\langle \mathtt{z} := \mathtt{x},\ s_0\rangle \Rightarrow s_0[\mathtt{z}\mapsto\mathbf{5}]}{\langle \mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y},\ s_0\rangle \Rightarrow \langle \mathtt{x} := \mathtt{y},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle}}{\langle(\mathtt{z} := \mathtt{x};\ \mathtt{x} := \mathtt{y});\ \mathtt{y} := \mathtt{z},\ s_0\rangle \Rightarrow \langle \mathtt{x} := \mathtt{y};\ \mathtt{y} := \mathtt{z},\ s_0[\mathtt{z}\mapsto\mathbf{5}]\rangle}$$

and it has been constructed from the axiom $[\mathrm{ass}_{\mathrm{sos}}]$ and the rules $[\mathrm{comp}^1_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$. The derivation tree for the second step is constructed in a similar way using only $[\mathrm{ass}_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$, and for the third step it simply is an instance of $[\mathrm{ass}_{\mathrm{sos}}]$. $\square$

Example 2.15

Assume that $s\ \mathtt{x} = \mathbf{3}$. The first step of execution from the configuration

$$\langle \mathtt{y}{:=}1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y}{:=}\mathtt{y} \star \mathtt{x};\ \mathtt{x}{:=}\mathtt{x}{-}1),\ s\rangle$$

will give the configuration

$$\langle \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y}{:=}\mathtt{y} \star \mathtt{x};\ \mathtt{x}{:=}\mathtt{x}{-}1),\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle$$

This is achieved using the axiom [ass_{sos}] and the rule [$\text{comp}^2_{\text{sos}}$] as shown by the derivation tree

$$\frac{\langle \texttt{y:=1},\ s\rangle \Rightarrow s[\texttt{y}\mapsto\mathbf{1}]}{\begin{array}{l}\langle \texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s\rangle \Rightarrow\\ \qquad\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\end{array}}$$

The next step of the execution will rewrite the loop as a conditional using the axiom [$\text{while}_{\text{sos}}$] so we get the configuration

$$\begin{array}{l}\langle \texttt{if}\ \neg(\texttt{x=1})\ \texttt{then}\ \ ((\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1});\\ \qquad\qquad\qquad\qquad \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1}))\\ \qquad\qquad\qquad \texttt{else skip},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\end{array}$$

The following step will perform the test and yields (according to [$\text{if}^{\text{tt}}_{\text{sos}}$]) the configuration

$$\langle(\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}); \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle$$

We can then use [ass_{sos}], [$\text{comp}^2_{\text{sos}}$], and [$\text{comp}^1_{\text{sos}}$] to obtain the configuration

$$\langle \texttt{x:=x}-\texttt{1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{3}]\rangle$$

as is verified by the derivation tree

$$\frac{\dfrac{\langle \texttt{y:=y}\star\texttt{x},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\Rightarrow s[\texttt{y}\mapsto\mathbf{3}]}{\langle \texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle\Rightarrow\langle \texttt{x:=x}-\texttt{1},\ s[\texttt{y}\mapsto\mathbf{3}]\rangle}}{\begin{array}{l}\langle(\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}); \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{1}]\rangle \Rightarrow\\ \qquad\langle \texttt{x:=x}-\texttt{1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{3}]\rangle\end{array}}$$

Using [ass_{sos}] and [$\text{comp}^2_{\text{sos}}$], the next configuration will then be

$$\langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\ \star\ \texttt{x; x:=x}-\texttt{1)},\ s[\texttt{y}\mapsto\mathbf{3}][\texttt{x}\mapsto\mathbf{2}]\rangle$$

Continuing in this way, we eventually reach the final state $s[\texttt{y}\mapsto\mathbf{6}][\texttt{x}\mapsto\mathbf{1}]$. □

Exercise 2.16

Construct a derivation sequence for the statement

$$\texttt{z:=0; while y}\leq\texttt{x do (z:=z+1; x:=x}-\texttt{y)}$$

when executed in a state where `x` has the value **17** and `y` has the value **5**. Determine a state s such that the derivation sequence obtained for the statement above and s is infinite. □

Given a statement S in the language **While** and a state s, it is always possible to find *at least one* derivation sequence that starts in the configuration $\langle S,\ s\rangle$: simply apply axioms and rules forever or until a terminal or stuck configuration is reached. Inspection of Table 2.2 shows that there are no stuck configurations in **While**, and Exercise 2.22 below will show that there is in fact only one derivation sequence that starts with $\langle S,\ s\rangle$. However, some of the constructs considered in Chapter 3 that extend **While** will have configurations that are stuck or more than one derivation sequence that starts in a given configuration.

In analogy with the terminology of the previous section, we shall say that the execution of a statement S on a state s

– *terminates* if and only if there is a finite derivation sequence starting with $\langle S,\ s\rangle$ and

– *loops* if and only if there is an infinite derivation sequence starting with $\langle S,\ s\rangle$.

We shall say that the execution of S on s *terminates successfully* if $\langle S,\ s\rangle \Rightarrow^* s'$ for some state s'; in **While** an execution terminates successfully if and only if it terminates because there are no stuck configurations. Finally, we shall say that a statement S *always terminates* if it terminates on all states, and *always loops* if it loops on all states.

Exercise 2.17

Extend **While** with the construct `repeat` S `until` b and specify a structural operational semantics for it. (The semantics for the `repeat`-construct is not allowed to rely on the existence of a `while`-construct.) □

Exercise 2.18

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify the structural operational semantics for it. Hint: You may need to assume that you have an "inverse" to $\mathcal{N}$ so that there is a numeral for each number that may arise during the computation. (The semantics for the `for`-construct is not allowed to rely on the existence of a `while`-construct.) □

Properties of the Semantics

For structural operational semantics, it is often useful to conduct proofs by induction on the *lengths* of the finite derivation sequences considered. The proof technique may be summarized as follows:

Induction on the Length of Derivation Sequences

1: Prove that the property holds for all derivation sequences of length 0.

2: Prove that the property holds for all finite derivation sequences: Assume that the property holds for all derivation sequences of length at most k (this is called the *induction hypothesis*) and show that it holds for derivation sequences of length k+1.

The induction step of a proof following this principle will often be done by inspecting either

– the structure of the syntactic element or

– the derivation tree validating the first transition of the derivation sequence.

Note that the proof technique is a simple application of mathematical induction.

To illustrate the use of the proof technique, we shall prove the following lemma (to be used in the next section). Intuitively, the lemma expresses that the execution of a composite construct $S_1;S_2$ can be split into two parts, one corresponding to S_1 and the other corresponding to S_2.

Lemma 2.19

If $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$, then there exists a state s' and natural numbers k_1 and k_2 such that $\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1} s'$ and $\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_2} s''$, where $\mathrm{k} = \mathrm{k}_1+\mathrm{k}_2$.

Proof: The proof is by induction on the number k; that is, by induction on the length of the derivation sequence $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$.

If k = 0, then the result holds vacuously (because $\langle S_1;S_2,\ s\rangle$ and s'' are different).

For the induction step, we assume that the lemma holds for $\mathrm{k} \leq \mathrm{k}_0$, and we shall prove it for k_0+1. So assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}_0+1} s''$$

This means that the derivation sequence can be written as

$$\langle S_1;S_2,\ s\rangle \Rightarrow \gamma \Rightarrow^{\mathrm{k}_0} s''$$

for some configuration γ. Now one of two cases applies depending on which of the two rules $[\mathrm{comp}^1_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$ was used to obtain $\langle S_1;S_2,\ s\rangle \Rightarrow \gamma$.

In the first case, where $[\mathrm{comp}^1_{\mathrm{sos}}]$ is used, we have $\gamma = \langle S_1'; S_2, s'\rangle$ and

$$\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle$$

because

$$\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle$$

We therefore have

$$\langle S_1';S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_0}\ s''$$

and the induction hypothesis can be applied to this derivation sequence because it is shorter than the one with which we started. This means that there is a state s_0 and natural numbers k_1 and k_2 such that

$$\langle S_1',\ s'\rangle \Rightarrow^{\mathrm{k}_1}\ s_0 \text{ and } \langle S_2,\ s_0\rangle \Rightarrow^{\mathrm{k}_2}\ s''$$

where k_1+k_2=k_0. Using that $\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle$ and $\langle S_1',\ s'\rangle \Rightarrow^{\mathrm{k}_1}\ s_0$, we get

$$\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1+1}\ s_0$$

We have already seen that $\langle S_2,\ s_0\rangle \Rightarrow^{\mathrm{k}_2}\ s''$, and since $(\mathrm{k}_1+1)+\mathrm{k}_2 = \mathrm{k}_0+1$, we have proved the required result.

The second possibility is that [comp$^2_{\mathrm{sos}}$] has been used to obtain the derivation $\langle S_1;S_2,\ s\rangle \Rightarrow \gamma$. Then we have

$$\langle S_1,\ s\rangle \Rightarrow s'$$

and γ is $\langle S_2,\ s'\rangle$ so that

$$\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_0}\ s''$$

The result now follows by choosing k_1=1 and k_2=k_0. □

Exercise 2.20

Suppose that $\langle S_1;S_2,\ s\rangle \Rightarrow^* \langle S_2,\ s'\rangle$. Show that it is *not* necessarily the case that $\langle S_1,\ s\rangle \Rightarrow^* s'$. □

Exercise 2.21 (Essential)

Prove that

$$\text{if } \langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}}\ s' \text{ then } \langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}}\ \langle S_2,\ s'\rangle$$

I.e., the execution of S_1 is not influenced by the statement following it. □

In the previous section, we defined a notion of determinism based on the natural semantics. For the structural operational semantics, we define the similar notion as follows. The semantics of Table 2.2 is *deterministic* if for all choices of S, s, γ, and γ' we have that

$$\langle S,\ s\rangle \Rightarrow \gamma \text{ and } \langle S,\ s\rangle \Rightarrow \gamma' \text{ imply } \gamma = \gamma'$$

Exercise 2.22 (Essential)

Show that the structural operational semantics of Table 2.2 is deterministic. Deduce that there is exactly one derivation sequence starting in a configuration $\langle S,\ s\rangle$. Argue that a statement S of **While** cannot both terminate and loop on a state s and hence cannot both be always terminating and always looping. □

In the previous section, we defined a notion of two statements S_1 and S_2 being semantically equivalent. The similar notion can be defined based on the structural operational semantics: S_1 and S_2 are *semantically equivalent* if for all states s

- $\langle S_1,\ s\rangle \Rightarrow^* \gamma$ if and only if $\langle S_2,\ s\rangle \Rightarrow^* \gamma$, whenever γ is a configuration that is either stuck or terminal, and
- there is an infinite derivation sequence starting in $\langle S_1,\ s\rangle$ if and only if there is one starting in $\langle S_2,\ s\rangle$.

Note that in the first case the lengths of the two derivation sequences may be different.

Exercise 2.23

Show that the following statements of **While** are semantically equivalent in the sense above:

- S;`skip` and S
- `while` b `do` S and `if` b `then` (S; `while` b `do` S) `else skip`
- S_1;(S_2;S_3) and (S_1;S_2);S_3

You may use the result of Exercise 2.22. Discuss to what extent the notion of semantic equivalence introduced above is the same as that defined from the natural semantics. □

Exercise 2.24

Prove that `repeat` S `until` b (as defined in Exercise 2.17) is semantically equivalent to S; `while` $\neg$ b `do` S. □

The Semantic Function $\boldsymbol{\mathcal{S}}_{\textbf{sos}}$

As in the previous section, the *meaning* of statements can be summarized by a (partial) function from **State** to **State**:

$$\mathcal{S}_{\text{sos}}\colon \textbf{Stm} \to (\textbf{State} \hookrightarrow \textbf{State})$$

It is given by

$$\mathcal{S}_{\text{sos}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S,\ s\rangle \Rightarrow^* s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

The well-definedness of the definition follows from Exercise 2.22.

Exercise 2.25

Determine whether or not semantic equivalence of S_1 and S_2 amounts to $\mathcal{S}_{\text{sos}}[\![S_1]\!] = \mathcal{S}_{\text{sos}}[\![S_2]\!]$. □

2.3 An Equivalence Result

We have given two definitions of the semantics of **While** and we shall now address the question of their equivalence.

Theorem 2.26

For every statement S of **While**, we have $\mathcal{S}_{\text{ns}}[\![S]\!] = \mathcal{S}_{\text{sos}}[\![S]\!]$.

This result expresses two properties:

- If the execution of S from some state terminates in one of the semantics, then it also terminates in the other and the resulting states will be equal.
- If the execution of S from some state loops in one of the semantics, then it will also loop in the other.

It should be fairly obvious that the first property follows from the theorem because there are no stuck configurations in the structural operational semantics of **While**. For the other property, suppose that the execution of S on state s loops in one of the semantics. If it terminates in the other semantics, we have a contradiction with the first property because both semantics are deterministic (Theorem 2.9 and Exercise 2.22). Hence S will have to loop on state s also in the other semantics.

The theorem is proved in two stages, as expressed by Lemma 2.27 and Lemma 2.28 below. We shall first prove Lemma 2.27.

Lemma 2.27

For every statement S of **While** and states s and s' we have

$$\langle S,\, s\rangle \rightarrow s' \text{ implies } \langle S,\, s\rangle \Rightarrow^* s'$$

So if the execution of S from s terminates in the natural semantics, then it will terminate in the same state in the structural operational semantics.

Proof: The proof proceeds by induction on the shape of the derivation tree for $\langle S,\, s\rangle \rightarrow s'$.

The case [ass$_{\text{ns}}$]: We assume that

$$\langle x := a,\, s\rangle \rightarrow s[x{\mapsto}\mathcal{A}[\![a]\!]s]$$

From [ass$_{\text{sos}}$], we get the required

$$\langle x := a,\, s\rangle \Rightarrow s[x{\mapsto}\mathcal{A}[\![a]\!]s]$$

The case [skip$_{\text{ns}}$]: Analogous.

The case [comp$_{\text{ns}}$]: Assume that

$$\langle S_1;S_2,\, s\rangle \rightarrow s''$$

because

$$\langle S_1,\, s\rangle \rightarrow s' \text{ and } \langle S_2,\, s'\rangle \rightarrow s''$$

The induction hypothesis can be applied to both of the premises $\langle S_1,\, s\rangle \rightarrow s'$ and $\langle S_2,\, s'\rangle \rightarrow s''$ and gives

$$\langle S_1,\, s\rangle \Rightarrow^* s' \text{ and } \langle S_2,\, s'\rangle \Rightarrow^* s''$$

From Exercise 2.21, we get

$$\langle S_1;S_2,\, s\rangle \Rightarrow^* \langle S_2,\, s'\rangle$$

and thereby $\langle S_1;S_2,\, s\rangle \Rightarrow^* s''$.

The case [if$^{\text{tt}}_{\text{ns}}$]: Assume that

$$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\, s\rangle \rightarrow s'$$

because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1,\, s\rangle \rightarrow s'$$

Since $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, we get

$$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\, s\rangle \Rightarrow \langle S_1,\, s\rangle \Rightarrow^* s'$$

where the first relationship comes from [$\text{if}^{\text{tt}}_{\text{sos}}$] and the second from the induction hypothesis applied to the premise $\langle S_1,\, s\rangle \to s'$.

The case [$\text{if}^{\text{ff}}_{\text{ns}}$]: Analogous.

The case [$\text{while}^{\text{tt}}_{\text{ns}}$]: Assume that

$$\langle \texttt{while } b \texttt{ do } S,\, s\rangle \to s''$$

because

$$\mathcal{B}[\![b]\!]s = \textbf{tt},\ \langle S,\, s\rangle \to s' \text{ and } \langle \texttt{while } b \texttt{ do } S,\, s'\rangle \to s''$$

The induction hypothesis can be applied to both of the premises $\langle S,\, s\rangle \to s'$ and $\langle \texttt{while } b \texttt{ do } S,\, s'\rangle \to s''$ and gives

$$\langle S,\, s\rangle \Rightarrow^* s' \text{ and } \langle \texttt{while } b \texttt{ do } S,\, s'\rangle \Rightarrow^* s''$$

Using Exercise 2.21, we get

$$\langle S;\, \texttt{while } b \texttt{ do } S,\, s\rangle \Rightarrow^* s''$$

Using [$\text{while}_{\text{sos}}$] and [$\text{if}^{\text{tt}}_{\text{sos}}$] (with $\mathcal{B}[\![b]\!]s = \textbf{tt}$), we get the first two steps of

$$\begin{aligned}
&\langle \texttt{while } b \texttt{ do } S,\, s\rangle\\
&\Rightarrow \langle \texttt{if } b \texttt{ then } (S;\, \texttt{while } b \texttt{ do } S) \texttt{ else skip},\, s\rangle\\
&\Rightarrow \langle S;\, \texttt{while } b \texttt{ do } S,\, s\rangle\\
&\Rightarrow^* s''
\end{aligned}$$

and we have already argued for the last part.

The case [$\text{while}^{\text{ff}}_{\text{ns}}$]: Straightforward. □

This completes the proof of Lemma 2.27. The second part of the theorem follows from Lemma 2.28.

Lemma 2.28

For every statement S of **While**, states s and s' and natural number k, we have that

$$\langle S,\, s\rangle \Rightarrow^{\text{k}} s' \text{ implies } \langle S,\, s\rangle \to s'.$$

So if the execution of S from s terminates in the structural operational semantics, then it will terminate in the same state in the natural semantics.

Proof: The proof proceeds by induction on the length of the derivation sequence $\langle S,\ s\rangle \Rightarrow^{\mathrm{k}} s'$; that is, by induction on k.

If k=0, then the result holds vacuously.

To prove the induction step we assume that the lemma holds for $\mathrm{k} \leq \mathrm{k}_0$, and we shall then prove that it holds for k_0+1. We proceed by cases on how the first step of $\langle S,\ s\rangle \Rightarrow^{\mathrm{k}_0+1} s'$ is obtained; that is, by inspecting the derivation tree for the first step of computation in the structural operational semantics.

The case [$\mathrm{ass}_{\mathrm{sos}}$]: Straightforward (and $\mathrm{k}_0 = 0$).

The case [$\mathrm{skip}_{\mathrm{sos}}$]: Straightforward (and $\mathrm{k}_0 = 0$).

The cases [$\mathrm{comp}^1_{\mathrm{sos}}$] and [$\mathrm{comp}^2_{\mathrm{sos}}$]: In both cases, we assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}_0+1} s''$$

We can now apply Lemma 2.19 and get that there exists a state s' and natural numbers k_1 and k_2 such that

$$\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1} s' \text{ and } \langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_2} s''$$

where k_1+k_2=k_0+1. The induction hypothesis can now be applied to each of these derivation sequences because $\mathrm{k}_1 \leq \mathrm{k}_0$ and $\mathrm{k}_2 \leq \mathrm{k}_0$. So we get

$$\langle S_1,\ s\rangle \to s' \text{ and } \langle S_2,\ s'\rangle \to s''$$

Using [$\mathrm{comp}_{\mathrm{ns}}$], we now get the required $\langle S_1;S_2,\ s\rangle \to s''$.

The case [$\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}$]: Assume that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_0} s'$$

The induction hypothesis can be applied to the derivation $\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_0} s'$ and gives

$$\langle S_1,\ s\rangle \to s'$$

The result now follows using [$\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}$].

The case [$\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}$]: Analogous.

The case [$\mathrm{while}_{\mathrm{sos}}$]: We have

$$\begin{aligned}&\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle\\ &\Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle\\ &\Rightarrow^{\mathrm{k}_0} s''\end{aligned}$$

The induction hypothesis can be applied to the k_0 last steps of the derivation sequence and gives

$$\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle \to s''$$

and from Lemma 2.5 we get the required

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$$

Proof of Theorem 2.26: For an arbitrary statement S and state s, it follows from Lemmas 2.27 and 2.28 that if $\mathcal{S}_{\rm ns}[\![S]\!]s = s'$ then $\mathcal{S}_{\rm sos}[\![S]\!]s = s'$ and vice versa. This suffices for showing that the functions $\mathcal{S}_{\rm ns}[\![S]\!]$ and $\mathcal{S}_{\rm sos}[\![S]\!]$ must be equal: if one is defined on a state s, then so is the other, and therefore if one is not defined on a state s, then neither is the other. $\square$

Exercise 2.29

Consider the extension of the language **While** with the statement `repeat` S `until` b. The natural semantics of the construct was considered in Exercise 2.7 and the structural operational semantics in Exercise 2.17. Modify the proof of Theorem 2.26 so that the theorem applies to the extended language. $\square$

Exercise 2.30

Consider the extension of the language **While** with the statement `for` $x := a_1$ `to` a_2 `do` S. The natural semantics of the construct was considered in Exercise 2.8 and the structural operational semantics in Exercise 2.18. Modify the proof of Theorem 2.26 so that the theorem applies to the extended language. $\square$

The proof technique employed in the proof of Theorem 2.26 may be summarized as follows:

Proof Summary for While:

Equivalence of two Operational Semantics

1: Prove by *induction on the shape of derivation trees* that for each derivation tree in the natural semantics there is a corresponding finite derivation sequence in the structural operational semantics.

2: Prove by *induction on the length of derivation sequences* that for each finite derivation sequence in the structural operational semantics there is a corresponding derivation tree in the natural semantics.

When proving the equivalence of two operational semantics for a language with additional programming constructs, one may need to amend the above technique above. One reason is that the equivalence result may have to be expressed differently from that of Theorem 2.26 (as will be the case if the extended language is non-deterministic).

3
More on Operational Semantics

For the **While** language, the choice between the structural operational semantics and the natural semantics is largely a matter of taste — as expressed by Theorem 2.26, the two semantics are equivalent. For other language constructs, the situation may be more complex: sometimes it is easy to specify the semantics in one style but difficult or even impossible in the other. Also, there are situations where equivalent semantics can be specified in the two styles but where one of the semantics is to be preferred because of a particular application.

In the first part of this chapter, we shall study extensions of the language **While** with non-sequential constructs such as abortion, non-determinism, and parallelism, and in each case we shall consider how to modify the operational semantics of the previous chapter. In the second part, we shall extend **While** with blocks and local variable and procedure declarations. This leads to the introduction of important semantics concepts such as environments and store, and we shall illustrate this in the setting of a natural semantics.

3.1 Non-sequential Language Constructs

In order to illustrate the power and weakness of the two approaches to operational semantics, we shall in this section consider various extensions of the language **While**. For each extension, we shall discuss how to modify the two styles of operational semantics.

Abortion

We first extend **While** with the simple statement `abort`. The idea is that `abort` *stops* the execution of the complete program. This means that `abort` behaves differently from `while true do skip` in that it causes the execution to stop rather than loop. Also, `abort` behaves differently from `skip` because a statement following `abort` will never be executed, whereas one following `skip` certainly will.

Formally, the new syntax of statements is given by

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\ & \mid & \texttt{while } b \texttt{ do } S \mid \texttt{abort} \end{array}$$

We shall not repeat the definitions of the sets of configurations but tacitly assume that they are modified so as to correspond to the extended syntax. The task that remains, therefore, is to define the new transition relations $\to$ and $\Rightarrow$.

The fact that `abort` stops the execution of the program may be modelled by ensuring that the configurations of the form $\langle\texttt{abort}, s\rangle$ are *stuck*. Therefore the *natural semantics* of the extended language is still defined by the transition relation $\to$ of Table 2.1. So although the language and thereby the set of configurations have been extended, we do not modify the definition of the transition relation. Similarly, the *structural operational semantics* of the extended language is still defined by Table 2.2.

From the structural operational semantics point of view, it is clear now that `abort` and `skip` cannot be semantically equivalent. This is because

$$\langle\texttt{skip}, s\rangle \Rightarrow s$$

is the only derivation sequence for `skip` starting in s and

$$\langle\texttt{abort}, s\rangle$$

is the only derivation sequence for `abort` starting in s. Similarly, `abort` cannot be semantically equivalent to `while true do skip` because

$$\begin{array}{l} \langle\texttt{while true do skip}, s\rangle \\ \Rightarrow \langle\texttt{if true then (skip; while true do skip) else skip}, s\rangle \\ \Rightarrow \langle\texttt{skip; while true do skip}, s\rangle \\ \Rightarrow \langle\texttt{while true do skip}, s\rangle \\ \Rightarrow \cdots \end{array}$$

is an infinite derivation sequence for `while true do skip`, whereas `abort` has none. Thus we shall claim that the structural operational semantics captures the informal explanation given earlier.

From the natural semantics point of view, it is also clear that `skip` and `abort` cannot be semantically equivalent. However, somewhat surprisingly, it turns out that `while true do skip` and `abort` *are* semantically equivalent! The reason is that in the natural semantics we are only concerned with executions that terminate properly. So if we do not have a derivation tree for $\langle S, s\rangle \to s'$, then we cannot tell whether it is because we entered a stuck configuration or a looping execution. We can summarize this as follows:

Natural Semantics versus Structural Operational Semantics

- In a natural semantics, we cannot distinguish between *looping* and *abnormal termination.*
- In a structural operational semantics, *looping* is reflected by infinite derivation sequences and *abnormal termination* by finite derivation sequences ending in a stuck configuration.

We should note, however, that if abnormal termination is modelled by "normal termination" in a special error configuration (included in the set of terminal configurations), then we can distinguish among the three statements in both semantic styles.

Exercise 3.1

Theorem 2.26 expresses that the natural semantics and the structural operational semantics of **While** are equivalent. Discuss whether or not a similar result holds for **While** extended with `abort`. □

Exercise 3.2

Extend **While** with the statement

$$\texttt{assert}\ b\ \texttt{before}\ S$$

The idea is that if b evaluates to true, then we execute S, and otherwise the execution of the complete program aborts. Extend the structural operational semantics of Table 2.2 to express this (without assuming that **While** contains the `abort`-statement). Show that `assert true before` S is semantically equivalent to S but that `assert false before` S is equivalent to neither `while true do skip` nor `skip`. □

Non-determinism

The second extension of **While** has statements given by

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1 \;;\; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid S_1\ \texttt{or}\ S_2 \end{array}$$

The idea here is that in S_1 `or` S_2 we can non-deterministically choose to execute either S_1 or S_2. So we shall expect that execution of the statement

$$\texttt{x := 1 or (x := 2; x := x + 2)}$$

could result in a state where `x` has the value **1**, but it could as well result in a state where `x` has the value **4**.

When specifying the *natural semantics*, we extend Table 2.1 with the following two rules:

$$[\text{or}^1_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \to s'}{\langle S_1\ \texttt{or}\ S_2,\ s\rangle \to s'}$$

$$[\text{or}^2_{\text{ns}}] \qquad \frac{\langle S_2,\ s\rangle \to s'}{\langle S_1\ \texttt{or}\ S_2,\ s\rangle \to s'}$$

Corresponding to the configuration $\langle$`x := 1 or (x := 2; x := x+2)`$,\ s\rangle$, we have derivation trees for

$$\langle \texttt{x := 1 or (x := 2; x := x+2)},\ s\rangle \to s[\texttt{x}\mapsto\mathbf{1}]$$

as well as

$$\langle \texttt{x := 1 or (x := 2; x := x+2)},\ s\rangle \to s[\texttt{x}\mapsto\mathbf{4}]$$

It is important to note that if we replace `x := 1` by `while true do skip` in the statement above, then we will only have one derivation tree, namely that for

$$\langle \texttt{(while true do skip) or (x := 2; x := x+2)},\ s\rangle \to s[\texttt{x}\mapsto\mathbf{4}]$$

Turning to the *structural operational semantics*, we shall extend Table 2.2 with the following two axioms:

$$[\text{or}^1_{\text{sos}}] \qquad \langle S_1\ \texttt{or}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$$

$$[\text{or}^2_{\text{sos}}] \qquad \langle S_1\ \texttt{or}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$$

For the statement `x := 1 or (x := 2; x := x+2)`, we have two derivation sequences:

$$\langle \texttt{x := 1 or (x := 2; x := x+2)},\ s\rangle \Rightarrow^* s[\texttt{x}\mapsto\mathbf{1}]$$

and

$$\langle \texttt{x := 1 or (x := 2; x := x+2)}, s\rangle \Rightarrow^* s[\texttt{x}\mapsto\mathbf{4}]$$

If we replace `x := 1` by `while true do skip` in the statement above, then we still have two derivation sequences. One is infinite

$$\begin{aligned}
&\langle \texttt{(while true do skip) or (x := 2; x := x+2)}, s\rangle\\
&\Rightarrow\ \langle \texttt{while true do skip}, s\rangle\\
&\Rightarrow^3 \langle \texttt{while true do skip}, s\rangle\\
&\Rightarrow \cdots
\end{aligned}$$

and the other is finite

$$\langle \texttt{(while true do skip) or (x := 2; x := x+2)}, s\rangle \Rightarrow^* s[\texttt{x}\mapsto\mathbf{4}]$$

Comparing the natural semantics and the structural operational semantics, we see that the latter can choose the "wrong" branch of the `or`-statement, whereas the first always chooses the "right" branch. This is summarized as follows:

Natural Semantics versus Structural Operational Semantics

- In a natural semantics, *non-determinism suppresses looping*, if possible.
- In a structural operational semantics, *non-determinism does not suppress looping.*

Exercise 3.3

Consider the statement

$$\texttt{x := }-\texttt{1; while x}\leq\texttt{0 do (x := x}-\texttt{1 or x := (}-\texttt{1)}\star\texttt{x)}$$

Given a state s, describe the set of final states that may result according to the natural semantics. Further, describe the set of derivation sequences that are specified by the structural operational semantics. Based on this, discuss whether or not you would regard the natural semantics as being equivalent to the structural operational semantics for this particular statement. ☐

Exercise 3.4

We shall now extend **While** with the statement

$$\texttt{random}(x)$$

and the idea is that its execution will change the value of x to be any positive natural number. Extend the natural semantics as well as the structural operational semantics to express this. Discuss whether $\texttt{random}(x)$ is a superfluous construct in the case where **While** is also extended with the `or` construct. $\square$

Parallelism

We shall now consider an extension of **While** with a parallel construct. So the syntax of expressions is given by

$$\begin{array}{rcl} S & ::= & x := a \mid \texttt{skip} \mid S_1\ ;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid S_1\ \texttt{par}\ S_2 \end{array}$$

The idea is that both statements of S_1 `par` S_2 have to be executed but that the execution can be *interleaved*. This means that a statement such as

$$\texttt{x := 1 par (x := 2; x := x+2)}$$

can give three different results for `x`, namely **4**, **1**, and **3**. If we first execute `x := 1` and then `x := 2; x := x+2`, we get the final value **4**. Alternatively, if we first execute `x := 2; x := x+2` and then `x := 1`, we get the final value **1**. Finally, we have the possibility of first executing `x := 2`, then `x := 1`, and lastly `x := x+2`, and we then get the final value **3**.

To express this in the *structural operational semantics*, we extend Table 2.2 with the following rules:

$$[\text{par}^1_{\text{sos}}] \qquad \frac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \Rightarrow \langle S_1'\ \texttt{par}\ S_2,\ s'\rangle}$$

$$[\text{par}^2_{\text{sos}}] \qquad \frac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$$

$$[\text{par}^3_{\text{sos}}] \qquad \frac{\langle S_2,\ s\rangle \Rightarrow \langle S_2',\ s'\rangle}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \Rightarrow \langle S_1\ \texttt{par}\ S_2',\ s'\rangle}$$

$$[\text{par}^4_{\text{sos}}] \qquad \frac{\langle S_2,\ s\rangle \Rightarrow s'}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s'\rangle}$$

The first two rules take account of the case where we begin by executing the first step of statement S_1. If the execution of S_1 is not fully completed, we modify the configuration so as to remember how far we have reached. Otherwise only S_2 has to be executed and we update the configuration accordingly. The last two rules are similar but for the case where we begin by executing the first step of S_2.

Using these rules, we get the following derivation sequences for the example statement:

$$\begin{array}{lll}
\langle \texttt{x := 1 par (x := 2; x := x+2)}, s\rangle & \Rightarrow & \langle \texttt{x := 2; x := x+2}, s[\texttt{x}\mapsto\mathbf{1}]\rangle \\
& \Rightarrow & \langle \texttt{x := x+2}, s[\texttt{x}\mapsto\mathbf{2}]\rangle \\
& \Rightarrow & s[\texttt{x}\mapsto\mathbf{4}]
\end{array}$$

$$\begin{array}{lll}
\langle \texttt{x := 1 par (x := 2; x := x+2)}, s\rangle & \Rightarrow & \langle \texttt{x := 1 par x := x+2}, s[\texttt{x}\mapsto\mathbf{2}]\rangle \\
& \Rightarrow & \langle \texttt{x := 1}, s[\texttt{x}\mapsto\mathbf{4}]\rangle \\
& \Rightarrow & s[\texttt{x}\mapsto\mathbf{1}]
\end{array}$$

and

$$\begin{array}{lll}
\langle \texttt{x := 1 par (x := 2; x := x+2)}, s\rangle & \Rightarrow & \langle \texttt{x := 1 par x := x+2}, s[\texttt{x}\mapsto\mathbf{2}]\rangle \\
& \Rightarrow & \langle \texttt{x := x+2}, s[\texttt{x}\mapsto\mathbf{1}]\rangle \\
& \Rightarrow & s[\texttt{x}\mapsto\mathbf{3}]
\end{array}$$

Turning to the *natural semantics*, we might start by extending Table 2.1 with the two rules

$$\frac{\langle S_1, s\rangle \rightarrow s', \langle S_2, s'\rangle \rightarrow s''}{\langle S_1 \texttt{ par } S_2, s\rangle \rightarrow s''}$$

$$\frac{\langle S_2, s\rangle \rightarrow s', \langle S_1, s'\rangle \rightarrow s''}{\langle S_1 \texttt{ par } S_2, s\rangle \rightarrow s''}$$

However, it is easy to see that this will not do because the rules only express that either S_1 is executed before S_2 or vice versa. This means that we have lost the ability to *interleave* the execution of two statements. Furthermore, it seems impossible to be able to express this in the natural semantics because we consider the execution of a statement as an atomic entity that cannot be split into smaller pieces. This may be summarized as follows:

Natural Semantics versus Structural Operational Semantics

- In a natural semantics, the execution of the immediate constituents is an *atomic entity* so we cannot express interleaving of computations.
- In a structural operational semantics, we concentrate on the *small steps* of the computation so we can easily express interleaving.

Exercise 3.5

Consider an extension of **While** that in addition to the par-construct also contains the construct

$$\texttt{protect}\ S\ \texttt{end}$$

The idea is that the statement S has to be executed as an atomic entity so that for example

$$\texttt{x := 1 par protect (x := 2; x := x+2) end}$$

only has two possible outcomes, namely **1** and **4**. Extend the structural operational semantics to express this. Can you specify a natural semantics for the extended language? ☐

Exercise 3.6

Specify a structural operational semantics for arithmetic expressions where the individual parts of an expression may be computed in parallel. Try to prove that you still obtain the result that was specified by $\mathcal{A}$. ☐

3.2 Blocks and Procedures

We now extend the language **While** with blocks containing declarations of variables and procedures. In doing so, we introduce a couple of important concepts:

– variable and procedure environments and

– locations and stores.

We shall concentrate on the natural semantics and will consider dynamic scope as well as static scope and non-recursive as well as recursive procedures.

Blocks and Simple Declarations

We first extend the language **While** with blocks containing declarations of local variables. The new language is called **Block** and its syntax is

$$\begin{array}{rcl} S & ::= & x := a \mid \texttt{skip} \mid S_1\ ;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_V\ S\ \texttt{end} \end{array}$$

where D_V is a meta-variable ranging over the syntactic category $\mathbf{Dec}_\mathrm{V}$ of *variable declarations*. The syntax of variable declarations is given by

$$D_V \quad ::= \quad \texttt{var}\ x := a;\ D_V \mid \varepsilon$$

[block$_{\text{ns}}$]	$\dfrac{\langle D_V,\ s\rangle \to_D s',\ \langle S,\ s'\rangle \to s''}{\langle \texttt{begin}\ D_V\ S\ \texttt{end},\ s\rangle \to s''[\text{DV}(D_V)\longmapsto s]}$

Table 3.1 Natural semantics for statements of **Block**

[var$_{\text{ns}}$]	$\dfrac{\langle D_V,\ s[x\mapsto\mathcal{A}[\![a]\!]s]\rangle \to_D s'}{\langle \texttt{var}\ x := a;\ D_V,\ s\rangle \to_D s'}$
[none$_{\text{ns}}$]	$\langle \varepsilon,\ s\rangle \to_D s$

Table 3.2 Natural semantics for variable declarations

where ε is the empty declaration. The idea is that the variables declared inside a block `begin` D_V S `end` are *local* to it. So in a statement such as

```
begin var y := 1;
        (x := 1;
        begin var x := 2; y := x+1 end;
        x := y+x)
end
```

the `x` in `y := x+1` relates to the local variable `x` introduced by `var x := 2`, whereas the `x` in `x := y+x` relates to the global variable `x` that is also used in the statement `x := 1`. In both cases, the `y` refers to the `y` declared in the outer block. Therefore, the statement `y := x+1` assigns `y` the value **3** rather than **2**, and the statement `x := y+x` assigns `x` the value **4** rather than **5**.

Before going into the details of how to specify the semantics, we shall define the set $\text{DV}(D_V)$ of variables declared in D_V:

$$\begin{aligned} \text{DV}(\texttt{var}\ x := a;\ D_V) &= \{x\} \cup \text{DV}(D_V) \\ \text{DV}(\varepsilon) &= \emptyset \end{aligned}$$

We next define the *natural semantics*. The idea will be to have one transition system for *each* of the syntactic categories **Stm** and **Dec**$_{\text{V}}$. For statements, the transition system is as in Table 2.1 but extended with the rule of Table 3.1 to be explained below. The transition system for variable declarations has configurations of the two forms $\langle D_V,\ s\rangle$ and s and the idea is that the transition relation $\to_D$ specifies the relationship between initial and final states as before:

$$\langle D_V,\ s\rangle \to_D s'$$

The relation $\to_D$ for variable declarations is given in Table 3.2. We generalize the substitution operation on states and write $s'[X\longmapsto s]$ for the state that is like s' except for variables in the set X, where it is as specified by s. Formally,

$$(s'[X{\longmapsto}s])\ x = \begin{cases} s\ x & \text{if } x \in X \\ s'\ x & \text{if } x \notin X \end{cases}$$

This operation is used in the rule of Table 3.1 to ensure that local variables are restored to their previous values when the block is left.

Exercise 3.7

Use the natural semantics of Table 3.1 to show that execution of the statement

```
begin var y := 1;
      (x := 1;
      begin var x := 2; y := x+1 end;
      x := y+x)
end
```

will lead to a state where `x` has the value **4**. □

It is somewhat harder to specify a *structural operational semantics* for the extended language. One approach is to replace states with a structure that is similar to the run-time stacks used in the implementation of block structured languages. Another is to extend the statements with fragments of the state. However, we shall not go further into this.

Procedures

We shall now extend the language **Block** with procedure declarations. The syntax of the language **Proc** is:

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1\ ;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_V\ D_P\ S\ \texttt{end} \mid \texttt{call}\ p \\ D_V & ::= & \texttt{var}\ x := a;\ D_V \mid \varepsilon \\ D_P & ::= & \texttt{proc}\ p\ \texttt{is}\ S;\ D_P \mid \varepsilon \end{array}$$

Here p is a meta-variable ranging over the syntactic category **Pname** of procedure names; in the concrete syntax, there need not be any difference between procedure names and variable names, but in the abstract syntax it is convenient to be able to distinguish between the two. Furthermore, D_P is a meta-variable ranging over the syntactic category **Dec**$_{\mathbf{P}}$ of *procedure declarations*.

We shall give three different semantics of this language. They differ in their choice of scope rules for variables and procedures:

– dynamic scope for variables as well as procedures,

– dynamic scope for variables but static scope for procedures, and

– static scope for variables as well as procedures.

To illustrate the difference, consider the statement

```
begin var x := 0;
          proc p is x := x ⋆ 2;
          proc q is call p;
            begin var x := 5;
                      proc p is x := x + 1;
                      call q; y := x
            end
end
```

If *dynamic scope* is used for variables as well as procedures, then the final value of `y` is **6**. The reason is that `call q` will call the *local* procedure `p`, which will update the *local* variable `x`. If we use dynamic scope for variables but *static scope* for procedures, then `y` gets the value **10**. The reason is that now `call q` will call the *global* procedure `p` and it will update the *local* variable `x`. Finally, if we use static scope for variables as well as procedures, then `y` gets the value **5**. The reason is that `call q` will now call the *global* procedure `p`, which will update the *global* variable `x` so the local variable `x` is unchanged.

Dynamic scope rules for variables and procedures. The general idea is that to execute the statement `call` p we shall execute the body of the procedure. This means that we have to keep track of the association of procedure names with procedure bodies. To facilitate this, we shall introduce the notion of a *procedure environment*. Given a procedure name, the procedure environment env_P will return the statement that is its body. So env_P is an element of

$$\mathbf{Env}_{\mathrm{P}} = \mathbf{Pname} \hookrightarrow \mathbf{Stm}$$

The next step will be to extend the natural semantics to take the environment into account. We shall extend the transition system for statements to have transitions of the form

$$env_P \vdash \langle S,\, s\rangle \rightarrow s'$$

$[\text{ass}_{\text{ns}}]$	$env_P \vdash \langle x := a,\ s\rangle \to s[x\mapsto\mathcal{A}[\![a]\!]s]$	
$[\text{skip}_{\text{ns}}]$	$env_P \vdash \langle \texttt{skip},\ s\rangle \to s$	
$[\text{comp}_{\text{ns}}]$	$\dfrac{env_P \vdash \langle S_1,\ s\rangle \to s',\ env_P \vdash \langle S_2,\ s'\rangle \to s''}{env_P \vdash \langle S_1;S_2,\ s\rangle \to s''}$	
$[\text{if}^{\text{tt}}_{\text{ns}}]$	$\dfrac{env_P \vdash \langle S_1,\ s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{ns}}]$	$\dfrac{env_P \vdash \langle S_2,\ s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'}$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}^{\text{tt}}_{\text{ns}}]$	$\dfrac{env_P \vdash \langle S,\ s\rangle \to s',\ env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}^{\text{ff}}_{\text{ns}}]$	$env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{block}_{\text{ns}}]$	$\dfrac{\langle D_V,\ s\rangle \to_D s',\ \text{upd}_{\text{P}}(D_P,\ env_P) \vdash \langle S,\ s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\ s\rangle \to s''[\text{DV}(D_V)\longmapsto s]}$	
$[\text{call}^{\text{rec}}_{\text{ns}}]$	$\dfrac{env_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'}$	where $env_P\ p = S$

Table 3.3 Natural semantics for **Proc** with dynamic scope rules

The presence of the environment means that we can always access it and therefore get hold of the bodies of declared procedures. The result of modifying Table 2.1 to incorporate this extra information is shown in Table 3.3.

Concerning the rule for $\texttt{begin}\ D_V\ D_P\ S\ \texttt{end}$, the idea is that we update the procedure environment so that the procedures declared in D_P will be available when executing S. Given a global environment env_P and a declaration D_P, the updated procedure environment, $\text{upd}_{\text{P}}(D_P,\ env_P)$, is specified by

$$\begin{aligned}\text{upd}_{\text{P}}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_P) &= \text{upd}_{\text{P}}(D_P,\ env_P[p\mapsto S])\\ \text{upd}_{\text{P}}(\varepsilon,\ env_P) &= env_P\end{aligned}$$

As the variable declarations do not need to access the procedure environment, it is not necessary to extend the transition system for declarations with the extra component. So for variable declarations we still have transitions of the form

$$\langle D,\, s\rangle \to_D s'$$

The relation is defined as for the language **Block**; that is, by Table 3.2.

We can now complete the specification of the semantics of blocks and procedure calls. Note that in the rule [block_{ns}] of Table 3.3, we use the updated environment when executing the body of the block. In the rule [$\text{call}_{\text{ns}}^{\text{rec}}$] for procedure calls, we make use of the information provided by the environment. It follows that procedures will *always* be allowed to be recursive. Note that attempting to call a non-existing procedure will abort.

Exercise 3.8

Consider the following statement of **Proc**:

```
begin proc fac is  begin var z := x;
                         if x = 1 then skip
                         else (x := x−1; call fac; y := z⋆y)
                   end;
      (y := 1; call fac)
end
```

Construct a derivation tree for the execution of this statement from a state s, where s x = **3**. □

Exercise 3.9

Use the semantics to verify that the statement

```
begin var x := 0;
      proc p is x := x ⋆ 2;
      proc q is call p;
      begin  var x := 5;
             proc p is x := x + 1;
             call q; y := x
      end
end
```

$[\text{call}_{\text{ns}}]$	$$\frac{env'_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'}$$ where $env_P\ p = (S,\ env'_P)$
$[\text{call}^{\text{rec}}_{\text{ns}}]$	$$\frac{env'_P[p{\mapsto}(S,\ env'_P)] \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'}$$ where $env_P\ p = (S,\ env'_P)$

Table 3.4 Procedure calls in case of mixed scope rules (choose one)

considered earlier does indeed assign the expected value **6** to `y`. □

Static scope rules for procedures. We shall now modify the semantics of **Proc** to specify static scope rules for procedures. Basically this amounts to ensuring that each procedure knows which other procedures already have been introduced when it itself was declared. To provide this information, the first step will be to extend the procedure environment env_P so that procedure names are associated with their body as well as the procedure environment at the point of declaration. To this end, we define

$$\mathbf{Env}_{\mathrm{P}} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_{\mathrm{P}}$$

This definition may seem problematic because $\mathbf{Env}_{\mathrm{P}}$ is defined in terms of itself. However, this is not really a problem because a concrete procedure environment always will be built from smaller environments, starting with the empty procedure environment. The function $\mathrm{upd}_{\mathrm{P}}$ updating the procedure environment can then be redefined as

$$\begin{aligned} \mathrm{upd}_{\mathrm{P}}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_P) &= \mathrm{upd}_{\mathrm{P}}(D_P,\ env_P[p{\mapsto}(S,\ env_P)]) \\ \mathrm{upd}_{\mathrm{P}}(\varepsilon,\ env_P) &= env_P \end{aligned}$$

The semantics of variable declarations are unaffected, and so is the semantics of most of the statements. Compared with Table 3.3, we shall only need to modify the rules for procedure calls. In the case where the procedures of **Proc** are assumed to be *non-recursive*, we simply consult the procedure environment to determine the body of the procedure and the environment at the point of declaration. This is expressed by the rule $[\text{call}_{\text{ns}}]$ of Table 3.4, where we, of course, make sure to use the procedure environment env'_P when executing the body of the procedure. In the case where the procedures of **Proc** are assumed to be *recursive*, we have to make sure that occurrences of `call` p inside the body of p refer to the procedure itself. We shall therefore update the procedure environment to contain that information. This is expressed by the rule $[\text{call}^{\text{rec}}_{\text{ns}}]$

of Table 3.4. The remaining axioms and rules are as in Tables 3.3 (without [$\text{call}_{\text{ns}}^{\text{rec}}$]) and 3.2. (Clearly a choice should be made between [call_{ns}] or [$\text{call}_{\text{ns}}^{\text{rec}}$].)

Exercise 3.10

Construct a statement that illustrates the difference between the two rules for procedure calls given in Table 3.4. Validate your claim by constructing derivation trees for the executions of the statement from a suitable state. □

Exercise 3.11

Use the semantics to verify that the statement of Exercise 3.9 assigns the expected value **10** to y. □

Static scope rules for variables. We shall now modify the semantics of **Proc** to specify static scope rules for variables as well as procedures. To achieve this, we shall replace the states with two mappings: a *variable environment* that associates a *location* with each variable and a *store* that associates a value with each location. Formally, we define a variable environment env_V as an element of

$$\mathbf{Env_V} = \mathbf{Var} \rightarrow \mathbf{Loc}$$

where **Loc** is a set of locations; one may think of locations as a kind of abstract addresses. For the sake of simplicity, we shall take $\mathbf{Loc} = \mathbf{Z}$. A store *sto* is an element of

$$\mathbf{Store} = \mathbf{Loc} \cup \{\,\mathsf{next}\,\} \rightarrow \mathbf{Z}$$

where next is a special token used to hold the next free location. We shall need a function

$$\text{new}\colon \mathbf{Loc} \rightarrow \mathbf{Loc}$$

that given a location will produce the next free one. In our case, where **Loc** is **Z**, we shall simply take 'new' to be the successor function on the integers.

So rather than having a single mapping s from variables to values, we have split it into two mappings, env_V and sto, and the idea is that $s = sto \circ env_V$. To determine the value of a variable x, we shall first

– determine the location $l = env_V\ x$ associated with x and then

– determine the value $sto\ l$ associated with the location l.

Similarly, to assign a value v to a variable x, we shall first

$$[\text{var}_{\text{ns}}] \quad \frac{\langle D_V,\ env_V[x{\mapsto}l],\ sto[l{\mapsto}v][\mathsf{next}{\mapsto}\text{new } l]\rangle \to_D (env'_V,\ sto')}{\langle \mathtt{var}\ x := a;\ D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')}$$
$$\text{where } v = \mathcal{A}[\![a]\!](sto\circ env_V) \text{ and } l = sto\ \mathsf{next}$$

$$[\text{none}_{\text{ns}}] \quad \langle \varepsilon,\ env_V,\ sto\rangle \to_D (env_V,\ sto)$$

Table 3.5 Natural semantics for variable declarations using locations

– determine the location $l = env_V\ x$ associated with x and then

– update the store to have $sto\ l = v$.

The initial variable environment could for example map all variables to the location **0** and the initial store could for example map next to **1**. The variable environment (and the store) is updated by the variable declarations. The transition system for variable declarations is therefore modified to have the form

$$\langle D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')$$

because a variable declaration will modify the variable environment as well as the store. The relation is defined in Table 3.5. Note that we use *sto* next to determine the location l to be associated with x in the variable environment. Also, the store is updated to hold the correct value for l as well as next. Finally, note that the declared variables will get positive locations.

To obtain static scoping for variables, we shall extend the procedure environment to hold the variable environment at the point of declaration. Therefore env_P will now be an element of

$$\mathbf{Env}_{\text{P}} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_{\text{V}} \times \mathbf{Env}_{\text{P}}$$

The procedure environment is updated by the procedure declarations as before, the only difference being that the current variable environment is supplied as an additional parameter. The function upd_{P} is now defined by

$$\begin{aligned}\text{upd}_{\text{P}}(\mathtt{proc}\ p\ \mathtt{is}\ S;\ D_P,\ env_V,\ env_P) &= \text{upd}_{\text{P}}(D_P,\ env_V,\\ &\qquad env_P[p{\mapsto}(S,\ env_V,\ env_P)])\\ \text{upd}_{\text{P}}(\varepsilon,\ env_V,\ env_P) &= env_P\end{aligned}$$

Finally, the transition system for statements will have the form

$$env_V,\ env_P \vdash \langle S,\ sto\rangle \to sto'$$

so given a variable environment and a procedure environment, we get a relationship between an initial store and a final store. The modification of Tables 3.3 and 3.4 is rather straightforward and is given in Table 3.6. Note that in the [ass$_{\text{ns}}$]

$$\mathit{env}_V,\ \mathit{env}_P \vdash \langle x := a,\ \mathit{sto}\rangle \to \mathit{sto}[l{\mapsto}v]$$
$$\text{where } l = \mathit{env}_V\ x \text{ and } v = \mathcal{A}[\![a]\!](\mathit{sto}{\circ}\mathit{env}_V)$$

[skip$_{\text{ns}}$]

$$\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{skip},\ \mathit{sto}\rangle \to \mathit{sto}$$

[comp$_{\text{ns}}$]

$$\frac{\begin{array}{c}\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_1,\ \mathit{sto}\rangle \to \mathit{sto}' \\ \mathit{env}_V,\ \mathit{env}_P \vdash \langle S_2,\ \mathit{sto}'\rangle \to \mathit{sto}''\end{array}}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_1;S_2,\ \mathit{sto}\rangle \to \mathit{sto}''}$$

[if$^{\text{tt}}_{\text{ns}}$]

$$\frac{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_1,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \mathit{sto}\rangle \to \mathit{sto}'}$$
$$\text{if } \mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{tt}$$

[if$^{\text{ff}}_{\text{ns}}$]

$$\frac{\mathit{env}_V,\ \mathit{env}_P \vdash \langle S_2,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \mathit{sto}\rangle \to \mathit{sto}'}$$
$$\text{if } \mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{ff}$$

[while$^{\text{tt}}_{\text{ns}}$]

$$\frac{\begin{array}{c}\mathit{env}_V,\ \mathit{env}_P \vdash \langle S,\ \mathit{sto}\rangle \to \mathit{sto}', \\ \mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\ \mathit{sto}'\rangle \to \mathit{sto}''\end{array}}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\ \mathit{sto}\rangle \to \mathit{sto}''}$$
$$\text{if } \mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{tt}$$

[while$^{\text{ff}}_{\text{ns}}$]

$$\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\ \mathit{sto}\rangle \to \mathit{sto}$$
$$\text{if } \mathcal{B}[\![b]\!](\mathit{sto}{\circ}\mathit{env}_V) = \mathbf{ff}$$

[block$_{\text{ns}}$]

$$\frac{\begin{array}{c}\langle D_V,\ \mathit{env}_V,\ \mathit{sto}\rangle \to_D (\mathit{env}'_V,\ \mathit{sto}'), \\ \mathit{env}'_V,\ \mathit{env}'_P \vdash \langle S,\ \mathit{sto}'\rangle \to \mathit{sto}''\end{array}}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\ \mathit{sto}\rangle \to \mathit{sto}''}$$
$$\text{where } \mathit{env}'_P = \text{upd}_{\text{P}}(D_P,\ \mathit{env}'_V,\ \mathit{env}_P)$$

[call$_{\text{ns}}$]

$$\frac{\mathit{env}'_V,\ \mathit{env}'_P \vdash \langle S,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{call}\ p,\ \mathit{sto}\rangle \to \mathit{sto}'}$$
$$\text{where } \mathit{env}_P\ p = (S,\ \mathit{env}'_V,\ \mathit{env}'_P)$$

[call$^{\text{rec}}_{\text{ns}}$]

$$\frac{\mathit{env}'_V,\ \mathit{env}'_P[p{\mapsto}(S,\ \mathit{env}'_V,\ \mathit{env}'_P)] \vdash \langle S,\ \mathit{sto}\rangle \to \mathit{sto}'}{\mathit{env}_V,\ \mathit{env}_P \vdash \langle \texttt{call}\ p,\ \mathit{sto}\rangle \to \mathit{sto}'}$$
$$\text{where } \mathit{env}_P\ p = (S,\ \mathit{env}'_V,\ \mathit{env}'_P)$$

Table 3.6 Natural semantics for **Proc** with static scope rules

new rule for blocks there is no analogue of $s''[\mathrm{DV}(D_V)\longmapsto s]$, as the values of variables can only be obtained by accessing the environment.

Exercise 3.12

Apply the natural semantics of Table 3.6 to the factorial statement of Exercise 3.8 and a store where the location for `x` has the value **3**. □

Exercise 3.13

Verify that the semantics applied to the statement of Exercise 3.9 gives the expected result. □

Exercise 3.14 (*)

An alternative semantics of the language **While** is defined by the axioms and rules $[\mathrm{ass}_{\mathrm{ns}}]$, $[\mathrm{skip}_{\mathrm{ns}}]$, $[\mathrm{comp}_{\mathrm{ns}}]$, $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]$, $[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}]$, $[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}]$, and $[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}]$ of Table 3.6. Formulate and prove the equivalence between this semantics and that of Table 2.1. □

Exercise 3.15

Modify the syntax of procedure declarations so that procedures take two *call-by-value* parameters:

$$\begin{aligned} D_P &::= \texttt{proc}\ p(x_1,x_2)\ \texttt{is}\ S;\ D_P \mid \varepsilon \\ S &::= \cdots \mid \texttt{call}\ p(a_1,a_2) \end{aligned}$$

Procedure environments will now be elements of

$$\mathbf{Env_P} = \mathbf{Pname} \hookrightarrow \mathbf{Var} \times \mathbf{Var} \times \mathbf{Stm} \times \mathbf{Env_V} \times \mathbf{Env_P}$$

Modify the semantics given above to handle this language. In particular, provide new rules for procedure calls: one for non-recursive procedures and another for recursive procedures. Construct statements that illustrate how the new rules are used. □

Exercise 3.16

Now consider the language **Proc** and the task of achieving *mutual recursion*. The procedure environment is now defined to be an element of

$$\mathbf{Env_P} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env_V} \times \mathbf{Env_P} \times \mathbf{Dec_P}$$

The idea is that if $env_P\ p = (S,\ env'_V,\ env'_P,\ D^\star_P)$, then $D^\star_P$ contains all the procedure declarations that are made in the same block as p. Define upd'_P by

$$\begin{aligned}\text{upd}'_P(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_V,\ env_P,\ D^\star_P) &= \\ &\text{upd}'_P(D_P,\ env_V,\ env_P[p{\mapsto}(S,\ env_V,\ env_P,\ D^\star_P)],\ D^\star_P) \\ \text{upd}'_P(\varepsilon,\ env_V,\ env_P, D^\star_P) &= env_P\end{aligned}$$

Next redefine upd_P by

$$\text{upd}_P(D_P,\ env_V,\ env_P) = \text{upd}'_P(D_P,\ env_V,\ env_P,\ D_P)$$

Modify the semantics of **Proc** so as to obtain mutual recursion among procedures defined in the same block. Illustrate how the new rules are used on an interesting statement of your choice.

(Hint: Convince yourself that $[\text{call}^{\text{rec}}_{\text{ns}}]$ is the only rule that needs to be changed. Then consider whether or not the function upd_P might be useful in the new definition of $[\text{call}^{\text{rec}}_{\text{ns}}]$.) □

Exercise 3.17

We shall consider a variant of the semantics where we use the variable environment rather than the store to hold the next free location. So assume that

$$\mathbf{Env}_\text{V} = \mathbf{Var} \cup \{\,\mathsf{next}\,\} \to \mathbf{Loc}$$

and

$$\mathbf{Store} = \mathbf{Loc} \to \mathbf{Z}$$

As before we shall write $sto \circ env_V$ for the state obtained by first using env_V to find the location of the variable and then sto to find the value of the location. The clauses of Table 3.5 are now replaced by

$$\frac{\langle D_V,\ env_V[x{\mapsto}l][\mathsf{next}{\mapsto}\text{new}\ l],\ sto[l{\mapsto}v]\rangle \to_D (env'_V,\ sto')}{\langle \texttt{var}\ x := a;\ D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')}$$

$$\text{where } v = \mathcal{A}[\![a]\!](sto{\circ}env_V) \text{ and } l = env_V\ \mathsf{next}$$

$$\langle \varepsilon,\ env_V,\ sto\rangle \to_D (env_V,\ sto)$$

Construct a statement that computes different results under the two variants of the semantics. Validate your claim by constructing derivation trees for the executions of the statement from a suitable state. Discuss whether or not this is a useful semantics. □

4 Provably Correct Implementation

A formal specification of the semantics of a programming language is useful when implementing it. In particular, it becomes possible to argue about the correctness of the implementation. We shall illustrate this by showing how to translate the language **While** into a structured form of assembler code for an abstract machine, and we shall then prove that the translation is correct. The idea is that we first define the *meaning* of the abstract machine instructions by an operational semantics. Then we define *translation functions* that will map expressions and statements in the **While** language into sequences of such instructions. The correctness result will then state that if we

– translate a program into code and

– execute the code on the abstract machine,

then we get the same result as was specified by the semantic functions $\mathcal{S}_{\text{ns}}$ and $\mathcal{S}_{\text{sos}}$ of Chapter 2.

4.1 The Abstract Machine

When specifying the abstract machine, it is convenient first to present its configurations and next its instructions and their meanings.

The abstract machine **AM** has configurations of the form $\langle c,\ e,\ s\rangle$, where

– c is the sequence of instructions (or code) to be executed,

– e is the evaluation stack, and

– s is the storage.

We use the *evaluation stack* to evaluate arithmetic and boolean expressions. Formally, it is a list of values, so writing

$$\mathbf{Stack} = (\mathbf{Z} \cup \mathbf{T})^\star$$

we have $e \in \mathbf{Stack}$. For the sake of simplicity, we shall assume that the *storage* is similar to the state — that is, $s \in \mathbf{State}$ — and it is used to hold the values of variables.

The *instructions* of **AM** are given by the abstract syntax

$$\begin{array}{lcl} \mathit{inst} & ::= & \text{PUSH-}n \mid \text{ADD} \mid \text{MULT} \mid \text{SUB} \\ & \mid & \text{TRUE} \mid \text{FALSE} \mid \text{EQ} \mid \text{LE} \mid \text{AND} \mid \text{NEG} \\ & \mid & \text{FETCH-}x \mid \text{STORE-}x \\ & \mid & \text{NOOP} \mid \text{BRANCH}(c,\, c) \mid \text{LOOP}(c,\, c) \\ c & ::= & \varepsilon \mid \mathit{inst}{:}c \end{array}$$

where ε is the empty sequence. We shall write **Code** for the syntactic category of *sequences of instructions*, so c is a meta-variable ranging over **Code**. Therefore we have

$$\langle c,\, e,\, s\rangle \in \mathbf{Code} \times \mathbf{Stack} \times \mathbf{State}$$

A configuration is a *terminal* (or final) configuration if its code component is the empty sequence; that is, if it has the form $\langle \varepsilon,\, e,\, s\rangle$.

The semantics of the instructions of the abstract machine is given by an *operational semantics*. As in the previous chapters, it will be specified by a transition system. The configurations have the form $\langle c,\, e,\, s\rangle$ as described above, and the transition relation $\triangleright$ specifies how to execute the instructions:

$$\langle c,\, e,\, s\rangle \triangleright \langle c',\, e',\, s'\rangle$$

The idea is that *one step of execution* will transform the configuration $\langle c,\, e,\, s\rangle$ into $\langle c',\, e',\, s'\rangle$. The relation is defined by the axioms of Table 4.1, where we use the notation ':' both for appending two instruction sequences and for prepending an element to a sequence. The evaluation stack is represented as a sequence of elements. It has the top of the stack to the left, and we shall write ε for the empty sequence.

In addition to the usual arithmetic and boolean operations, we have six instructions that modify the evaluation stack. The operation PUSH-n pushes a constant value n onto the stack, and TRUE and FALSE push the constants **tt** and **ff**, respectively, onto the stack. The operation FETCH-x pushes the value

$\langle \textsc{push}\text{-}n{:}c,\ e,\ s\rangle$	$\rhd$	$\langle c,\ \mathcal{N}[\![n]\!]{:}e,\ s\rangle$	
$\langle \textsc{add}{:}c,\ z_1{:}z_2{:}e,\ s\rangle$	$\rhd$	$\langle c,\ (z_1{+}z_2){:}e,\ s\rangle$	if $z_1,\ z_2{\in}\mathbf{Z}$
$\langle \textsc{mult}{:}c,\ z_1{:}z_2{:}e,\ s\rangle$	$\rhd$	$\langle c,\ (z_1{\star}z_2){:}e,\ s\rangle$	if $z_1,\ z_2{\in}\mathbf{Z}$
$\langle \textsc{sub}{:}c,\ z_1{:}z_2{:}e,\ s\rangle$	$\rhd$	$\langle c,\ (z_1{-}z_2){:}e,\ s\rangle$	if $z_1,\ z_2{\in}\mathbf{Z}$
$\langle \textsc{true}{:}c,\ e,\ s\rangle$	$\rhd$	$\langle c,\ \mathbf{tt}{:}e,\ s\rangle$	
$\langle \textsc{false}{:}c,\ e,\ s\rangle$	$\rhd$	$\langle c,\ \mathbf{ff}{:}e,\ s\rangle$	
$\langle \textsc{eq}{:}c,\ z_1{:}z_2{:}e,\ s\rangle$	$\rhd$	$\langle c,\ (z_1{=}z_2){:}e,\ s\rangle$	if $z_1,\ z_2{\in}\mathbf{Z}$
$\langle \textsc{le}{:}c,\ z_1{:}z_2{:}e,\ s\rangle$	$\rhd$	$\langle c,\ (z_1{\leq}z_2){:}e,\ s\rangle$	if $z_1,\ z_2{\in}\mathbf{Z}$
$\langle \textsc{and}{:}c,\ t_1{:}t_2{:}e,\ s\rangle$	$\rhd$	$\begin{cases} \langle c, \mathbf{tt} : e, s\rangle & \text{if } t_1{=}\mathbf{tt} \text{ and } t_2{=}\mathbf{tt} \\ \langle c, \mathbf{ff} : e, s\rangle & \text{if } t_1{=}\mathbf{ff} \text{ or } t_2{=}\mathbf{ff}\text{, and } t_1,\ t_2{\in}\mathbf{T} \end{cases}$	
$\langle \textsc{neg}{:}c,\ t{:}e,\ s\rangle$	$\rhd$	$\begin{cases} \langle c, \mathbf{ff} : e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{if } t{=}\mathbf{ff} \end{cases}$	
$\langle \textsc{fetch}\text{-}x{:}c,\ e,\ s\rangle$	$\rhd$	$\langle c,\ (s\ x){:}e,\ s\rangle$	
$\langle \textsc{store}\text{-}x{:}c,\ z{:}e,\ s\rangle$	$\rhd$	$\langle c,\ e,\ s[x{\mapsto}z]\rangle$	if $z{\in}\mathbf{Z}$
$\langle \textsc{noop}{:}c,\ e,\ s\rangle$	$\rhd$	$\langle c,\ e,\ s\rangle$	
$\langle \textsc{branch}(c_1,\ c_2){:}c,\ t{:}e,\ s\rangle$	$\rhd$	$\begin{cases} \langle c_1 : c, e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{if } t{=}\mathbf{ff} \end{cases}$	
$\langle \textsc{loop}(c_1,\ c_2){:}c,\ e,\ s\rangle$	$\rhd$	$\langle c_1{:}\textsc{branch}(c_2{:}\textsc{loop}(c_1,\ c_2),\ \textsc{noop}){:}c,\ e,\ s\rangle$	

Table 4.1 Operational semantics for **AM**

bound to x onto the stack, whereas STORE-x pops the topmost element off the stack and updates the storage so that the popped value is bound to x. The instruction BRANCH(c_1, c_2) will also change the flow of control. If the top of the stack is the value **tt** (that is, some boolean expression has been evaluated to true), then the stack is popped and c_1 is to be executed next. Otherwise, if the top element of the stack is **ff**, then it will be popped and c_2 will be executed next.

The abstract machine has two instructions that change the flow of control. The instruction BRANCH(c_1, c_2) will be used to implement the conditional: as described above, it will choose the code component c_1 or c_2 depending on the current value on top of the stack. If the top of the stack is not a truth value, the machine will halt, as there is no next configuration (since the meaning of BRANCH(···,···) is not defined in that case). A looping construct such as the `while`-construct of **While** can be implemented using the instruction LOOP(c_1, c_2). The semantics of this instruction is defined by rewriting it to a combination of other constructs including the BRANCH-instruction and itself. We shall see shortly how this can be used.

The operational semantics of Table 4.1 is indeed a structural operational semantics for **AM**. Corresponding to the derivation sequences of Chapter 2, we shall define a *computation sequence* for **AM**. Given a sequence c of instructions and a storage s, a computation sequence for c and s is either

1. a *finite* sequence

   $$\gamma_0, \gamma_1, \gamma_2, \cdots, \gamma_{\mathrm{k}}$$

   sometimes written

   $$\gamma_0 \rhd \gamma_1 \rhd \gamma_2 \rhd \cdots \rhd \gamma_{\mathrm{k}}$$

   consisting of configurations satisfying $\gamma_0 = \langle c,\ \varepsilon,\ s\rangle$ and $\gamma_{\mathrm{i}} \rhd \gamma_{\mathrm{i+1}}$ for $0{\le}\mathrm{i}{<}\mathrm{k}$, $\mathrm{k}{\ge}0$, and where there is no γ such that $\gamma_{\mathrm{k}} \rhd \gamma$, or it is

2. an *infinite* sequence

   $$\gamma_0, \gamma_1, \gamma_2, \cdots$$

   sometimes written

   $$\gamma_0 \rhd \gamma_1 \rhd \gamma_2 \rhd \cdots$$

   consisting of configurations satisfying $\gamma_0 = \langle c,\ \varepsilon,\ s\rangle$ and $\gamma_{\mathrm{i}} \rhd \gamma_{\mathrm{i+1}}$ for $0{\le}\mathrm{i}$.

Note that initial configurations always have an *empty* evaluation stack. A computation sequence is

– *terminating* if and only if it is finite and

– *looping* if and only if it is infinite.

A terminating computation sequence may end in a terminal configuration (that is, a configuration with an empty code component) or in a stuck configuration (for example, $\langle$ADD, ε, $s\rangle$).

Example 4.1

Consider the instruction sequence

$$\textsc{push}\text{-}1{:}\textsc{fetch}\text{-}\mathtt{x}{:}\textsc{add}{:}\textsc{store}\text{-}\mathtt{x}$$

From the initial storage s with $s\ \mathtt{x} = \mathbf{3}$ we get the terminating computation

$$\begin{aligned}
&\langle \textsc{push}\text{-}1{:}\textsc{fetch}\text{-}\mathtt{x}{:}\textsc{add}{:}\textsc{store}\text{-}\mathtt{x},\ \varepsilon,\ s\rangle\\
\rhd\ &\langle \textsc{fetch}\text{-}\mathtt{x}{:}\textsc{add}{:}\textsc{store}\text{-}\mathtt{x},\ \mathbf{1},\ s\rangle\\
\rhd\ &\langle \textsc{add}{:}\textsc{store}\text{-}\mathtt{x},\ \mathbf{3}{:}\mathbf{1},\ s\rangle\\
\rhd\ &\langle \textsc{store}\text{-}\mathtt{x},\ \mathbf{4},\ s\rangle\\
\rhd\ &\langle \varepsilon,\ \varepsilon,\ s[\mathtt{x}\mapsto\mathbf{4}]\rangle
\end{aligned}$$

At this point the computation stops because there is no next step. □

Example 4.2

Consider the code

$$\textsc{loop}(\textsc{true},\ \textsc{noop})$$

We have the following looping computation sequence

$$\begin{aligned}
&\langle \textsc{loop}(\textsc{true},\ \textsc{noop}),\ \varepsilon,\ s\rangle\\
\rhd\ &\langle \textsc{true}{:}\textsc{branch}(\textsc{noop}{:}\textsc{loop}(\textsc{true},\ \textsc{noop}),\ \textsc{noop}),\ \varepsilon,\ s\rangle\\
\rhd\ &\langle \textsc{branch}(\textsc{noop}{:}\textsc{loop}(\textsc{true},\ \textsc{noop}),\ \textsc{noop}),\ \mathbf{tt},\ s\rangle\\
\rhd\ &\langle \textsc{noop}{:}\textsc{loop}(\textsc{true},\ \textsc{noop}),\ \varepsilon,\ s\rangle\\
\rhd\ &\langle \textsc{loop}(\textsc{true},\ \textsc{noop}),\ \varepsilon,\ s\rangle\\
\rhd\ &\cdots
\end{aligned}$$

where the unfolding of the LOOP-instruction is repeated. □

Exercise 4.3

Consider the code

$$\begin{aligned}
&\textsc{push}\text{-}0{:}\textsc{store}\text{-}\mathtt{z}{:}\textsc{fetch}\text{-}\mathtt{x}{:}\textsc{store}\text{-}\mathtt{r}{:}\\
&\textsc{loop}(\ \ \textsc{fetch}\text{-}\mathtt{r}{:}\textsc{fetch}\text{-}\mathtt{y}{:}\textsc{le},\\
&\qquad\quad \textsc{fetch}\text{-}\mathtt{y}{:}\textsc{fetch}\text{-}\mathtt{r}{:}\textsc{sub}{:}\textsc{store}\text{-}\mathtt{r}{:}\\
&\qquad\quad \textsc{push}\text{-}1{:}\textsc{fetch}\text{-}\mathtt{z}{:}\textsc{add}{:}\textsc{store}\text{-}\mathtt{z})
\end{aligned}$$

Determine the function computed by this code. □

Properties of AM

The semantics we have specified for the abstract machine is concerned with the execution of individual instructions and is therefore close in spirit to the structural operational semantics studied in Chapter 2. When proving the correctness of the code generation, we shall need a few results analogous to those holding for the structural operational semantics. As their proofs follow the same lines as those for the structural operational semantics, we shall leave them as exercises and only *reformulate* the proof technique from Section 2.2:

Induction on the Length of Computation Sequences	
1:	Prove that the property holds for all computation sequences of length 0.
2:	Prove that the property holds for all other computation sequences: Assume that the property holds for all computation sequences of length at most k (this is called the *induction hypothesis*) and show that it holds for computation sequences of length k+1.

The induction step of a proof following this technique will often be done by a case analysis on the first instruction of the code component of the configuration.

Exercise 4.4 (Essential)

By analogy with Exercise 2.21, prove that

if $\langle c_1,\ e_1,\ s\rangle \rhd^{\mathrm{k}} \langle c',\ e',\ s'\rangle$ then $\langle c_1{:}c_2,\ e_1{:}e_2,\ s\rangle \rhd^{\mathrm{k}} \langle c'{:}c_2,\ e'{:}e_2,\ s'\rangle$

This means that we can extend the code component as well as the stack component without changing the behaviour of the machine. □

Exercise 4.5 (Essential)

By analogy with Lemma 2.19, prove that if

$$\langle c_1{:}c_2,\ e,\ s\rangle \rhd^{\mathrm{k}} \langle \varepsilon,\ e'',\ s''\rangle$$

then there exists a configuration $\langle \varepsilon,\ e',\ s'\rangle$ and natural numbers k_1 and k_2 with $\mathrm{k}_1+\mathrm{k}_2=\mathrm{k}$ such that

$$\langle c_1,\ e,\ s\rangle \rhd^{\mathrm{k}_1} \langle \varepsilon,\ e',\ s'\rangle \text{ and } \langle c_2,\ e',\ s'\rangle \rhd^{\mathrm{k}_2} \langle \varepsilon,\ e'',\ s''\rangle$$

This means that the execution of a composite sequence of instructions can be split into two pieces. □

The notion of determinism is defined as for the structural operational semantics. So the semantics of an abstract machine is *deterministic* if for all choices of γ, γ', and γ'' we have that

$$\gamma \rhd \gamma' \text{ and } \gamma \rhd \gamma'' \text{ imply } \gamma' = \gamma''$$

Exercise 4.6 (Essential)

Show that the machine semantics of Table 4.1 is deterministic. Use this to deduce that there is exactly one computation sequence starting in a configuration $\langle c,\ e,\ s\rangle$. □

The Execution Function $\mathcal{M}$

We shall define the *meaning* of a sequence of instructions as a (partial) function from **State** to **State**:

$$\mathcal{M}\colon \textbf{Code} \to (\textbf{State} \hookrightarrow \textbf{State})$$

It is given by

$$\mathcal{M}[\![c]\!]\ s = \begin{cases} s' & \text{if } \langle c,\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ e,\ s'\rangle \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

The function is well-defined because of Exercise 4.6. Note that the definition does not require the stack component of the terminal configuration to be empty but it does require the code component to be so.

The abstract machine **AM** may seem far removed from more traditional machine architectures. In the next few exercises we shall gradually bridge this gap.

Exercise 4.7

AM refers to variables by their *name* rather than by their *address*. The abstract machine $\mathbf{AM}_1$ differs from **AM** in that

- the configurations have the form $\langle c,\ e,\ m\rangle$, where c and e are as in **AM** and m, the *memory*, is a (finite) list of values; that is, $m \in \mathbf{Z}^\star$, and
- the instructions FETCH-x and STORE-x are replaced by instructions GET-n and PUT-n, where n is a natural number (an address).

Specify the operational semantics of the machine. You may write $m[n]$ to select the nth value in the list m (when n is positive but less than or equal to the length of m). What happens if we reference an address that is outside the memory — does this correspond to your expectations? $\square$

Exercise 4.8

The next step is to get rid of the operations BRANCH($\cdots$,$\cdots$) and LOOP($\cdots$,$\cdots$). The idea is to introduce instructions for *defining labels* and for *jumping to labels*. The abstract machine $\mathbf{AM}_2$ differs from $\mathbf{AM}_1$ (of Exercise 4.7) in that

– the configurations have the form $\langle pc, c, e, m\rangle$, where c, e, and m are as before and pc is the program counter (a natural number) pointing to an instruction in c, and

– the instructions BRANCH($\cdots$,$\cdots$) and LOOP($\cdots$,$\cdots$) are replaced by the instructions LABEL-l, JUMP-l, and JUMPFALSE-l, where l is a label (a natural number).

The idea is that we will execute the instruction in c that pc points to and in most cases this will cause the program counter to be incremented by 1. The instruction LABEL-l has no effect except updating the program counter. The instruction JUMP-l will move the program counter to the unique instruction LABEL-l (if it exists). The instruction JUMPFALSE-l will only move the program counter to the instruction LABEL-l if the value on top of the stack is **ff**; if it is **tt**, the program counter will be incremented by 1.

Specify an operational semantics for $\mathbf{AM}_2$. You may write $c[pc]$ for the instruction in c pointed to by pc (if pc is positive and less than or equal to the length of c). What happens if the same label is defined more than once — does this correspond to your expectations? $\square$

Exercise 4.9

Finally, we shall consider an abstract machine $\mathbf{AM}_3$ where the labels of the instructions JUMP-l and JUMPFALSE-l of Exercise 4.8 are *absolute addresses*; so JUMP-7 means jump to the 7th instruction of the code (rather than to the instruction LABEL-7). Specify the operational semantics of the machine. What happens if we jump to an instruction that is not in the code? $\square$

$$
\begin{aligned}
\mathcal{CA}[\![n]\!] &= \textsc{push}\text{-}n \\
\mathcal{CA}[\![x]\!] &= \textsc{fetch}\text{-}x \\
\mathcal{CA}[\![a_1{+}a_2]\!] &= \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{add} \\
\mathcal{CA}[\![a_1 \star a_2]\!] &= \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{mult} \\
\mathcal{CA}[\![a_1{-}a_2]\!] &= \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{sub} \\
\mathcal{CB}[\![\texttt{true}]\!] &= \textsc{true} \\
\mathcal{CB}[\![\texttt{false}]\!] &= \textsc{false} \\
\mathcal{CB}[\![a_1 = a_2]\!] &= \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{eq} \\
\mathcal{CB}[\![a_1{\leq}a_2]\!] &= \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{le} \\
\mathcal{CB}[\![\neg b]\!] &= \mathcal{CB}[\![b]\!]{:}\textsc{neg} \\
\mathcal{CB}[\![b_1{\wedge}b_2]\!] &= \mathcal{CB}[\![b_2]\!]{:}\mathcal{CB}[\![b_1]\!]{:}\textsc{and}
\end{aligned}
$$

Table 4.2 Translation of expressions

4.2 Specification of the Translation

We shall now study how to generate code for the abstract machine.

Expressions

Arithmetic and boolean expressions will be evaluated on the evaluation stack of the machine, and the code to be generated must effect this. This is accomplished by the (total) functions

$$\mathcal{CA}\colon \mathbf{Aexp} \rightarrow \mathbf{Code}$$

and

$$\mathcal{CB}\colon \mathbf{Bexp} \rightarrow \mathbf{Code}$$

specified in Table 4.2. Note that the code generated for binary expressions consists of the code for the *right* argument followed by that for the *left* argument and finally the appropriate instruction for the operator. In this way it is ensured that the arguments appear on the evaluation stack in the order required by the instructions (in Table 4.1). Note that $\mathcal{CA}$ and $\mathcal{CB}$ are defined compositionally.

$$\begin{array}{rcl}
\mathcal{CS}[\![x := a]\!] & = & \mathcal{CA}[\![a]\!]\text{:}\textsc{store}\text{-}x \\
\mathcal{CS}[\![\texttt{skip}]\!] & = & \textsc{noop} \\
\mathcal{CS}[\![S_1;S_2]\!] & = & \mathcal{CS}[\![S_1]\!]\text{:}\mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \mathcal{CB}[\![b]\!]\text{:}\textsc{branch}(\mathcal{CS}[\![S_1]\!],\mathcal{CS}[\![S_2]\!]) \\
\mathcal{CS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] & = & \textsc{loop}(\mathcal{CB}[\![b]\!],\mathcal{CS}[\![S]\!])
\end{array}$$

Table 4.3 Translation of statements in **While**

Example 4.10

For the arithmetic expression x+1, we calculate the code as follows:

$$\mathcal{CA}[\![\texttt{x+1}]\!] = \mathcal{CA}[\![\texttt{1}]\!]\text{:}\mathcal{CA}[\![\texttt{x}]\!]\text{:}\textsc{add} = \textsc{push}\text{-}\texttt{1}\text{:}\textsc{fetch}\text{-}x\text{:}\textsc{add}$$

Exercise 4.11

It is clear that $\mathcal{A}[\![(a_1+a_2)+a_3]\!]$ equals $\mathcal{A}[\![a_1+(a_2+a_3)]\!]$. Show that it is *not* the case that $\mathcal{CA}[\![(a_1+a_2)+a_3]\!]$ equals $\mathcal{CA}[\![a_1+(a_2+a_3)]\!]$. Nonetheless, show that $\mathcal{CA}[\![(a_1+a_2)+a_3]\!]$ and $\mathcal{CA}[\![a_1+(a_2+a_3)]\!]$ do in fact *behave* similarly. □

Statements

The translation of statements into abstract machine code is given by the function

$$\mathcal{CS}\text{: } \mathbf{Stm} \to \mathbf{Code}$$

specified in Table 4.3. The code generated for an arithmetic expression a ensures that the value of the expression is on top of the evaluation stack when it has been computed. So in the code for $x := a$ it suffices to append the code for a with the instruction STORE-x. This instruction assigns to x the appropriate value and additionally pops the stack. For the skip-statement, we generate the NOOP-instruction. For sequencing of statements, we just concatenate the two instruction sequences. When generating code for the conditional, the code for the boolean expression will ensure that a truth value will be placed on top of the evaluation stack and the BRANCH-instruction will then inspect (and pop) that value and select the appropriate piece of code. Finally, the code for the while-construct uses the LOOP-instruction. Again we may note that $\mathcal{CS}$ is defined in a compositional manner — and that it never generates an empty sequence of instructions.

Example 4.12

The code generated for the factorial statement considered earlier is as follows:

$$
\begin{aligned}
&\mathcal{CS}\,[\![\texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1})]\!]\\
&\quad=\mathcal{CS}[\![\texttt{y:=1}]\!]:\mathcal{CS}[\![\texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1})]\!]\\
&\quad=\mathcal{CA}[\![1]\!]:\textsc{store}\text{-}\texttt{y}:\textsc{loop}(\mathcal{CB}[\![\neg(\texttt{x=1})]\!],\mathcal{CS}[\![\texttt{y:=y}\star\texttt{x; x:=x}-\texttt{1}]\!])\\
&\quad=\textsc{push}\text{-}\texttt{1}:\textsc{store}\text{-}\texttt{y}:\textsc{loop}(\mathcal{CB}[\![\texttt{x=1}]\!]:\textsc{neg},\mathcal{CS}[\![\texttt{y:=y}\star\texttt{x}]\!]:\mathcal{CS}[\![\texttt{x:=x}-\texttt{1}]\!])\\
&\quad\ \vdots\\
&\quad=\textsc{push}\text{-}\texttt{1}:\textsc{store}\text{-}\texttt{y}:\textsc{loop}(\ \textsc{push}\text{-}\texttt{1}:\textsc{fetch}\text{-}\texttt{x}:\textsc{eq}:\textsc{neg},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \textsc{fetch}\text{-}\texttt{x}:\textsc{fetch}\text{-}\texttt{y}:\textsc{mult}:\textsc{store}\text{-}\texttt{y}:\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textsc{push}\text{-}\texttt{1}:\textsc{fetch}\text{-}\texttt{x}:\textsc{sub}:\textsc{store}\text{-}\texttt{x})
\end{aligned}
$$

Exercise 4.13

Use $\mathcal{CS}$ to generate code for the statement

$$\texttt{z:=0; while y}\leq\texttt{x do (z:=z+1; x:=x}-\texttt{y)}$$

Trace the computation of the code starting from a storage where `x` is **17** and `y` is **5**. □

Exercise 4.14

Extend **While** with the construct `repeat` S `until` b and specify how to generate code for it. Note that the definition has to be compositional and that it is *not* necessary to extend the instruction set of the abstract machine. □

Exercise 4.15

Extend **While** with the construct `for` $x := a_1$ `to` a_2 `do` S and specify how to generate code for it. As in Exercise 4.14, the definition has to be compositional, but you may have to introduce an instruction COPY that duplicates the element on top of the evaluation stack. □

The Semantic Function $\mathcal{S}_{\mathrm{am}}$

The meaning of a statement S can now be obtained by first translating it into code for **AM** and next executing the code on the abstract machine. The effect of this is expressed by the function

$$\mathcal{S}_{\mathrm{am}}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

defined by

$$\mathcal{S}_{\mathrm{am}}[\![S]\!] = (\mathcal{M} \circ \mathcal{CS})[\![S]\!]$$

Exercise 4.16

Modify the code generation so as to translate **While** into code for the abstract machine $\mathbf{AM}_1$ of Exercise 4.7. You may assume the existence of a function

$$env\colon \mathbf{Var} \to \mathbf{N}$$

that maps variables to their addresses. Apply the code generation function to the factorial statement also considered in Example 4.12 and execute the code so obtained starting from a memory where `x` is **3**. ☐

Exercise 4.17

Modify the code generation so as to translate **While** into code for the abstract machine $\mathbf{AM}_2$ of Exercise 4.8. Be careful to generate unique labels, for example by having "the next unused label" as an additional parameter to the code generation functions. Apply the code generation function to the factorial statement and execute the code so obtained starting from a memory where `x` has the value **3**. ☐

4.3 Correctness

The correctness of the implementation amounts to showing that, if we first translate a statement into code for **AM** and then execute that code, then we must obtain the same result as specified by the operational semantics of **While**.

Expressions

The correctness of the implementation of arithmetic expressions is expressed by the following lemma.

Lemma 4.18

For all arithmetic expressions a, we have

$$\langle \mathcal{CA}[\![a]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \mathcal{A}[\![a]\!]s,\ s\rangle$$

Furthermore, all intermediate configurations of this computation sequence will have a non-empty evaluation stack.

Proof: The proof is by structural induction on a. Below we shall give the proof for three illustrative cases, leaving the remaining ones to the reader.

The case n: We have $\mathcal{CA}[\![n]\!] = \textsc{push}\text{-}n$, and from Table 4.1 we get

$$\langle \textsc{push}\text{-}n,\ \varepsilon,\ s\rangle \rhd \langle \varepsilon,\ \mathcal{N}[\![n]\!],\ s\rangle$$

Since $\mathcal{A}[\![n]\!]s = \mathcal{N}[\![n]\!]$ (see Table 1.1) we have completed the proof in this case.

The case x: We have $\mathcal{CA}[\![x]\!] = \textsc{fetch}\text{-}x$, and from Table 4.1 we get

$$\langle \textsc{fetch}\text{-}x,\ \varepsilon,\ s\rangle \rhd \langle \varepsilon,\ (s\ x),\ s\rangle$$

Since $\mathcal{A}[\![x]\!]s = s\ x$, this is the required result.

The case $a_1{+}a_2$: We have $\mathcal{CA}[\![a_1{+}a_2]\!] = \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{add}$. The induction hypothesis applied to a_1 and a_2 gives that

$$\langle \mathcal{CA}[\![a_1]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \mathcal{A}[\![a_1]\!]s,\ s\rangle$$

and

$$\langle \mathcal{CA}[\![a_2]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \mathcal{A}[\![a_2]\!]s,\ s\rangle$$

In both cases, all intermediate configurations will have a non-empty evaluation stack. Using Exercise 4.4, we get that

$$\langle \mathcal{CA}[\![a_2]\!]{:}\mathcal{CA}[\![a_1]\!]{:}\textsc{add},\ \varepsilon,\ s\rangle \rhd^* \langle \mathcal{CA}[\![a_1]\!]{:}\textsc{add},\ \mathcal{A}[\![a_2]\!]s,\ s\rangle$$

Applying the exercise once more, we get that

$$\langle \mathcal{CA}[\![a_1]\!]{:}\textsc{add},\ \mathcal{A}[\![a_2]\!]s,\ s\rangle \rhd^* \langle \textsc{add},\ (\mathcal{A}[\![a_1]\!]s){:}(\mathcal{A}[\![a_2]\!]s),\ s\rangle$$

Using the transition relation for $\textsc{add}$ given in Table 4.1, we get

$$\langle \textsc{add},\ (\mathcal{A}[\![a_1]\!]s){:}(\mathcal{A}[\![a_2]\!]s),\ s\rangle \rhd \langle \varepsilon,\ \mathcal{A}[\![a_1]\!]s{+}\mathcal{A}[\![a_2]\!]s,\ s\rangle$$

It is easy to check that all intermediate configurations have a non-empty evaluation stack. Since $\mathcal{A}[\![a_1{+}a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$, we thus have the desired result. □

We have a similar result for boolean expressions.

Exercise 4.19 (Essential)

Show that, for all boolean expressions b, we have

$$\langle \mathcal{CB}[\![b]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \mathcal{B}[\![b]\!]s,\ s\rangle$$

Furthermore, show that all intermediate configurations of this computation sequence will have a non-empty evaluation stack. □

Statements

When formulating the correctness of the result for statements, we have a choice between using

– the natural semantics or

– the structural operational semantics.

Here we shall use the natural semantics, but in the next section we sketch the proof in the case where the structural operational semantics is used.

The correctness of the translation of statements is expressed by the following theorem.

Theorem 4.20

For every statement S of **While**, we have $\mathcal{S}_{\mathrm{ns}}[\![S]\!] = \mathcal{S}_{\mathrm{am}}[\![S]\!]$.

This theorem relates the behaviour of a statement under the natural semantics with the behaviour of the code on the abstract machine under its operational semantics. In analogy with Theorem 2.26, it expresses two properties:

– If the execution of S from some state terminates in one of the semantics, then it also terminates in the other and the resulting states will be equal.

– Furthermore, if the execution of S from some state loops in one of the semantics, then it will also loop in the other.

The theorem is proved in two parts, as expressed by Lemmas 4.21 and 4.22 below. We shall first prove Lemma 4.21.

Lemma 4.21

For every statement S of **While** and states s and s', we have that

$$\text{if } \langle S,\ s\rangle \to s' \ \text{ then } \langle \mathcal{CS}[\![S]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \varepsilon,\ s'\rangle$$

So if the execution of S from s terminates in the natural semantics, then the execution of the code for S from storage s will terminate and the resulting states and storages will be equal.

Proof: We proceed by induction on the shape of the derivation tree for $\langle S,\ s\rangle \to s'$.

The case [ass$_{\text{ns}}$]: We assume that

$$\langle x{:=}a,\ s\rangle \to s'$$

where $s'{=}s[x{\mapsto}\mathcal{A}[\![a]\!]s]$. From Table 4.3, we have

$$\mathcal{CS}[\![x{:=}a]\!] = \mathcal{CA}[\![a]\!]{:}\text{STORE-}x$$

From Lemma 4.18 applied to a, we get

$$\langle \mathcal{CA}[\![a]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \mathcal{A}[\![a]\!]s,\ s\rangle$$

and then Exercise 4.4 gives the first part of

$$\begin{array}{lcl} \langle \mathcal{CA}[\![a]\!]{:}\text{STORE-}x,\ \varepsilon,\ s\rangle & \rhd^* & \langle \text{STORE-}x,\ (\mathcal{A}[\![a]\!]s),\ s\rangle \\ & \rhd & \langle \varepsilon,\ \varepsilon,\ s[x{\mapsto}\mathcal{A}[\![a]\!]s]\rangle \end{array}$$

and the second part follows from the operational semantics for STORE-x given in Table 4.1. Since $s' = s[x{\mapsto}\mathcal{A}[\![a]\!]s]$, this completes the proof.

The case [skip$_{\text{ns}}$]: Straightforward.

The case [comp$_{\text{ns}}$]: Assume that

$$\langle S_1;S_2,\ s\rangle \to s''$$

holds because

$$\langle S_1,\ s\rangle \to s' \text{ and } \langle S_2,\ s'\rangle \to s''$$

From Table 4.3, we have

$$\mathcal{CS}[\![S_1;S_2]\!] = \mathcal{CS}[\![S_1]\!]{:}\mathcal{CS}[\![S_2]\!]$$

We shall now apply the induction hypothesis to the premises $\langle S_1,\ s\rangle \to s'$ and $\langle S_2,\ s'\rangle \to s''$, and we get

$$\langle \mathcal{CS}[\![S_1]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \varepsilon,\ \varepsilon,\ s'\rangle$$

and

$$\langle \mathcal{CS}[\![S_2]\!],\ \varepsilon,\ s'\rangle \rhd^* \langle \varepsilon,\ \varepsilon,\ s''\rangle$$

Using Exercise 4.4, we then have

$$\langle \mathcal{CS}[\![S_1]\!]{:}\mathcal{CS}[\![S_2]\!],\ \varepsilon,\ s\rangle \rhd^* \langle \mathcal{CS}[\![S_2]\!],\ \varepsilon,\ s'\rangle \rhd^* \langle \varepsilon,\ \varepsilon,\ s''\rangle$$

and the result follows.

The case [$\text{if}^{\text{tt}}_{\text{ns}}$]: Assume that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'$$

because $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and

$$\langle S_1,\ s\rangle \to s'$$

From Table 4.3, we get that

$$\mathcal{CS}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \mathcal{CB}[\![b]\!]{:}\text{BRANCH}(\mathcal{CS}[\![S_1]\!],\ \mathcal{CS}[\![S_2]\!])$$

Using Exercises 4.19 and 4.4 we get the first part of

$$\begin{aligned}
&\langle \mathcal{CB}[\![b]\!]{:}\text{BRANCH}(\mathcal{CS}[\![S_1]\!],\ \mathcal{CS}[\![S_2]\!]),\ \varepsilon,\ s\rangle\\
&\rhd^* \langle \text{BRANCH}(\mathcal{CS}[\![S_1]\!],\ \mathcal{CS}[\![S_2]\!]),\ (\mathcal{B}[\![b]\!]s),\ s\rangle\\
&\rhd \langle \mathcal{CS}[\![S_1]\!],\ \varepsilon,\ s\rangle\\
&\rhd^* \langle \varepsilon,\ \varepsilon,\ s'\rangle
\end{aligned}$$

The second part follows from the definition of the meaning of the instruction BRANCH in the case where the element on top of the evaluation stack is **tt** (which is the value of $\mathcal{B}[\![b]\!]s$). The third part of the computation sequence comes from applying the induction hypothesis to the premise $\langle S_1,\ s\rangle \to s'$.

The case [$\text{if}^{\text{ff}}_{\text{ns}}$]: Analogous.

The case [$\text{while}^{\text{tt}}_{\text{ns}}$]: Assume that

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$$

because $\mathcal{B}[\![b]\!]s = \mathbf{tt}$,

$$\langle S,\ s\rangle \to s' \text{ and } \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$$

From Table 4.3, we have

$$\mathcal{CS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \text{LOOP}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!])$$

and get

$$\begin{aligned}
&\langle \textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \varepsilon, s\rangle\\
&\rhd\ \langle \mathcal{CB}[\![b]\!]{:}\textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \textsc{noop}), \varepsilon, s\rangle\\
&\rhd^*\ \langle \textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \textsc{noop}), (\mathcal{B}[\![b]\!]s), s\rangle\\
&\rhd\ \langle \mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \varepsilon, s\rangle
\end{aligned}$$

Here the first part follows from the meaning of the LOOP-instruction (see Table 4.1) and the second part from Exercises 4.19 and 4.4. Since $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ the third part follows from the meaning of the BRANCH-instruction. The induction hypothesis can now be applied to the premises $\langle S, s\rangle \to s'$ and $\langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to s''$ and gives

$$\langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \varepsilon, s'\rangle$$

and

$$\langle \textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \varepsilon, s'\rangle \rhd^* \langle \varepsilon, \varepsilon, s''\rangle$$

so using Exercise 4.4 we get

$$\begin{aligned}
&\langle \mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \varepsilon, s\rangle\\
&\rhd^*\ \langle \textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \varepsilon, s'\rangle\\
&\rhd^*\ \langle \varepsilon, \varepsilon, s''\rangle
\end{aligned}$$

The case [while$^{\text{ff}}_{\text{ns}}$]: Assume that $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s'$ holds because $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and then $s = s'$. We have

$$\begin{aligned}
&\langle \textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \varepsilon, s\rangle\\
&\rhd\ \langle \mathcal{CB}[\![b]\!]{:}\textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \textsc{noop}), \varepsilon, s\rangle\\
&\rhd^*\ \langle \textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!]), \textsc{noop}), (\mathcal{B}[\![b]\!]s), s\rangle\\
&\rhd\ \langle \textsc{noop}, \varepsilon, s\rangle\\
&\rhd\ \langle \varepsilon, \varepsilon, s\rangle
\end{aligned}$$

using the definitions of the LOOP-, BRANCH-, and NOOP-instructions in Table 4.1 together with Exercises 4.19 and 4.4. □

This proves Lemma 4.21. The second part of the theorem follows from Lemma 4.22:

Lemma 4.22

For every statement S of **While** and states s and s', we have that

$$\text{if } \langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^{\text{k}} \langle \varepsilon, e, s'\rangle \text{ then } \langle S, s\rangle \to s' \text{ and } e = \varepsilon$$

So if the execution of the code for S from a storage s terminates, then the natural semantics of S from s will terminate in a state being equal to the storage of the terminal configuration.

Proof: We shall proceed by induction on the length k of the computation sequence of the abstract machine. If $\mathrm{k} = 0$, the result holds vacuously because $\mathcal{CS}[\![S]\!] = \varepsilon$ cannot occur. So assume that it holds for $\mathrm{k} \leq \mathrm{k}_0$ and we shall prove that it holds for $\mathrm{k} = \mathrm{k}_0{+}1$. We proceed by cases on the statement S.

The case $x{:=}a$: We then have $\mathcal{CS}[\![x := a]\!] = \mathcal{CA}[\![a]\!]{:}\textsc{store}\text{-}x$ so assume that

$$\langle \mathcal{CA}[\![a]\!]{:}\textsc{store}\text{-}x,\ \varepsilon,\ s\rangle \rhd^{\mathrm{k}_0+1} \langle \varepsilon,\ e,\ s'\rangle$$

Then, by Exercise 4.5, there must be a configuration of the form $\langle \varepsilon,\ e'',\ s''\rangle$ such that

$$\langle \mathcal{CA}[\![a]\!],\ \varepsilon,\ s\rangle \rhd^{\mathrm{k}_1} \langle \varepsilon,\ e'',\ s''\rangle$$

and

$$\langle \textsc{store}\text{-}x,\ e'',\ s''\rangle \rhd^{\mathrm{k}_2} \langle \varepsilon,\ e,\ s'\rangle$$

where $\mathrm{k}_1 + \mathrm{k}_2 = \mathrm{k}_0 + 1$. From Lemma 4.18 and Exercise 4.6, we get that e'' must be $(\mathcal{A}[\![a]\!]s)$ and s'' must be s. Using the semantics of $\textsc{store}\text{-}x$ we therefore see that s' is $s[x{\mapsto}\mathcal{A}[\![a]\!]s]$ and e is ε. It now follows from [ass$_{\mathrm{ns}}$] that $\langle x{:=}a,\ s\rangle{\to}s'$.

The case `skip`: Straightforward.

The case $S_1;S_2$: Assume that

$$\langle \mathcal{CS}[\![S_1]\!]{:}\mathcal{CS}[\![S_2]\!],\ \varepsilon,\ s\rangle \rhd^{\mathrm{k}_0+1} \langle \varepsilon,\ e,\ s''\rangle$$

Then, by Exercise 4.5, there must be a configuration of the form $\langle \varepsilon,\ e',\ s'\rangle$ such that

$$\langle \mathcal{CS}[\![S_1]\!],\ \varepsilon,\ s\rangle \rhd^{\mathrm{k}_1} \langle \varepsilon,\ e',\ s'\rangle$$

and

$$\langle \mathcal{CS}[\![S_2]\!],\ e',\ s'\rangle \rhd^{\mathrm{k}_2} \langle \varepsilon,\ e,\ s''\rangle$$

where $\mathrm{k}_1 + \mathrm{k}_2 = \mathrm{k}_0 + 1$. Since $\mathcal{CS}[\![S_2]\!]$ is non-empty, we have $\mathrm{k}_2 > 0$ and hence $\mathrm{k}_1 \leq \mathrm{k}_0$. The induction hypothesis can now be applied to the first of these computation sequences and gives

$$\langle S_1,\ s\rangle \to s' \text{ and } e' = \varepsilon$$

Thus we have $\langle \mathcal{CS}[\![S_2]\!],\ \varepsilon,\ s'\rangle \rhd^{\mathrm{k}_2} \langle \varepsilon,\ e,\ s''\rangle$ and since $\mathrm{k}_2 \leq \mathrm{k}_0$ the induction hypothesis can be applied to this computation sequence and gives

$$\langle S_2,\ s'\rangle \to s'' \text{ and } e = \varepsilon$$

The rule [comp$_{\text{ns}}$] now gives $\langle S_1;S_2,\ s\rangle \to s''$ as required.

The case `if` b `then` S_1 `else` S_2: The code generated for the conditional is

$$\mathcal{CB}[\![b]\!]{:}\text{BRANCH}(\mathcal{CS}[\![S_1]\!],\ \mathcal{CS}[\![S_2]\!])$$

so we assume that

$$\langle \mathcal{CB}[\![b]\!]{:}\text{BRANCH}(\mathcal{CS}[\![S_1]\!],\ \mathcal{CS}[\![S_2]\!]),\ \varepsilon,\ s\rangle \rhd^{\mathrm{k_0}+1} \langle \varepsilon,\ e,\ s'\rangle$$

Then, by Exercise 4.5, there must be a configuration of the form $\langle \varepsilon,\ e'',\ s''\rangle$ such that

$$\langle \mathcal{CB}[\![b]\!],\ \varepsilon,\ s\rangle \rhd^{\mathrm{k_1}} \langle \varepsilon,\ e'',\ s''\rangle$$

and

$$\langle \text{BRANCH}(\mathcal{CS}[\![S_1]\!],\ \mathcal{CS}[\![S_2]\!]),\ e'',\ s''\rangle \rhd^{\mathrm{k_2}} \langle \varepsilon,\ e,\ s'\rangle$$

where $\mathrm{k_1} + \mathrm{k_2} = \mathrm{k_0} + 1$. From Exercises 4.19 and 4.6, we get that e'' must be $\mathcal{B}[\![b]\!]s$ and s'' must be s. We shall now assume that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Then there must be a configuration $\langle \mathcal{CS}[\![S_1]\!],\ \varepsilon,\ s\rangle$ such that

$$\langle \mathcal{CS}[\![S_1]\!],\ \varepsilon,\ s\rangle \rhd^{\mathrm{k_2}-1} \langle \varepsilon,\ e,\ s'\rangle$$

The induction hypothesis can now be applied to this computation sequence because $\mathrm{k_2} - 1 \leq \mathrm{k_0}$ and we get

$$\langle S_1,\ s\rangle \to s' \text{ and } e = \varepsilon$$

The rule [if$^{\text{tt}}_{\text{ns}}$] gives the required $\langle$`if` b `then` S_1 `else` $S_2,\ s\rangle \to s'$. The case where $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ is similar.

The case `while` b `do` S: The code for the `while`-loop is $\text{LOOP}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!])$ and we therefore assume that

$$\langle \text{LOOP}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \varepsilon,\ s\rangle \rhd^{\mathrm{k_0}+1} \langle \varepsilon,\ e,\ s''\rangle$$

Using the definition of the LOOP-instruction, this means that the computation sequence can be rewritten as

$$\begin{aligned}
&\langle \text{LOOP}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \varepsilon,\ s\rangle \\
&\rhd \langle \mathcal{CB}[\![b]\!]{:}\text{BRANCH}(\mathcal{CS}[\![S]\!]{:}\text{LOOP}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \text{NOOP}),\ \varepsilon,\ s\rangle \\
&\rhd^{\mathrm{k_0}} \langle \varepsilon,\ e,\ s''\rangle
\end{aligned}$$

According to Exercise 4.5, there will then be a configuration $\langle \varepsilon,\ e',\ s'\rangle$ such that

$$\langle \mathcal{CB}[\![b]\!],\ \varepsilon,\ s\rangle \rhd^{\mathrm{k_1}} \langle \varepsilon,\ e',\ s'\rangle$$

and

$$\langle \textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \textsc{noop}),\ e',\ s'\rangle \rhd^{k_2} \langle \varepsilon,\ e,\ s''\rangle$$

where $k_1 + k_2 = k_0$. From Exercises 4.19 and 4.6, we get $e' = \mathcal{B}[\![b]\!]s$ and $s' = s$. We now have two cases.

In the first case, assume that $\mathcal{B}[\![b]\!]s = \mathbf{ff}$. We then have

$$\begin{aligned}
&\langle \textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \textsc{noop}),\ \mathcal{B}[\![b]\!]s,\ s\rangle\\
&\rhd \langle \textsc{noop},\ \varepsilon,\ s\rangle\\
&\rhd \langle \varepsilon,\ \varepsilon,\ s\rangle
\end{aligned}$$

so $e = \varepsilon$ and $s = s''$. Using rule [while$^{\text{ff}}_{\text{ns}}$], we get $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$ as required.

In the second case, assume that $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Then we have

$$\begin{aligned}
&\langle \textsc{branch}(\mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \textsc{noop}),\ \mathcal{B}[\![b]\!]s,\ s\rangle\\
&\rhd \langle \mathcal{CS}[\![S]\!]{:}\textsc{loop}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \varepsilon,\ s\rangle\\
&\rhd^{k_2-1} \langle \varepsilon,\ e,\ s''\rangle
\end{aligned}$$

We then proceed very much as in the case of the composition statement and get a configuration $\langle \varepsilon,\ e',\ s'\rangle$ such that

$$\langle \mathcal{CS}[\![S]\!],\ \varepsilon,\ s\rangle \rhd^{k_3} \langle \varepsilon,\ e',\ s'\rangle$$

and

$$\langle \textsc{loop}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ e',\ s'\rangle \rhd^{k_4} \langle \varepsilon,\ e,\ s''\rangle$$

where $k_3 + k_4 = k_2 - 1$. Since $k_3 \le k_0$, we can apply the induction hypothesis to the first of these computation sequences and get

$$\langle S,\ s\rangle \to s' \text{ and } e' = \varepsilon$$

We can then use that $k_4 \le k_0$ and apply the induction hypothesis to the computation sequence $\langle \textsc{loop}(\mathcal{CB}[\![b]\!],\ \mathcal{CS}[\![S]\!]),\ \varepsilon,\ s'\rangle \rhd^{k_4} \langle \varepsilon,\ e,\ s''\rangle$ and get

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s'' \text{ and } e = \varepsilon$$

Using rule [while$^{\text{tt}}_{\text{ns}}$], we then get $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$ as required. This completes the proof of the lemma. □

The proof technique employed in the proof above may be summarized as follows:

Proof Summary for While:

Correctness of Implementation

1: Prove by *induction on the shape of derivation trees* that for each derivation tree in the natural semantics there is a corresponding finite computation sequence on the abstract machine.

2: Prove by *induction on the length of computation sequences* that for each finite computation sequence obtained from executing a statement of **While** on the abstract machine there is a corresponding derivation tree in the natural semantics.

Note the *similarities* between this proof technique and that for showing the equivalence of two operational semantics (see Section 2.3). Again one has to be careful when adapting this approach to a language with additional programming constructs or a different machine language.

Exercise 4.23

Consider the "optimized" code generation function $\mathcal{CS}'$ that is like $\mathcal{CS}$ of Table 4.3 except that $\mathcal{CS}'[\![\texttt{skip}]\!] = \varepsilon$. Would this complicate the proof of Theorem 4.20? $\square$

Exercise 4.24

Extend the proof of Theorem 4.20 to hold for the **While** language extended with `repeat` S `until` b. The code generated for this construct was studied in Exercise 4.14 and its natural semantics in Exercise 2.7. $\square$

Exercise 4.25

Prove that the code generated for $\mathbf{AM}_1$ in Exercise 4.16 is correct. What assumptions do you need to make about *env*? $\square$

Exercise 4.26 (**)

The abstract machine uses disjoint representations of integer and boolean values. For example, the code sequence

$$\text{PUSH-}5\text{:FALSE:ADD}$$

will get stuck rather than compute some strange result.

Now reconsider the development of this chapter and suppose that the stack only holds integers. For this to work, the instructions operating on the truth values **ff** and **tt** instead operate on integers **0** and **1**. How should the development of the present chapter be modified? In particular, how would the correctness proof need to be modified? □

4.4 An Alternative Proof Technique

In Theorem 4.20, we proved the correctness of the implementation with respect to the natural semantics. It is obvious that the implementation will also be correct with respect to the structural operational semantics; that is,

$$\mathcal{S}_{\text{sos}}[\![S]\!] = \mathcal{S}_{\text{am}}[\![S]\!] \text{ for all statements } S \text{ of } \textbf{While}$$

because we showed in Theorem 2.26 that the natural semantics is equivalent to the structural operational semantics. However, one might argue that it would be easier to give a direct proof of the correctness of the implementation with respect to the structural operational semantics because both approaches are based on the idea of specifying the individual steps of the computation. We shall comment upon this shortly.

A direct proof of the correctness result with respect to the structural operational semantics could proceed as follows. We shall define a *bisimulation* relation $\approx$ between the configurations of the structural operational semantics and those of the operational semantics for **AM**. It is defined by

$$\begin{array}{rcl} \langle S,\, s\rangle & \approx & \langle \mathcal{CS}[\![S]\!],\, \varepsilon,\, s\rangle \\ s & \approx & \langle \varepsilon,\, \varepsilon,\, s\rangle \end{array}$$

for all statements S and states s. The first stage will then be to prove that whenever *one* step of the structural operational semantics *changes* the configuration, then there is a *sequence* of steps in the semantics of **AM** that will make a *similar change* in the configuration of the abstract machine.

Exercise 4.27 (*)

Show that if

$$\gamma_{\text{sos}} \approx \gamma_{\text{am}} \text{ and } \gamma_{\text{sos}} \Rightarrow \gamma'_{\text{sos}}$$

then there exists a configuration γ'_{am} such that

$$\gamma_{\text{am}} \rhd^{+} \gamma'_{\text{am}} \text{ and } \gamma'_{\text{sos}} \approx \gamma'_{\text{am}}$$

Argue that if $\langle S,\, s\rangle \Rightarrow^{*} s'$, then $\langle \mathcal{CS}[\![S]\!],\, \varepsilon,\, s\rangle \rhd^{*} \langle \varepsilon,\, \varepsilon,\, s'\rangle$. □

The second part of the proof is to show that whenever **AM** makes a sequence of moves from a configuration with an *empty* evaluation stack to another configuration with an *empty* evaluation stack, then the structural operational semantics can make a similar change of configurations. Note that **AM** may have to make more than one step to arrive at a configuration with an empty stack due to the way it evaluates expressions; in the structural operational semantics, however, expressions are evaluated as part of a single step.

Exercise 4.28 (**)

Assume that $\gamma_{\text{sos}} \approx \gamma_{\text{am}}^{1}$ and

$$\gamma_{\text{am}}^{1} \rhd \gamma_{\text{am}}^{2} \rhd \cdots \rhd \gamma_{\text{am}}^{\text{k}}$$

where k>1 and only γ_{am}^{1} and $\gamma_{\text{am}}^{\text{k}}$ have empty evaluation stacks (that is, are of the form $\langle c,\, \varepsilon,\, s\rangle$). Show that there exists a configuration γ'_{sos} such that

$$\gamma_{\text{sos}} \Rightarrow \gamma'_{\text{sos}} \text{ and } \gamma'_{\text{sos}} \approx \gamma_{\text{am}}^{\text{k}}$$

Argue that if $\langle \mathcal{CS}[\![S]\!],\, \varepsilon,\, s\rangle \rhd^{*} \langle \varepsilon,\, \varepsilon,\, s'\rangle$, then $\langle S,\, s\rangle \Rightarrow^{*} s'$. □

Exercise 4.29

Show that Exercises 4.27 and 4.28 together constitute a direct proof of $\mathcal{S}_{\text{sos}}[\![S]\!] = \mathcal{S}_{\text{am}}[\![S]\!]$ for all statements S of **While**. □

The success of this approach relies on the two semantics proceeding in *lock-step*: that one is able to find configurations in the two derivation sequences that correspond to one another (as specified by the bisimulation relation). Often this is not possible and then one has to raise the level of abstraction for one of the semantics. This is exactly what happens when the structural operational semantics is replaced by the natural semantics: we do not care about the individual steps of the execution but only about the result.

The proof technique employed in the sketch of proof above may be summarized as follows:

Proof Summary for While:

Correctness of Implementation Using Bisimulation

1: Prove that one step in the structural operational semantics can be simulated by a non-empty sequence of steps on the abstract machine. Show that this extends to sequences of steps in the structural operational semantics.

2: Prove that a carefully selected non-empty sequence of steps on the abstract machine can be simulated by a step in the structural operational semantics. Show that this extends to more general sequences of steps on the abstract machine.

Again, this method may need to be modified when considering a programming language with additional constructs or a different abstract machine.

Exercise 4.30 (*)

Consider the following, seemingly innocent, modification of the structural operational semantics of Table 2.2 in which [while$_{\text{sos}}$] is replaced by the two axioms

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow \langle S; \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow s \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Show that the modified semantic function, $\mathcal{S}'_{\text{sos}}$, satisfies

$$\mathcal{S}_{\text{sos}}[\![S]\!] = \mathcal{S}'_{\text{sos}}[\![S]\!] \text{ for all statements } S \text{ of } \textbf{While}$$

Investigate whether or not this complicates the proofs of (analogues of) Exercises 4.27 and 4.28. □

5
Denotational Semantics

In the operational approach, we were interested in *how* a program is executed. This is contrary to the denotational approach, where we are merely interested in the *effect* of executing a program. By effect we mean here an association between initial states and final states. The idea then is to define a *semantic function* for each *syntactic category.* It maps each *syntactic construct* to a *mathematical object*, often a function, that describes the effect of executing that construct.

The hallmark of denotational semantics is that semantic functions are defined *compositionally*; that is,

– there is a semantic clause for each of the basis elements of the syntactic category, and

– for each method of constructing a composite element (in the syntactic category) there is a semantic clause defined in terms of the semantic function applied to the immediate constituents of the composite element.

The functions $\mathcal{A}$ and $\mathcal{B}$ defined in Chapter 1 are examples of denotational definitions: the mathematical objects associated with arithmetic expressions are functions in $\mathbf{State} \to \mathbf{Z}$ and those associated with boolean expressions are functions in $\mathbf{State} \to \mathbf{T}$. The functions $\mathcal{S}_{\mathrm{ns}}$ and $\mathcal{S}_{\mathrm{sos}}$ introduced in Chapter 2 associate mathematical objects with each statement, namely partial functions in $\mathbf{State} \hookrightarrow \mathbf{State}$. However, they are *not* examples of denotational definitions because they are *not* defined compositionally.

$$
\begin{array}{rcl}
\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s & = & s[x\mapsto\mathcal{A}[\![a]\!]s] \\
\mathcal{S}_{\mathrm{ds}}[\![\texttt{skip}]\!] & = & \mathrm{id} \\
\mathcal{S}_{\mathrm{ds}}[\![S_1 \texttt{ ; } S_2]\!] & = & \mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!] \\
\mathcal{S}_{\mathrm{ds}}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\mathrm{ds}}[\![S_1]\!], \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]) \\
\mathcal{S}_{\mathrm{ds}}[\![\texttt{while } b \texttt{ do } S]\!] & = & \mathsf{FIX}\ F \\
\text{where } F\ g & = & \mathsf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!], \mathrm{id})
\end{array}
$$

Table 5.1 Denotational semantics for **While**

5.1 Direct Style Semantics: Specification

The effect of executing a statement S is to change the state so we shall define the meaning of S to be a partial function on states:

$$\mathcal{S}_{\mathrm{ds}}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

This is also the functionality of $\mathcal{S}_{\mathrm{ns}}$ and $\mathcal{S}_{\mathrm{sos}}$, and the need for partiality is again demonstrated by the statement `while true do skip`. The definition is summarized in Table 5.1 and we explain it in detail below; in particular, we shall define the *auxiliary functions* cond and FIX.

For assignment, the clause

$$\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s = s[x\mapsto\mathcal{A}[\![a]\!]s]$$

ensures that if $\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s = s'$, then $s'\ x = \mathcal{A}[\![a]\!]s$ and $s'\ y = s\ y$ for $y{\neq}x$. The clause for `skip` expresses that no state change takes place: the function id is the identity function on **State** so $\mathcal{S}_{\mathrm{ds}}[\![\texttt{skip}]\!]s = s$.

For sequencing, the clause is

$$\mathcal{S}_{\mathrm{ds}}[\![S_1 \texttt{ ; } S_2]\!] = \mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]$$

So the effect of executing S_1 ; S_2 is the functional composition of the effect of executing S_1 and that of executing S_2. Functional composition is defined such that if one of the functions is undefined on a given argument, then their composition is undefined as well. Given a state s, we therefore have

$$\begin{aligned}\mathcal{S}_{\mathrm{ds}}[\![S_1\ ;\ S_2]\!]s &= (\mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!])s \\ &= \begin{cases} s'' & \text{if there exists } s' \text{ such that } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = s' \\ & \text{and } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s' = s'' \\ \underline{\text{undef}} & \text{if } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = \underline{\text{undef}} \\ & \text{or if there exists } s' \text{ such that } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = s' \\ & \text{but } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s' = \underline{\text{undef}} \end{cases}\end{aligned}$$

It follows that the sequencing construct will only give a defined result if both components do.

For conditional, the clause is

$$\mathcal{S}_{\mathrm{ds}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!],\ \mathcal{S}_{\mathrm{ds}}[\![S_2]\!])$$

and the auxiliary function cond has functionality

$$\begin{aligned}\mathsf{cond}\colon\ & (\mathbf{State} \to \mathbf{T}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \\ & \to (\mathbf{State} \hookrightarrow \mathbf{State})\end{aligned}$$

and is defined by

$$\mathsf{cond}(p,\ g_1,\ g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \mathbf{tt} \\ g_2\ s & \text{if } p\ s = \mathbf{ff} \end{cases}$$

The first parameter to cond is a function that, when supplied with an argument, will select either the second or the third parameter of cond and then supply that parameter with the same argument. Thus we have

$$\begin{aligned}&\mathcal{S}_{\mathrm{ds}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\ s \\ =\ & \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!],\ \mathcal{S}_{\mathrm{ds}}[\![S_2]\!])\ s \\ =\ & \begin{cases} s' & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = s' \\ & \text{or if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \text{ and } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s = s' \\ \underline{\text{undef}} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = \underline{\text{undef}} \\ & \text{or if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \text{ and } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s = \underline{\text{undef}} \end{cases}\end{aligned}$$

So if the selected branch gives a defined result then so does the conditional. Note that since $\mathcal{B}[\![b]\!]$ is a total function, $\mathcal{B}[\![b]\!]s$ cannot be undef.

Defining the effect of `while` b `do` S is a major task. To motivate the actual definition, we first observe that the effect of `while` b `do` S must equal that of

$$\texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}$$

Using the parts of $\mathcal{S}_{\rm ds}$ that have already been defined, this gives

$$\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \textsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}_{\rm ds}[\![S]\!],\ \mathrm{id}) \qquad (*)$$

Note that we cannot use (*) as the definition of $\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ because then $\mathcal{S}_{\rm ds}$ would *not* be a compositional definition. However, (*) expresses that

$$\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$$

must be a *fixed point* of the functional F defined by

$$F\ g = \textsf{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}_{\rm ds}[\![S]\!],\ \mathrm{id})$$

that is $\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = F\ (\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!])$. In this way, we will get a compositional definition of $\mathcal{S}_{\rm ds}$ because when defining F we only apply $\mathcal{S}_{\rm ds}$ to the immediate constituents of $\texttt{while}\ b\ \texttt{do}\ S$ and not to the construct itself. Thus we write

$$\begin{aligned}&\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \textsf{FIX}\ F\\ &\text{where } F\ g = \textsf{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}_{\rm ds}[\![S]\!],\ \mathrm{id})\end{aligned}$$

to indicate that $\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ is a fixed point of F. The functionality of the auxiliary function $\textsf{FIX}$ is

$$\textsf{FIX}\colon ((\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})) \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

Example 5.1

Consider the statement

$$\texttt{while}\ \neg(\texttt{x} = \texttt{0})\ \texttt{do skip}$$

It is easy to verify that the corresponding functional F' is defined by

$$(F'\ g)\ s = \begin{cases} g\ s & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases}$$

The function g_1 defined by

$$g_1\ s = \begin{cases} \underline{\mathrm{undef}} & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases}$$

is a fixed point of F' because

$$\begin{aligned}(F'\ g_1)\ s &= \begin{cases} g_1\ s & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases}\\ &= \begin{cases} \underline{\mathrm{undef}} & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases}\\ &= g_1\ s\end{aligned}$$

Next we claim that the function g_2 defined by

$$g_2\ s = \underline{\text{undef}} \text{ for all } s$$

cannot be a fixed point for F'. The reason is that if s' is a state with $s'\ \mathtt{x} = \mathbf{0}$, then $(F'\ g_2)\ s' = s'$, whereas $g_2\ s' = \underline{\text{undef}}$. □

Unfortunately, this does *not* suffice for defining $\mathcal{S}_{\text{ds}}[\![\mathtt{while}\ b\ \mathtt{do}\ S]\!]$. We face two problems:

– there are functionals that have *more than one fixed point*, and

– there are functionals that have *no fixed point* at all.

The functional F' of Example 5.1 has more than one fixed point. In fact, *every* function g' of **State** $\hookrightarrow$ **State** satisfying $g'\ s = s$ if $s\ \mathtt{x} = \mathbf{0}$ will be a fixed point of F'.

For an example of a functional that has no fixed points, consider F_1 defined by

$$F_1\ g = \begin{cases} g_1 & \text{if } g = g_2 \\ g_2 & \text{otherwise} \end{cases}$$

If $g_1 \neq g_2$, then clearly there will be no function g_0 such that $F_1\ g_0 = g_0$. Thus F_1 has no fixed points at all.

Exercise 5.2

Determine the functional F associated with the statement

$$\mathtt{while}\ \neg(\mathtt{x}{=}0)\ \mathtt{do}\ \mathtt{x} := \mathtt{x}{-}1$$

using the semantic equations of Table 5.1. Consider the following partial functions of **State** $\hookrightarrow$ **State**:

$$\begin{aligned}
g_1\ s &= \underline{\text{undef}} \text{ for all } s \\
g_2\ s &= \begin{cases} s[\mathtt{x}\mapsto\mathbf{0}] & \text{if } s\ \mathtt{x} \geq \mathbf{0} \\ \underline{\text{undef}} & \text{if } s\ \mathtt{x} < \mathbf{0} \end{cases} \\
g_3\ s &= \begin{cases} s[\mathtt{x}\mapsto\mathbf{0}] & \text{if } s\ \mathtt{x} \geq \mathbf{0} \\ s & \text{if } s\ \mathtt{x} < \mathbf{0} \end{cases} \\
g_4\ s &= s[\mathtt{x}\mapsto\mathbf{0}] \text{ for all } s \\
g_5\ s &= s \text{ for all } s
\end{aligned}$$

Determine which of these functions are fixed points of F. □

Exercise 5.3

Consider the following fragment of the factorial statement:

$$\texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do}\ (\texttt{y} := \texttt{y}\star\texttt{x};\ \texttt{x} := \texttt{x}-1)$$

Determine the functional F associated with this statement. Determine at least two different fixed points for F. $\square$

Requirements on the Fixed Point

Our solution to the two problems listed above will be to develop a framework where

– we impose requirements on the fixed points and show that there is at most one fixed point fulfilling these requirements, and

– all functionals originating from statements in **While** do have a fixed point that satisfies these requirements.

To motivate our choice of requirements, let us consider the execution of a statement `while` b `do` S from a state s_0. There are three possible outcomes:

A: It *terminates*.

B: It *loops locally*; that is, there is a construct in S that loops.

C: It *loops globally*; that is, the outer `while`-construct loops.

We shall now investigate what can be said about the functional F and its fixed points in each of the three cases.

The case A: In this case, the execution of `while` b `do` S from s_0 terminates. This means that there are states $s_1, \cdots, s_n$ such that

$$\mathcal{B}[\![b]\!]\ s_i = \begin{cases} \textbf{tt} & \text{if i<n} \\ \textbf{ff} & \text{if i=n} \end{cases}$$

and

$$\mathcal{S}_{\text{ds}}[\![S]\!]\ s_i = s_{i+1} \quad \text{for i<n}$$

An example of a statement and a state satisfying these conditions is the statement

$$\texttt{while}\ 0{\le}\texttt{x}\ \texttt{do}\ \texttt{x} := \texttt{x}-1$$

and any state where `x` has a non-negative value.

Let g_0 be any fixed point of F; that is, assume that $F\ g_0 = g_0$. In the case where i<n, we calculate

$$\begin{aligned} g_0\ s_\mathrm{i} &= (F\ g_0)\ s_\mathrm{i} \\ &= \mathsf{cond}(\mathcal{B}[\![b]\!],\ g_0 \circ \mathcal{S}_\mathrm{ds}[\![S]\!],\ \mathrm{id})\ s_\mathrm{i} \\ &= g_0\ (\mathcal{S}_\mathrm{ds}[\![S]\!]\ s_\mathrm{i}) \\ &= g_0\ s_\mathrm{i+1} \end{aligned}$$

In the case where i=n, we get

$$\begin{aligned} g_0\ s_\mathrm{n} &= (F\ g_0)\ s_\mathrm{n} \\ &= \mathsf{cond}(\mathcal{B}[\![b]\!],\ g_0 \circ \mathcal{S}_\mathrm{ds}[\![S]\!],\ \mathrm{id})\ s_\mathrm{n} \\ &= \mathrm{id}\ s_\mathrm{n} \\ &= s_\mathrm{n} \end{aligned}$$

Thus *every* fixed point g_0 of F will satisfy

$$g_0\ s_0 = s_\mathrm{n}$$

so in this case we do not obtain any additional requirements that will help us to choose one of the fixed points as the preferred one.

The case B: In this case, the execution of `while` b `do` S from s_0 loops *locally*. This means that there are states $s_1, \cdots, s_\mathrm{n}$ such that

$$\mathcal{B}[\![b]\!]s_\mathrm{i} = \mathbf{tt} \text{ for i} \leq \text{n}$$

and

$$\mathcal{S}_\mathrm{ds}[\![S]\!]s_\mathrm{i} = \begin{cases} s_\mathrm{i+1} & \text{for i<n} \\ \underline{\mathrm{undef}} & \text{for i=n} \end{cases}$$

An example of a statement and a state satisfying these conditions is the statement

```
while 0≤x do (if x=0 then (while true do skip)
                      else x := x−1)
```

and any state where `x` has a non-negative value.

Let g_0 be any fixed point of F; that is, $F\ g_0 = g_0$. In the case where i<n, we obtain

$$g_0\ s_\mathrm{i} = g_0\ s_\mathrm{i+1}$$

just as in the previous case. However, in the case where i=n, we get

$$\begin{aligned} g_0\ s_\mathrm{n} &= (F\ g_0)\ s_\mathrm{n} \\ &= \mathsf{cond}(\mathcal{B}[\![b]\!],\ g_0 \circ \mathcal{S}_\mathrm{ds}[\![S]\!],\ \mathrm{id})\ s_\mathrm{n} \\ &= (g_0 \circ \mathcal{S}_\mathrm{ds}[\![S]\!])\ s_\mathrm{n} \\ &= \underline{\mathrm{undef}} \end{aligned}$$

Thus *any* fixed point g_0 of F will satisfy

$$g_0\ s_0 = \underline{\text{undef}}$$

so, again, in this case we do not obtain any additional requirements that will help us to choose one of the fixed points as the preferred one.

The case C: The potential difference between fixed points comes to light when we consider the possibility that the execution of `while` b `do` S from s_0 loops *globally*. This means that there is an infinite sequence of states $s_1, \cdots$ such that

$$\mathcal{B}[\![b]\!]s_\text{i} = \textbf{tt} \text{ for all i}$$

and

$$\mathcal{S}_\text{ds}[\![S]\!]s_\text{i} = s_{\text{i}+1} \text{ for all i.}$$

An example of a statement and a state satisfying these conditions is the statement

$$\texttt{while}\ \neg(\texttt{x}{=}0)\ \texttt{do skip}$$

and any state where `x` is not equal to **0**.

Let g_0 be any fixed point of F; that is, $F\ g_0 = g_0$. As in the previous cases, we get

$$g_0\ s_\text{i} = g_0\ s_{\text{i}+1}$$

for all i≥0. Thus we have

$$g_0\ s_0 = g_0\ s_\text{i} \text{ for all i}$$

and we cannot determine the value of $g_0\ s_0$ in this way. This is the situation in which the various fixed points of F may differ.

This is not surprising because the statement `while` ¬(x=0) `do skip` of Example 5.1 has the functional F' given by

$$(F'\ g)\ s = \begin{cases} g\ s & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases}$$

and *any* partial function g of **State** ↪ **State** satisfying $g\ s = s$ if $s\ \texttt{x} = \mathbf{0}$ will indeed be a fixed point of F'. However, our computational experience tells us that we want

$$\mathcal{S}_\text{ds}[\![\texttt{while}\ \neg(\texttt{x}{=}0)\ \texttt{do skip}]\!]s_0 = \begin{cases} \underline{\text{undef}} & \text{if } s_0\ \texttt{x} \neq \mathbf{0} \\ s_0 & \text{if } s_0\ \texttt{x} = \mathbf{0} \end{cases}$$

in order to record the looping. Thus our preferred fixed point of F' is the function g_0 defined by

$$g_0\ s = \begin{cases} \underline{\text{undef}} & \text{if } s\ \mathtt{x} \neq \mathbf{0} \\ s & \text{if } s\ \mathtt{x} = \mathbf{0} \end{cases}$$

The property that distinguishes g_0 from some other fixed point g' of F' is that whenever $g_0\ s = s'$ then we also have $g'\ s = s'$ but not vice versa.

Generalizing this experience leads to the following requirement: the desired fixed point $\mathsf{FIX}\ F$ should be a partial function g_0: **State** $\hookrightarrow$ **State** such that

– g_0 is a fixed point of F (that is, $F\ g_0 = g_0$), and

– if g is another fixed point of F (that is, $F\ g = g$), then

$$g_0\ s = s' \text{ implies } g\ s = s'$$

for all choices of s and s'.

Note that if $g_0\ s = \underline{\text{undef}}$, then there are no requirements on $g\ s$.

Exercise 5.4

Determine which of the fixed points considered in Exercise 5.2, if any, is the desired fixed point. □

Exercise 5.5

Determine the desired fixed point of the functional from Exercise 5.3. □

5.2 Fixed Point Theory

To prepare for a framework that guarantees the existence of the desired fixed point $\mathsf{FIX}\ F$, we shall reformulate the requirements to $\mathsf{FIX}\ F$ in a slightly more formal way. The first step will be to formalize the requirement that $\mathsf{FIX}\ F$ shares its results with all other fixed points. To do so, we define an *ordering* $\sqsubseteq$ on partial functions of **State** $\hookrightarrow$ **State**. We set

$$g_1 \sqsubseteq g_2$$

when the partial function g_1: **State** $\hookrightarrow$ **State** *shares its results* with the partial function g_2: **State** $\hookrightarrow$ **State** in the sense that

$$\text{if } g_1\ s = s' \text{ then } g_2\ s = s'$$

for all choices of s and s'.

Example 5.6

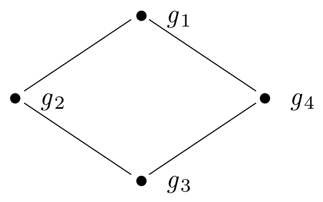
Let g_1, g_2, g_3, and g_4 be partial functions in **State** $\hookrightarrow$ **State** defined as follows:

$$\begin{aligned}
g_1\ s &= s \text{ for all } s \\
g_2\ s &= \begin{cases} s & \text{if } s\ \mathtt{x} \geq \mathbf{0} \\ \underline{\text{undef}} & \text{otherwise} \end{cases} \\
g_3\ s &= \begin{cases} s & \text{if } s\ \mathtt{x} = \mathbf{0} \\ \underline{\text{undef}} & \text{otherwise} \end{cases} \\
g_4\ s &= \begin{cases} s & \text{if } s\ \mathtt{x} \leq \mathbf{0} \\ \underline{\text{undef}} & \text{otherwise} \end{cases}
\end{aligned}$$

Then we have

$$\begin{aligned}
&g_1 \sqsubseteq g_1, \\
&g_2 \sqsubseteq g_1,\ g_2 \sqsubseteq g_2, \\
&g_3 \sqsubseteq g_1,\ g_3 \sqsubseteq g_2,\ g_3 \sqsubseteq g_3,\ g_3 \sqsubseteq g_4, \text{ and} \\
&g_4 \sqsubseteq g_1,\ g_4 \sqsubseteq g_4.
\end{aligned}$$

It is neither the case that $g_2 \sqsubseteq g_4$ nor that $g_4 \sqsubseteq g_2$. Pictorially, the ordering may be expressed by the following diagram (sometimes called a Hasse diagram):



The idea is that the smaller elements are at the bottom of the picture and that the lines indicate the order between the elements. However, we shall not draw lines when there already is a "broken line", so the fact that $g_3 \sqsubseteq g_1$ is left implicit in the picture. □

Exercise 5.7

Let g_1, g_2, and g_3 be defined as follows:

$$g_1\ s = \begin{cases} s & \text{if } s\ \mathtt{x} \text{ is even} \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

$$\begin{aligned} g_2\ s &= \begin{cases} s & \text{if } s\ \mathtt{x} \text{ is a prime} \\ \underline{\text{undef}} & \text{otherwise} \end{cases} \\ g_3\ s &= s \end{aligned}$$

First, determine the ordering among these partial functions. Next, determine a partial function g_4 such that $g_4 \sqsubseteq g_1$, $g_4 \sqsubseteq g_2$, and $g_4 \sqsubseteq g_3$. Finally, determine a partial function g_5 such that $g_1 \sqsubseteq g_5$, $g_2 \sqsubseteq g_5$, and $g_5 \sqsubseteq g_3$ but g_5 is equal to neither g_1, g_2, nor g_3. □

Exercise 5.8 (Essential)

An alternative characterization of the ordering $\sqsubseteq$ on **State** $\hookrightarrow$ **State** is

$$g_1 \sqsubseteq g_2 \text{ if and only if } \text{graph}(g_1) \subseteq \text{graph}(g_2) \qquad (*)$$

where graph(g) is the graph of the partial function g as defined in Appendix A. Prove that (*) is indeed correct. □

The set **State** $\hookrightarrow$ **State** equipped with the ordering $\sqsubseteq$ is an example of a partially ordered set, as we shall see in Lemma 5.13 below. In general, a *partially ordered set* is a pair $(D, \sqsubseteq_D)$, where D is a set and $\sqsubseteq_D$ is a relation on D satisfying

$$\begin{array}{ll} d \sqsubseteq_D d & \text{(reflexivity)} \\ d_1 \sqsubseteq_D d_2 \text{ and } d_2 \sqsubseteq_D d_3 \text{ imply } d_1 \sqsubseteq_D d_3 & \text{(transitivity)} \\ d_1 \sqsubseteq_D d_2 \text{ and } d_2 \sqsubseteq_D d_1 \text{ imply } d_1 = d_2 & \text{(anti-symmetry)} \end{array}$$

for all d, d_1 and d_1 in D.

The relation $\sqsubseteq_D$ is said to be a *partial order* on D and we shall often omit the subscript D of $\sqsubseteq_D$ and write $\sqsubseteq$. Occasionally, we may write $d_1 \sqsupseteq d_2$ instead of $d_2 \sqsubseteq d_1$, and we shall say that d_2 *shares its information with* d_1. An element d of D satisfying

$$d \sqsubseteq d' \text{ for all } d' \text{ of } D$$

is called *a least element* of D, and we shall say that it contains *no information.*

Fact 5.9

If a partially ordered set $(D, \sqsubseteq)$ has a least element d, then d is unique.

Proof: Assume that D has two least elements d_1 and d_2. Since d_1 is a least element, we have $d_1 \sqsubseteq d_2$. Since d_2 is a least element, we also have $d_2 \sqsubseteq d_1$.

The anti-symmetry of the ordering $\sqsubseteq$ then gives that $d_1 = d_2$. $\square$

This fact permits us to talk about *the* least element of D, if one exists, and we shall denote it by $\perp_D$ or simply $\perp$ (pronounced "bottom").

Example 5.10

For simplicity, let S be a non-empty set, and define

$$\mathcal{P}(S) = \{ \ K \mid K \subseteq S \ \}$$

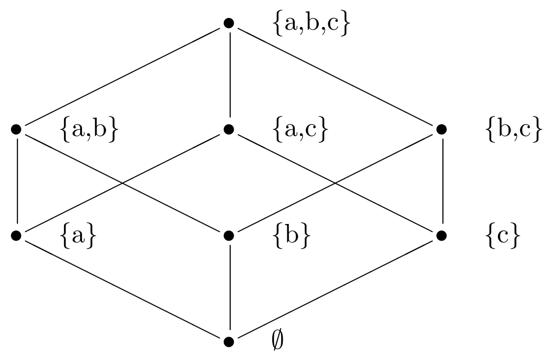
Then $(\mathcal{P}(S), \subseteq)$ is a partially ordered set because

– $\subseteq$ is reflexive: $K \subseteq K$

– $\subseteq$ is transitive: if $K_1 \subseteq K_2$ and $K_2 \subseteq K_3$ then $K_1 \subseteq K_3$

– $\subseteq$ is anti-symmetric: if $K_1 \subseteq K_2$ and $K_2 \subseteq K_1$ then $K_1 = K_2$

In the case where $S = \{$a,b,c$\}$, the ordering can be depicted as follows:



Also, $(\mathcal{P}(S), \subseteq)$ has a least element, namely $\emptyset$. $\square$

Exercise 5.11

Show that $(\mathcal{P}(S), \supseteq)$ is a partially ordered set, and determine the least element. Draw a picture of the ordering when $S = \{$a,b,c$\}$. $\square$

Exercise 5.12

Let S be a non-empty set, and define

$$\mathcal{P}_{\text{fin}}(S) = \{ \ K \mid K \text{ is finite and } K \subseteq S \ \}$$

Verify that $(\mathcal{P}_{\text{fin}}(S),\ \subseteq)$ and $(\mathcal{P}_{\text{fin}}(S),\ \supseteq)$ are partially ordered sets. Do both partially ordered sets have a least element for all choices of S? $\square$

Lemma 5.13

(**State** $\hookrightarrow$ **State**, $\sqsubseteq$) is a partially ordered set. The partial function $\perp$: **State** $\hookrightarrow$ **State** defined by

$$\perp\ s = \underline{\text{undef}} \text{ for all } s$$

is the least element of **State** $\hookrightarrow$ **State**.

Proof: We shall first prove that $\sqsubseteq$ fulfils the three requirements to a partial order. Clearly, $g \sqsubseteq g$ holds because $g\ s = s'$ trivially implies that $g\ s = s'$ so $\sqsubseteq$ is a *reflexive* ordering.

To see that it is a *transitive* ordering, assume that $g_1 \sqsubseteq g_2$ and $g_2 \sqsubseteq, g_3$ and we shall prove that $g_1 \sqsubseteq g_3$. Assume that $g_1\ s = s'$. From $g_1 \sqsubseteq g_2$, we get $g_2\ s = s'$, and then $g_2 \sqsubseteq g_3$ gives that $g_3\ s = s'$.

To see that it is an *anti-symmetric* ordering, assume that $g_1 \sqsubseteq g_2$ and $g_2 \sqsubseteq g_1$, and we shall then prove that $g_1 = g_2$. Assume that $g_1\ s = s'$. Then $g_2\ s = s'$ follows from $g_1 \sqsubseteq g_2$, so g_1 and g_2 are equal on s. If $g_1\ s = \underline{\text{undef}}$, then it must be the case that $g_2\ s = \underline{\text{undef}}$ since otherwise $g_2\ s = s'$ and the assumption $g_2 \sqsubseteq g_1$ then gives $g_1\ s = s'$, which is a contradiction. Thus g_1 and g_2 will be equal on s.

Finally, we shall prove that $\perp$ is the *least element* of **State** $\hookrightarrow$ **State**. It is easy to see that $\perp$ is indeed an element of **State** $\hookrightarrow$ **State**, and it is also obvious that $\perp \sqsubseteq g$ holds for all g since $\perp\ s = s'$ vacuously implies that $g\ s = s'$. $\square$

Having introduced an ordering on the partial functions, we can now give a more precise statement of the requirements to FIX F:

- FIX F is a *fixed point* of F (that is, $F(\text{FIX } F) = \text{FIX } F$), and
- FIX F is a *least* fixed point of F; that is,

$$\text{if } F\ g = g \text{ then FIX } F \sqsubseteq g.$$

Exercise 5.14

By analogy with Fact 5.9, show that if F has a least fixed point g_0, then g_0 is unique. $\square$

The next task will be to ensure that all functionals F that may arise do indeed have least fixed points. We shall do so by developing a general theory that gives more structure to the partially ordered sets and that imposes restrictions on the functionals so that they have least fixed points.

Exercise 5.15

Determine the least fixed points of the functionals considered in Exercises 5.2 and 5.3. Compare them with Exercises 5.4 and 5.5. □

Complete Partially Ordered Sets

Consider a partially ordered set $(D, \sqsubseteq)$ and assume that we have a subset Y of D. We shall be interested in an element of D that summarizes all the information of Y, and this is called an *upper bound* of Y; formally, it is an element d of D such that

$$\forall d' \in Y\colon d' \sqsubseteq d$$

An upper bound d of Y is a *least upper bound* if and only if

$$d' \text{ is an upper bound of } Y \text{ implies that } d \sqsubseteq d'$$

Thus a least upper bound of Y will add as little extra information as possible to that already present in the elements of Y.

Exercise 5.16

By analogy with Fact 5.9, show that if Y has a least upper bound d, then d is unique. □

If Y has a (necessarily unique) least upper bound, we shall denote it by $\bigsqcup Y$. Finally, a subset Y is called a *chain* if it is consistent in the sense that if we take any two elements of Y, then one will share its information with the other; formally, this is expressed by

$$\forall d_1,\ d_2 \in Y\colon d_1 \sqsubseteq d_2 \text{ or } d_2 \sqsubseteq d_1$$

Example 5.17

Consider the partially ordered set $(\mathcal{P}(\{\text{a,b,c}\}), \subseteq)$ of Example 5.10. Then the subset

$$Y_0 = \{\ \emptyset,\ \{\text{a}\},\ \{\text{a,c}\}\ \}$$

is a chain. Both {a,b,c} and {a,c} are upper bounds of Y_0, and {a,c} is the least upper bound. The element {a,b} is *not* an upper bound because {a,c} $\not\subseteq$ {a,b}. In general, the least upper bound of a non-empty chain in $\mathcal{P}(\{\text{a,b,c}\})$ will be the largest element of the chain.

The subset { ∅, {a}, {c}, {a,c} } is *not* a chain because {a} and {c} are unrelated by the ordering. However, it does have a least upper bound, namely {a,c}.

The *subset* ∅ of $\mathcal{P}(\{\text{a,b,c}\})$ is a chain and has any element of $\mathcal{P}(\{\text{a,b,c}\})$ as an upper bound. Its least upper bound is the *element* ∅. □

Exercise 5.18

Let S be a non-empty set, and consider the partially ordered set $(\mathcal{P}(S), \subseteq)$. Show that every subset of $\mathcal{P}(S)$ has a least upper bound. Repeat the exercise for the partially ordered set $(\mathcal{P}(S), \supseteq)$. □

Exercise 5.19

Let S be a non-empty set, and consider the partially ordered set $(\mathcal{P}_{\text{fin}}(S), \subseteq)$ as defined in Exercise 5.12. Show by means of an example that there are choices of S such that $(\mathcal{P}_{\text{fin}}(S), \subseteq)$ has a chain with no upper bound and therefore no least upper bound. □

Example 5.20

Let g_{n}: **State** $\hookrightarrow$ **State** be defined by

$$g_{\text{n}}\ s = \begin{cases} \underline{\text{undef}} & \text{if } s\ \texttt{x} > \text{n} \\ s[\texttt{x}\mapsto\mathbf{-1}] & \text{if } \mathbf{0} \le s\ \texttt{x} \text{ and } s\ \texttt{x} \le \text{n} \\ s & \text{if } s\ \texttt{x} < \mathbf{0} \end{cases}$$

It is straightforward to verify that $g_{\text{n}} \sqsubseteq g_{\text{m}}$ whenever n ≤ m because g_{n} will be undefined for more states than g_{m}. Now define Y_0 to be

$$Y_0 = \{\ g_{\text{n}} \mid \text{n} \ge 0\ \}$$

Then Y_0 is a chain because $g_{\text{n}} \sqsubseteq g_{\text{m}}$ whenever n ≤ m. The partial function

$$g\ s = \begin{cases} s[\texttt{x}\mapsto\mathbf{-1}] & \text{if } \mathbf{0} \le s\ \texttt{x} \\ s & \text{if } s\ \texttt{x} < \mathbf{0} \end{cases}$$

is the least upper bound of Y. □

Exercise 5.21

Construct a subset Y of **State** $\hookrightarrow$ **State** such that Y has no upper bound and hence no least upper bound. $\square$

Exercise 5.22

Let g_n be the partial function defined by

$$g_\mathrm{n}\ s = \begin{cases} s[\mathtt{y}\mapsto(s\ \mathtt{x})!][\mathtt{x}\mapsto\mathbf{1}] & \text{if } \mathbf{0} < s\ \mathtt{x} \text{ and } s\ \mathtt{x} \le \mathrm{n} \\ \underline{\mathrm{undef}} & \text{if } s\ \mathtt{x} \le \mathbf{0} \text{ or } s\ \mathtt{x} > \mathrm{n} \end{cases}$$

(where $m!$ denotes the factorial of m). Define $Y_0 = \{\ g_\mathrm{n} \mid \mathrm{n} \ge 0\ \}$ and show that it is a chain. Characterize the upper bounds of Y_0 and determine the least upper bound. $\square$

A partially ordered set $(D, \sqsubseteq)$ is called a *chain complete* partially ordered set (abbreviated *ccpo*) whenever $\bigsqcup Y$ exists for all chains Y. It is a *complete lattice* if $\bigsqcup Y$ exists for all subsets Y of D.

Example 5.23

Exercise 5.18 shows that $(\mathcal{P}(S), \subseteq)$ and $(\mathcal{P}(S), \supseteq)$ are complete lattices; it follows that they both satisfy the ccpo-property. Exercise 5.19 shows that $(\mathcal{P}_{\mathrm{fin}}(S), \subseteq)$ need not be a complete lattice nor a ccpo. $\square$

Fact 5.24

If $(D, \sqsubseteq)$ is a ccpo, then it has a least element $\bot$ given by $\bot = \bigsqcup\emptyset$.

Proof: It is straightforward to check that $\emptyset$ is a chain, and since $(D, \sqsubseteq)$ is a ccpo we get that $\bigsqcup\emptyset$ exists. Using the definition of $\bigsqcup\emptyset$, we see that for any element d of D, we have $\bigsqcup\emptyset \sqsubseteq d$. This means that $\bigsqcup\emptyset$ is the least element of D. $\square$

Exercise 5.21 shows that **State** $\hookrightarrow$ **State** is not a complete lattice. Fortunately, we have the following lemma.

Lemma 5.25

(**State** $\hookrightarrow$ **State**, $\sqsubseteq$) is a ccpo. The least upper bound $\bigsqcup Y$ of a chain Y is given by

$$\mathrm{graph}(\bigsqcup Y) = \bigcup\{\ \mathrm{graph}(g) \mid g \in Y\ \}$$

that is, $(\bigsqcup Y)s = s'$ if and only if $g\ s = s'$ for some $g \in Y$.

Proof: The proof is in three parts. First we prove that

$$\bigcup\{\ \text{graph}(g) \mid g \in Y\ \} \tag{*}$$

is indeed a graph of a partial function in **State** $\hookrightarrow$ **State**. Second, we prove that this function will be an upper bound of Y. Thirdly, we prove that it is less than any other upper bound of Y; that is, it is the least upper bound of Y.

To verify that (*) specifies a *partial function*, we only need to show that if $\langle s, s'\rangle$ and $\langle s, s''\rangle$ are elements of

$$X = \bigcup\{\ \text{graph}(g) \mid g{\in}Y\ \}$$

then $s' = s''$. When $\langle s, s'\rangle \in X$, there will be a partial function $g \in Y$ such that $g\ s = s'$. Similarly, when $\langle s, s''\rangle \in X$, then there will be a partial function $g' \in Y$ such that $g'\ s = s''$. Since Y is a chain, we will have that either $g \sqsubseteq g'$ or $g' \sqsubseteq g$. In any case, we get $g\ s = g'\ s$, and this means that $s' = s''$ as required. This completes the first part of the proof.

In the second part of the proof, we define the partial function g_0 by

$$\text{graph}(g_0) = \bigcup\{\ \text{graph}(g) \mid g \in Y\ \}$$

To show that g_0 is an upper bound of Y, let g be an element of Y. Then we have $\text{graph}(g) \subseteq \text{graph}(g_0)$, and using the result of Exercise 5.8 we see that $g \sqsubseteq g_0$ as required and we have completed the second part of the proof.

In the third part of the proof, we show that g_0 is the least upper bound of Y. So let g_1 be some upper bound of Y. Using the definition of an upper bound, we get that $g \sqsubseteq g_1$ must hold for all $g \in Y$. Exercise 5.8 gives that $\text{graph}(g) \subseteq \text{graph}(g_1)$. Hence it must be the case that

$$\bigcup\{\ \text{graph}(g) \mid g \in Y\ \} \subseteq \text{graph}(g_1)$$

But this is the same as $\text{graph}(g_0) \subseteq \text{graph}(g_1)$, and Exercise 5.8 gives that $g_0 \sqsubseteq g_1$. This shows that g_0 is the least upper bound of Y and thereby we have completed the proof. ☐

Continuous Functions

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ satisfy the ccpo-property, and consider a (total) function $f\colon D \to D'$. If $d_1 \sqsubseteq d_2$, then the intuition is that d_1 shares its information with d_2. So when the function f has been applied to the two elements d_1 and d_2, we shall expect that a similar relationship holds between the results. That is, we shall expect that $f\ d_1 \sqsubseteq' f\ d_2$, and when this is the case we say that f is *monotone*. Formally, f is monotone if and only if

$$d_1 \sqsubseteq d_2 \text{ implies } f\ d_1 \sqsubseteq' f\ d_2$$

for all choices of d_1 and d_2.

Example 5.26

Consider $(\mathcal{P}(\{a,b,c\}), \subseteq)$ and $(\mathcal{P}(\{d,e\}), \subseteq)$. The function f_1: $\mathcal{P}(\{a,b,c\}) \to \mathcal{P}(\{d,e\})$ defined by the table

X	{a,b,c}	{a,b}	{a,c}	{b,c}	{a}	{b}	{c}	$\emptyset$
$f_1\ X$	{d,e}	{d}	{d,e}	{d,e}	{d}	{d}	{e}	$\emptyset$

is monotone: it simply changes a's and b's to d's and c's to e's.

The function f_2: $\mathcal{P}(\{a,b,c\}) \to \mathcal{P}(\{d,e\})$ defined by the table

X	{a,b,c}	{a,b}	{a,c}	{b,c}	{a}	{b}	{c}	$\emptyset$
$f_2\ X$	{d}	{d}	{d}	{e}	{d}	{e}	{e}	{e}

is *not* monotone because $\{b,c\} \subseteq \{a,b,c\}$ but $f_2\ \{b,c\} \not\subseteq f_2\ \{a,b,c\}$. Intuitively, all sets that contain an a are mapped to {d}, whereas the others are mapped to {e}, and since the elements {d} and {e} are incomparable this does not give a monotone function. However, if we change the definition such that sets with an a are mapped to {d} and all other sets to $\emptyset$, then the function will indeed be monotone. $\square$

Exercise 5.27

Consider the ccpo $(\mathcal{P}(\mathbf{N}), \subseteq)$. Determine which of the following functions in $\mathcal{P}(\mathbf{N}) \to \mathcal{P}(\mathbf{N})$ are monotone:

$$\begin{array}{lcl}
f_1\ X & = & \mathbf{N} \setminus X \\
f_2\ X & = & X \cup \{\mathbf{27}\} \\
f_3\ X & = & X \cap \{\mathbf{7}, \mathbf{9}, \mathbf{13}\} \\
f_4\ X & = & \{\ n \in X \mid n \text{ is a prime }\} \\
f_5\ X & = & \{\ \mathbf{2} \cdot n \mid n \in X\ \}
\end{array}$$

Exercise 5.28

Determine which of the following functionals of

$$(\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

are monotone:

$$\begin{array}{rcl}
F_0\ g & = & g \\
F_1\ g & = & \begin{cases} g_1 & \text{if } g = g_2 \\ g_2 & \text{otherwise} \end{cases} \qquad \text{where } g_1 \neq g_2 \\
(F'\ g)\ s & = & \begin{cases} g\ s & \text{if } s\ \mathtt{x} \neq \mathbf{0} \\ s & \text{if } s\ \mathtt{x} = \mathbf{0} \end{cases}
\end{array}$$

The monotone functions have a couple of interesting properties. First we prove that the composition of two monotone functions is a monotone function.

Fact 5.29

Let $(D, \sqsubseteq)$, $(D', \sqsubseteq')$, and $(D'', \sqsubseteq'')$ satisfy the ccpo-property, and let $f\colon D \to D'$ and $f'\colon D' \to D''$ be monotone functions. Then $f' \circ f\colon D \to D''$ is a monotone function.

Proof: Assume that $d_1 \sqsubseteq d_2$. The monotonicity of f gives that $f\ d_1 \sqsubseteq' f\ d_2$. The monotonicity of f' then gives $f'\ (f\ d_1) \sqsubseteq'' f'\ (f\ d_2)$ as required. □

Next we prove that the image of a chain under a monotone function is itself a chain.

Lemma 5.30

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ satisfy the ccpo-property, and let $f\colon D \to D'$ be a monotone function. If Y is a chain in D, then $\{ f\ d \mid d \in Y \}$ is a chain in D'. Furthermore,

$$\bigsqcup{}'\{ f\ d \mid d \in Y \} \sqsubseteq' f(\bigsqcup Y)$$

Proof: If $Y = \emptyset$, then the result holds immediately since $\bot' \sqsubseteq' f\ \bot$; so for the rest of the proof we may assume that $Y \neq \emptyset$. We shall first prove that $\{ f\ d \mid d \in Y \}$ is a chain in D'. So let d_1' and d_2' be two elements of $\{ f\ d \mid d \in Y \}$. Then there are elements d_1 and d_2 in Y such that $d_1' = f\ d_1$ and $d_2' = f\ d_2$. Since Y is a chain, we have that either $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$. In either case, we get that the same order holds between d_1' and d_2' because of the monotonicity of f. This proves that $\{ f\ d \mid d \in Y \}$ is a chain.

To prove the second part of the lemma, consider an arbitrary element d of Y. Then it is the case that $d \sqsubseteq \bigsqcup Y$. The monotonicity of f gives that $f\ d \sqsubseteq' f(\bigsqcup Y)$. Since this holds for all $d \in Y$, we get that $f(\bigsqcup Y)$ is an upper bound on $\{ f\ d \mid d \in Y \}$; that is, $\bigsqcup' \{ f\ d \mid d \in Y \} \sqsubseteq' f(\bigsqcup Y)$. □

In general, we cannot expect that a monotone function preserves least upper bounds on chains; that is, $\bigsqcup' \{ f\ d \mid d \in Y \} = f(\bigsqcup Y)$. This is illustrated by the following example.

Example 5.31

From Example 5.23, we get that $(\mathcal{P}(\mathbf{N} \cup \{\text{a}\}), \subseteq)$ is a ccpo. Now consider the function f: $\mathcal{P}(\mathbf{N} \cup \{\text{a}\}) \to \mathcal{P}(\mathbf{N} \cup \{\text{a}\})$ defined by

$$f\ X = \begin{cases} X & \text{if } X \text{ is finite} \\ X \cup \{\text{a}\} & \text{if } X \text{ is infinite} \end{cases}$$

Clearly, f is a monotone function: if $X_1 \subseteq X_2$, then also $f\ X_1 \subseteq f\ X_2$. However, f does not preserve the least upper bounds of chains. To see this, consider the set

$$Y = \{ \{0,1,\cdots,\text{n}\} \mid \text{n}{\geq}0 \}$$

It consists of the elements $\{0\}$, $\{0,1\}$, $\{0,1,2\}$, $\cdots$ and it is straightforward to verify that it is a chain with $\mathbf{N}$ as its least upper bound; that is, $\bigsqcup Y = \mathbf{N}$. When we apply f to the elements of Y, we get

$$\bigsqcup \{ f\ X \mid X \in Y \} = \bigsqcup Y = \mathbf{N}$$

However, we also have

$$f\ (\bigsqcup Y) = f\ \mathbf{N} = \mathbf{N} \cup \{\text{a}\}$$

showing that f does not preserve the least upper bounds of chains. □

We shall be interested in functions that preserve least upper bounds of chains; that is, functions f that satisfy

$$\bigsqcup' \{ f\ d \mid d \in Y \} = f(\bigsqcup Y)$$

Intuitively, this means that we obtain the same information independently of whether we determine the least upper bound before or after applying the function f.

We shall say that a function f: $D \to D'$ defined on $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ is *continuous* if it is monotone and

$$\bigsqcup' \{ f\ d \mid d \in Y \} = f(\bigsqcup Y)$$

holds for all *non-empty* chains Y. If $\bigsqcup \{ f\ d \mid d \in Y \} = f(\bigsqcup Y)$ holds for the empty chain (that is, $\bot = f\ \bot$), then we shall say that f is *strict*.

Example 5.32

The function f_1 of Example 5.26 is also continuous. To see this, consider a non-empty chain Y in $\mathcal{P}(\{\text{a,b,c}\})$. The least upper bound of Y will be the largest element, say X_0, of Y (see Example 5.17). Therefore we have

$$\begin{array}{rcll} f_1\ (\bigsqcup Y) & = & f_1\ X_0 & \text{because } X_0 = \bigsqcup Y \\ & \subseteq & \bigsqcup\{\ f_1\ X \mid X \in Y\ \} & \text{because } X_0 \in Y \end{array}$$

Using that f_1 is monotone, we get from Lemma 5.30 that $\bigsqcup\{\ f_1\ X \mid X \in Y\ \} \subseteq f_1\ (\bigsqcup Y)$. It follows that f_1 is continuous. Also, f_1 is a strict function because $f_1\ \emptyset = \emptyset$.

The function f of Example 5.31 is *not* a continuous function because there is a chain for which it does not preserve the least upper bound. □

Exercise 5.33

Show that the functional F' of Example 5.1 is continuous. □

Exercise 5.34

Assume that $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ satisfy the ccpo-property, and assume that the function f: $D \to D'$ satisfies

$$\bigsqcup{}'\{\ f\ d \mid d \in Y\ \} = f(\bigsqcup Y)$$

for all *non-empty* chains Y of D. Show that f is monotone. □

We can extend the result of Lemma 5.29 to show that the composition of two continuous functions will also be continuous, as follows.

Lemma 5.35

Let $(D, \sqsubseteq)$, $(D', \sqsubseteq')$, and $(D'', \sqsubseteq'')$ satisfy the ccpo-property, and let the functions f: $D \to D'$ and f': $D' \to D''$ be be continuous. Then $f' \circ f$: $D \to D''$ is a continuous function.

Proof: From Fact 5.29 we get that $f' \circ f$ is monotone. To prove that it is continuous, let Y be a non-empty chain in D. The continuity of f gives

$$\bigsqcup{}'\{\ f\ d \mid d \in Y\ \} = f\ (\bigsqcup Y)$$

Since $\{\ f\ d \mid d \in Y\ \}$ is a (non-empty) chain in D', we can use the continuity of f' and get

$$\bigsqcup''\{\, f'\ d' \mid d' \in \{\, f\ d \mid d \in Y \,\}\,\} = f'\ (\bigsqcup'\{\, f\ d \mid d \in Y \,\})$$

which is equivalent to

$$\bigsqcup''\{\, f'\ (f\ d) \mid d \in Y \,\} = f'\ (f\ (\bigsqcup Y))$$

This proves the result. □

Exercise 5.36

Prove that if f and f' are strict functions, then so is $f' \circ f$. □

We can now define the required fixed point operator FIX as follows.

Theorem 5.37

Let f: $D \to D$ be a continuous function on the ccpo $(D, \sqsubseteq)$ with least element $\perp$. Then

$$\mathsf{FIX}\ f = \bigsqcup\{\, f^{\rm n}\ \perp \mid {\rm n}{\geq}0 \,\}$$

defines an element of D, and this element is the least fixed point of f.

Here we have used the notation that $f^0 = \text{id}$, and $f^{\rm n+1} = f \circ f^{\rm n}$ for n≥0.

Proof: We first show the *well-definedness* of FIX f. Note that $f^0\ \perp = \perp$ and that $\perp \sqsubseteq d$ for all $d \in D$. By induction on n, one may show that

$$f^{\rm n}\ \perp \sqsubseteq f^{\rm n}\ d$$

for all $d \in D$ since f is monotone. It follows that $f^{\rm n}\ \perp \sqsubseteq f^{\rm m}\ \perp$ whenever n≤m. Hence $\{\, f^{\rm n}\ \perp \mid {\rm n}{\geq}0 \,\}$ is a (non-empty) chain in D, and FIX f exists because D is a ccpo.

We next show that FIX f is a *fixed point*; that is, f (FIX f) = FIX f. We calculate

$$\begin{array}{rcll}
f\ (\mathsf{FIX}\ f) & = & f\ (\bigsqcup\{\, f^{\rm n}\ \perp \mid {\rm n}{\geq}0 \,\}) & \text{(definition of } \mathsf{FIX}\ f\text{)} \\
 & = & \bigsqcup\{\, f(f^{\rm n}\ \perp) \mid {\rm n}{\geq}0 \,\} & \text{(continuity of } f\text{)} \\
 & = & \bigsqcup\{\, f^{\rm n}\ \perp \mid {\rm n}{\geq}1 \,\} & \\
 & = & \bigsqcup(\{\, f^{\rm n}\ \perp \mid {\rm n}{\geq}1 \,\} \cup \{\perp\}) & (\bigsqcup(Y \cup \{\perp\}) = \bigsqcup Y \\
 & & & \text{for all chains } Y) \\
 & = & \bigsqcup\{\, f^{\rm n}\ \perp \mid {\rm n}{\geq}0 \,\} & (f^0\ \perp = \perp) \\
 & = & \mathsf{FIX}\ f & \text{(definition of } \mathsf{FIX}\ f\text{)}
\end{array}$$

To see that FIX f is the *least* fixed point, assume that d is some other fixed point. Clearly $\perp \sqsubseteq d$ so the monotonicity of f gives $f^{\text{n}}\ \perp \sqsubseteq f^{\text{n}}\ d$ for n≥0, and as d was a fixed point, we obtain $f^{\text{n}}\ \perp \sqsubseteq d$ for all n≥0. Hence d is an upper bound of the chain $\{\ f^{\text{n}}\ \perp \mid \text{n}{\geq}0\ \}$, and using that FIX f is the least upper bound, we have FIX $f \sqsubseteq d$. □

Example 5.38

Consider the function F' of Example 5.1:

$$(F'\ g)\ s = \begin{cases} g\ s & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases}$$

We shall determine its least fixed point using the approach of Theorem 5.37. The least element $\perp$ of **State** $\hookrightarrow$ **State** is given by Lemma 5.13 and has $\perp\ s$ = $\underline{\text{undef}}$ for all s. We then determine the elements of the set $\{\ F'^{\text{n}}\ \perp \mid \text{n}{\geq}0\ \}$ as follows:

$$\begin{array}{rcll}
(F'^{0}\ \perp)\ s & = & (\text{id}\ \perp)\ s & \text{(definition of } F'^{0}\ \perp) \\
& = & \underline{\text{undef}} & \text{(definition of id and } \perp) \\
(F'^{1}\ \perp)\ s & = & (F'\ \perp)\ s & \text{(definition of } F'^{1}\ \perp) \\
& = & \begin{cases} \perp\ s & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases} & \text{(definition of } F'\ \perp) \\
& = & \begin{cases} \underline{\text{undef}} & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases} & \text{(definition of } \perp) \\
(F'^{2}\ \perp)\ s & = & F'\ (F'^{1}\ \perp)\ s & \text{(definition of } F'^{2}\ \perp) \\
& = & \begin{cases} (F'^{1}\ \perp)\ s & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases} & \text{(definition of } F') \\
& = & \begin{cases} \underline{\text{undef}} & \text{if } s\ \texttt{x} \neq \mathbf{0} \\ s & \text{if } s\ \texttt{x} = \mathbf{0} \end{cases} & \text{(definition of } F'^{1}\ \perp) \\
\vdots & & &
\end{array}$$

In general, we have $F'^{\text{n}}\ \perp = F'^{\text{n}+1}\ \perp$ for n > 0. Therefore

$$\bigsqcup\{\ F'^{\text{n}}\ \perp \mid \text{n}{\geq}0\ \} = \bigsqcup\ \{F'^{0}\ \perp,\ F'^{1}\ \perp\} = F'^{1}\ \perp$$

because $F'^{0}\ \perp = \perp$. Thus the least fixed point of F' will be the function

$$g_1\ s = \begin{cases} \underline{\text{undef}} & \text{if } s\ \mathtt{x} \neq \mathbf{0} \\ s & \text{if } s\ \mathtt{x} = \mathbf{0} \end{cases}$$

Exercise 5.39

Redo Exercise 5.15 using the approach of Theorem 5.37; that is, deduce the general form of the iterands, $F^n\ \perp$, for the functional, F, of Exercises 5.2 and 5.3. □

Exercise 5.40 (Essential)

Let f: $D \to D$ be a continuous function on a ccpo $(D, \sqsubseteq)$ and let $d \in D$ satisfy $f\ d \sqsubseteq d$. Show that $\mathsf{FIX}\ f \sqsubseteq d$. □

The table below summarizes the development we have performed in order to demonstrate the existence of least fixed points:

Fixed Point Theory	
1:	We restrict ourselves to *chain complete partially ordered sets* (abbreviated ccpo).
2:	We restrict ourselves to *continuous functions* on chain complete partially ordered sets.
3:	We show that continuous functions on chain complete partially ordered sets always have *least fixed points* (Theorem 5.37).

Exercise 5.41 (*)

Let $(D, \sqsubseteq)$ be a ccpo and define $(D \to D, \sqsubseteq')$ by setting

$$f_1 \sqsubseteq' f_2 \text{ if and only if } f_1\ d \sqsubseteq f_2\ d \text{ for all } d \in D$$

Show that $(D \to D, \sqsubseteq')$ is a ccpo and that FIX is "continuous" in the sense that

$$\mathsf{FIX}(\bigsqcup' \mathcal{F}) = \bigsqcup\{\ \mathsf{FIX}\ f \mid f \in \mathcal{F}\ \}$$

holds for all non-empty chains $\mathcal{F} \subseteq D \to D$ of continuous functions. □

Exercise 5.42 (** For Mathematicians)

Given a ccpo $(D, \sqsubseteq)$, we define an *open set* of D to be a subset Y of D satisfying

(1) if $d_1 \in Y$ and $d_1 \sqsubseteq d_2$ then $d_2 \in Y$, and

(2) if Y' is a non-empty chain satisfying $\bigsqcup Y' \in Y$, then there exists an element d of Y' that also is an element of Y.

The set of open sets of D is denoted $\mathcal{O}_D$. Show that this is indeed a *topology* on D; that is, show that

– $\emptyset$ and D are members of $\mathcal{O}_D$,

– the intersection of two open sets is an open set, and

– the union of any collection of open sets is an open set.

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ satisfy the ccpo-property. A function $f{:}D{\rightarrow}D'$ is *topologically continuous* if and only if the function f^{-1}: $\mathcal{P}(D') \rightarrow \mathcal{P}(D)$ defined by

$$f^{-1}(Y') = \{ \; d \in D \mid f \; d \in Y' \; \}$$

maps open sets to open sets; that is, specializes to f^{-1}: $\mathcal{O}_{D'} \rightarrow \mathcal{O}_D$. Show that f is a continuous function between D and D' if and only if it is a topologically continuous function between D and D'. $\square$

5.3 Direct Style Semantics: Existence

We have now obtained the mathematical foundations needed to prove that the semantic clauses of Table 5.1 do indeed define a function. So consider once again the clause

$$\mathcal{S}_{\text{ds}}[\![\texttt{while } b \texttt{ do } S]\!] = \mathsf{FIX}\ F$$
$$\text{where } F\ g = \mathsf{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}_{\text{ds}}[\![S]\!],\ \text{id})$$

For this to make sense, we must show that F is continuous. To do so, we first observe that

$$F\ g = F_1\ (F_2\ g)$$

where

$$F_1\ g = \mathsf{cond}(\mathcal{B}[\![b]\!],\ g,\ \text{id})$$

and

$$F_2\ g = g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!]$$

Using Lemma 5.35, we then obtain the continuity of F by showing that F_1 and F_2 are continuous. We shall first prove that F_1 is continuous:

Lemma 5.43

Let g_0: **State** $\hookrightarrow$ **State**, p: **State** $\to$ **T**, and define

$$F\ g = \mathsf{cond}(p,\ g,\ g_0)$$

Then F is continuous.

Proof: We shall first prove that F is *monotone*. So assume that $g_1 \sqsubseteq g_2$ and we shall show that $F\ g_1 \sqsubseteq F\ g_2$. It suffices to consider an arbitrary state s and show that

$$(F\ g_1)\ s = s' \text{ implies } (F\ g_2)\ s = s'$$

If $p\ s = \mathbf{tt}$, then $(F\ g_1)\ s = g_1\ s$, and from $g_1 \sqsubseteq g_2$ we get that $g_1\ s = s'$ implies $g_2\ s = s'$. Since $(F\ g_2)\ s = g_2\ s$, we have proved the result. So consider the case where $p\ s = \mathbf{ff}$. Then $(F\ g_1)\ s = g_0\ s$ and similarly $(F\ g_2)\ s = g_0\ s$ and the result is immediate.

To prove that F is *continuous*, we shall let Y be a non-empty chain in **State** $\hookrightarrow$ **State**. We must show that

$$F\ (\bigsqcup Y) \sqsubseteq \bigsqcup\{\ F\ g \mid g{\in}Y\ \}$$

since $F\ (\bigsqcup Y) \sqsupseteq \bigsqcup\{\ F\ g \mid g{\in}Y\ \}$ follows from the monotonicity of F (see Lemma 5.30). Thus we have to show that

$$\text{graph}(F(\bigsqcup Y)) \subseteq \bigcup\{\ \text{graph}(F\ g) \mid g{\in}Y\ \}$$

using the characterization of least upper bounds of chains in **State** $\hookrightarrow$ **State** given in Lemma 5.25. So assume that $(F\ (\bigsqcup Y))\ s = s'$ and let us determine $g \in Y$ such that $(F\ g)\ s = s'$. If $p\ s = \mathbf{ff}$, we have $F\ (\bigsqcup Y)\ s = g_0\ s = s'$ and clearly, for every element g of the non-empty set Y we have $(F\ g)\ s = g_0\ s = s'$. If $p\ s = \mathbf{tt}$, then we get $(F\ (\bigsqcup Y))\ s = (\bigsqcup Y)\ s = s'$ so $\langle s,\ s'\rangle \in \text{graph}(\bigsqcup Y)$. Since

$$\text{graph}(\bigsqcup Y) = \bigcup\{\ \text{graph}(g) \mid g{\in}Y\ \}$$

(according to Lemma 5.25), we therefore have $g{\in}Y$ such that $g\ s = s'$, and it follows that $(F\ g)\ s = s'$. This proves the result. □

Exercise 5.44 (Essential)

Prove that (in the setting of Lemma 5.43) F defined by $F\ g = \mathsf{cond}(p,\ g_0,\ g)$ is continuous; that is, cond is continuous in its second and third arguments. □

Lemma 5.45

Let g_0: **State** $\hookrightarrow$ **State**, and define

$$F\ g = g \circ g_0$$

Then F is continuous.

Proof: We first prove that F is monotone. If $g_1 \sqsubseteq g_2$, then graph(g_1) $\subseteq$ graph(g_2) according to Exercise 5.8, so that graph(g_0) $\diamond$ graph(g_1), which is the relational composition of graph(g_0) and graph(g_1) (see Appendix A), satisfies

$$\text{graph}(g_0) \diamond \text{graph}(g_1) \subseteq \text{graph}(g_0) \diamond \text{graph}(g_2)$$

and this shows that $F\ g_1 \sqsubseteq F\ g_2$. Next we shall prove that F is continuous. If Y is a non-empty chain, then

$$\begin{array}{rcl}
\text{graph}(F(\bigsqcup Y)) & = & \text{graph}((\bigsqcup Y) \circ g_0) \\
& = & \text{graph}(g_0) \diamond \text{graph}(\bigsqcup Y) \\
& = & \text{graph}(g_0) \diamond \bigcup\{\text{graph}(g) \mid g{\in}Y\} \\
& = & \bigcup\{\text{graph}(g_0) \diamond \text{graph}(g) \mid g{\in}Y\} \\
& = & \text{graph}(\bigsqcup\{F\ g \mid g{\in}Y\})
\end{array}$$

where we have used Lemma 5.25 twice. Thus $F\ (\bigsqcup Y) = \bigsqcup\{F\ g \mid g{\in}\mathrm{Y}\}$. □

Exercise 5.46 (Essential)

Prove that (in the setting of Lemma 5.45) F defined by $F\ g = g_0 \circ g$ is continuous; that is, $\circ$ is continuous in both arguments. □

We have now established the results needed to show that the equations of Table 5.1 define a function $\mathcal{S}_{\mathrm{ds}}$ as follows.

Proposition 5.47

The semantic equations of Table 5.1 define a total function $\mathcal{S}_{\mathrm{ds}}$ in **Stm** $\rightarrow$ (**State** $\hookrightarrow$ **State**).

Proof: The proof is by structural induction on the statement S.

The case $x := a$: Clearly the function that maps a state s to the state $s[x \mapsto \mathcal{A}[\![a]\!]s]$ is well-defined.

The case `skip`: Clearly the function id is well-defined.

The case $S_1;S_2$: The induction hypothesis gives that $\mathcal{S}_{\rm ds}[\![S_1]\!]$ and $\mathcal{S}_{\rm ds}[\![S_2]\!]$ are well-defined, and clearly their composition will be well-defined.

The case `if` b `then` S_1 `else` S_2: The induction hypothesis gives that $\mathcal{S}_{\rm ds}[\![S_1]\!]$ and $\mathcal{S}_{\rm ds}[\![S_2]\!]$ are well-defined functions, and clearly this property is preserved by the function cond.

The case `while` b `do` S: The induction hypothesis gives that $\mathcal{S}_{\rm ds}[\![S]\!]$ is well-defined. The functions F_1 and F_2 defined by

$$F_1\ g = \mathsf{cond}(\mathcal{B}[\![b]\!],\ g,\ \mathrm{id})$$

and

$$F_2\ g = g \circ \mathcal{S}_{\rm ds}[\![S]\!]$$

are continuous according to Lemmas 5.43 and 5.45. Thus Lemma 5.35 gives that $F\ g = F_1\ (F_2\ g)$ is continuous. From Theorem 5.37, we then have that FIX F is well-defined and thereby that $\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]$ is well-defined. This completes the proof. □

Example 5.48

Consider the denotational semantics of the factorial statement:

$$\mathcal{S}_{\rm ds}[\![\texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)}]\!]$$

We shall be interested in applying this function to a state s_0 where `x` has the value **3**. To do that, we shall first apply the clauses of Table 5.1, and we then get that

$$\begin{aligned}&\mathcal{S}_{\rm ds}[\![\texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y:=y}\star\texttt{x; x:=x}-\texttt{1)}]\!]\ s_0\\ &\qquad = (\mathsf{FIX}\ F)\ s_0[\texttt{y}\mapsto\mathbf{1}]\end{aligned}$$

where

$$F\ g\ s = \begin{cases} g\ (\mathcal{S}_{\rm ds}[\![\texttt{y:= y}\star\texttt{x; x:=x}-\texttt{1}]\!]\ s) & \text{if } \mathcal{B}[\![\neg(\texttt{x=1})]\!]\ s = \mathbf{tt}\\ s & \text{if } \mathcal{B}[\![\neg(\texttt{x=1})]\!]\ s = \mathbf{ff}\end{cases}$$

or, equivalently,

$$F\ g\ s = \begin{cases} g\ (s[\mathtt{y}{\mapsto}(s\ \mathtt{y}){\cdot}(s\ \mathtt{x})][\mathtt{x}{\mapsto}(s\ \mathtt{x}){-}\mathbf{1}]) & \text{if } s\ \mathtt{x} \neq \mathbf{1} \\ s & \text{if } s\ \mathtt{x} = \mathbf{1} \end{cases}$$

We can now calculate the various functions $F^{\mathrm{n}}\ \bot$ used in the definition of FIX F in Theorem 5.37:

$$\begin{aligned} (F^0\ \bot)\ s &= \underline{\text{undef}} \\ (F^1\ \bot)\ s &= \begin{cases} \underline{\text{undef}} & \text{if } s\ \mathtt{x} \neq \mathbf{1} \\ s & \text{if } s\ \mathtt{x} = \mathbf{1} \end{cases} \\ (F^2\ \bot)\ s &= \begin{cases} \underline{\text{undef}} & \text{if } s\ \mathtt{x} \neq \mathbf{1} \text{ and } s\ \mathtt{x} \neq \mathbf{2} \\ s[\mathtt{y}{\mapsto}(s\ \mathtt{y}){\cdot}\mathbf{2}][\mathtt{x}{\mapsto}\mathbf{1}] & \text{if } s\ \mathtt{x} = \mathbf{2} \\ s & \text{if } s\ \mathtt{x} = \mathbf{1} \end{cases} \end{aligned}$$

Thus, if `x` is **1** or **2**, then $F^2\ \bot$ will give the correct value for `y`, and for all other values of `x` the result is undefined. This is a general pattern: the n'th *iterand* $F^{\mathrm{n}}\ \bot$ will determine the correct value if it can be computed with *at most* n *unfoldings* of the `while`-loop (that is, n evaluations of the boolean condition). The general formula is

$$(F^{\mathrm{n}}\ \bot)\ s = \begin{cases} \underline{\text{undef}} & \text{if } s\ \mathtt{x} < \mathbf{1} \text{ or } s\ \mathtt{x} > \mathrm{n} \\ s[\mathtt{y}{\mapsto}(s\ \mathtt{y}){\cdot}j{\cdot}\ldots{\cdot}\mathbf{2}{\cdot}\mathbf{1}][\mathtt{x}{\mapsto}\mathbf{1}] & \text{if } s\ \mathtt{x} = j \text{ and } \mathbf{1}{\leq}j \text{ and } j{\leq}\mathrm{n} \end{cases}$$

We then have

$$(\mathsf{FIX}\ F)\ s = \begin{cases} \underline{\text{undef}} & \text{if } s\ \mathtt{x} < \mathbf{1} \\ s[\mathtt{y}{\mapsto}(s\ \mathtt{y}){\cdot}n{\cdot}\ldots{\cdot}\mathbf{2}{\cdot}\mathbf{1}][\mathtt{x}{\mapsto}\mathbf{1}] & \text{if } s\ \mathtt{x} = n \text{ and } n{\geq}\mathbf{1} \end{cases}$$

So in the state s_0 where `x` has the value **3**, we get that the value computed by the factorial statement is

$$(\mathsf{FIX}\ F)\ (s_0[\mathtt{y}{\mapsto}\mathbf{1}])\ \mathtt{y} = \mathbf{1}\cdot\mathbf{3}\cdot\mathbf{2}\cdot\mathbf{1} = \mathbf{6}$$

as expected. □

Exercise 5.49

Consider the statement

$$\mathtt{z{:=}0;\ while\ y{\leq}x\ do\ (z{:=}z{+}1;\ x{:=}x{-}y)}$$

and perform a development analogous to that of Example 5.48. □

Exercise 5.50

Show that $\mathcal{S}_{\mathrm{ds}}[\![\mathtt{while\ true\ do\ skip}]\!]$ is the totally undefined function $\bot$. □

Exercise 5.51

Extend the language with the statement `repeat` S `until` b and give the new (compositional) clause for $\mathcal{S}_{\text{ds}}$. Validate the well-definedness of the extended version of $\mathcal{S}_{\text{ds}}$. □

Exercise 5.52

Extend the language with the statement `for` x := a_1 `to` a_2 `do` S and give the new (compositional) clause for $\mathcal{S}_{\text{ds}}$. Validate the well-definedness of the extended version of $\mathcal{S}_{\text{ds}}$. □

To summarize, the well-definedness of $\mathcal{S}_{\text{ds}}$ relies on the following results established above:

Proof Summary for While:

Well-definedness of Denotational Semantics

1: The set **State** $\hookrightarrow$ **State** equipped with an appropriate order $\sqsubseteq$ is a ccpo (Lemmas 5.13 and 5.25).

2: Certain functions Ψ: (**State** $\hookrightarrow$ **State**) $\rightarrow$ (**State** $\hookrightarrow$ **State**) are continuous (Lemmas 5.43 and 5.45).

3: In the definition of $\mathcal{S}_{\text{ds}}$, we only apply the fixed point operation to continuous functions (Proposition 5.47).

Properties of the Semantics

In the operational semantics, we defined a notion of two statements being semantically equivalent. A similar notion can be defined based on the denotational semantics: S_1 and S_2 are *semantically equivalent* if and only if

$$\mathcal{S}_{\text{ds}}[\![S_1]\!] = \mathcal{S}_{\text{ds}}[\![S_2]\!]$$

Exercise 5.53

Show that the following statements of **While** are semantically equivalent in the sense above:

– S;`skip` and S

– S_1;(S_2;S_3) and (S_1;S_2);S_3

– `while` b `do` S and `if` b `then` (S; `while` b `do` S) `else skip` □

Exercise 5.54 (*)

Prove that `repeat` S `until` b and S; `while` $\neg b$ `do` S are semantically equivalent using the denotational approach. The semantics of the `repeat`-construct is given in Exercise 5.51. □

5.4 An Equivalence Result

Having produced yet another semantics of the language **While**, we shall be interested in its relation to the operational semantics, and for this we shall focus on the structural operational semantics given in Section 2.2.

Theorem 5.55

For every statement S of **While**, we have $\mathcal{S}_{\rm sos}[\![S]\!] = \mathcal{S}_{\rm ds}[\![S]\!]$.

Both $\mathcal{S}_{\rm ds}[\![S]\!]$ and $\mathcal{S}_{\rm sos}[\![S]\!]$ are functions in **State** $\hookrightarrow$ **State**; that is, they are elements of a partially ordered set. To prove that two elements d_1 and d_2 of a partially ordered set are equal, it is sufficient to prove that $d_1 \sqsubseteq d_2$ and that $d_2 \sqsubseteq d_1$. Thus, to prove Theorem 5.55, we shall show that

– $\mathcal{S}_{\rm sos}[\![S]\!] \sqsubseteq \mathcal{S}_{\rm ds}[\![S]\!]$ and

– $\mathcal{S}_{\rm ds}[\![S]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S]\!]$.

The first result is expressed by the following lemma.

Lemma 5.56

For every statement S of **While**, we have $\mathcal{S}_{\rm sos}[\![S]\!] \sqsubseteq \mathcal{S}_{\rm ds}[\![S]\!]$.

Proof: It is sufficient to prove that for all states s and s'

$$\langle S,\, s\rangle \Rightarrow^* s' \text{ implies } \mathcal{S}_{\rm ds}[\![S]\!]s = s' \qquad (*)$$

To do so, we shall need to establish the following property

$$\begin{array}{lll}\langle S,\, s\rangle \Rightarrow s' & \text{implies} & \mathcal{S}_{\rm ds}[\![S]\!]s = s' \\ \langle S,\, s\rangle \Rightarrow \langle S',\, s'\rangle & \text{implies} & \mathcal{S}_{\rm ds}[\![S]\!]s = \mathcal{S}_{\rm ds}[\![S']\!]s'\end{array} \qquad (**)$$

Assuming that (**) holds, the proof of (*) is a straightforward induction on the length k of the derivation sequence $\langle S, s\rangle \Rightarrow^{\mathrm{k}} s'$ (see Section 2.2).

We now turn to the proof of (**), and for this we shall use induction on the shape of the derivation tree for $\langle S, s\rangle \Rightarrow s'$ or $\langle S, s\rangle \Rightarrow \langle S', s'\rangle$.

The case [ass$_{\mathrm{sos}}$]: We have

$$\langle x := a,\ s\rangle \Rightarrow s[x\mapsto\mathcal{A}[\![a]\!]s]$$

and since $\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s = s[x\mapsto\mathcal{A}[\![a]\!]s]$, the result follows.

The case [skip$_{\mathrm{sos}}$]: Analogous.

The case [comp$^{1}_{\mathrm{sos}}$]: Assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle$$

because $\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle$. Then the induction hypothesis applied to the latter transition gives $\mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = \mathcal{S}_{\mathrm{ds}}[\![S_1']\!]s'$ and we get

$$\begin{array}{rcl}\mathcal{S}_{\mathrm{ds}}[\![S_1;S_2]\!]\ s & = & \mathcal{S}_{\mathrm{ds}}[\![S_2]\!](\mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s)\\ & = & \mathcal{S}_{\mathrm{ds}}[\![S_2]\!](\mathcal{S}_{\mathrm{ds}}[\![S_1']\!]s')\\ & = & \mathcal{S}_{\mathrm{ds}}[\![S_1';S_2]\!]s'\end{array}$$

as required.

The case [comp$^{2}_{\mathrm{sos}}$]: Assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle$$

because $\langle S_1, s\rangle\Rightarrow s'$. Then the induction hypothesis applied to that transition gives $\mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s = s'$ and we get

$$\begin{array}{rcl}\mathcal{S}_{\mathrm{ds}}[\![S_1;S_2]\!]s & = & \mathcal{S}_{\mathrm{ds}}[\![S_2]\!](\mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s)\\ & = & \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]s'\end{array}$$

where the first equality comes from the definition of $\mathcal{S}_{\mathrm{ds}}$ and we just argued for the second equality. This proves the result.

The case [if$^{\mathrm{tt}}_{\mathrm{sos}}$]: Assume that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$$

because $\mathcal{B}[\![b]\!]\ s = \mathbf{tt}$. Then

$$\begin{array}{rcl}\mathcal{S}_{\mathrm{ds}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]s & = & \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!],\ \mathcal{S}_{\mathrm{ds}}[\![S_2]\!])s\\ & = & \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]s\end{array}$$

as required.

The case [if$^{\mathrm{ff}}_{\mathrm{sos}}$]: Analogous.

The case [while$_{\mathrm{sos}}$]: Assume that

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle$$

From the definition of $\mathcal{S}_{\rm ds}$, we have $\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \mathsf{FIX}\ F$, where $F\ g = \mathsf{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}_{\rm ds}[\![S]\!],\ \text{id})$. We therefore get

$$\begin{aligned}
\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] &= (\mathsf{FIX}\ F)\\
&= F\ (\mathsf{FIX}\ F)\\
&= \mathsf{cond}(\mathcal{B}[\![b]\!],\ (\mathsf{FIX}\ F) \circ \mathcal{S}_{\rm ds}[\![S]\!],\ \text{id})\\
&= \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}_{\rm ds}[\![S]\!],\ \text{id})\\
&= \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\rm ds}[\![S;\ \texttt{while}\ b\ \texttt{do}\ S]\!],\ \mathcal{S}_{\rm ds}[\![\texttt{skip}]\!])\\
&= \mathcal{S}_{\rm ds}[\![\texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}]\!]
\end{aligned}$$

as required. This completes the proof of (**). □

Note that (*) does *not* imply that $\mathcal{S}_{\rm sos}[\![S]\!] = \mathcal{S}_{\rm ds}[\![S]\!]$, as we have only proved that *if* $\mathcal{S}_{\rm sos}[\![S]\!]s \neq \underline{\text{undef}}$, *then* $\mathcal{S}_{\rm sos}[\![S]\!]s = \mathcal{S}_{\rm ds}[\![S]\!]s$. Still, there is the possibility that $\mathcal{S}_{\rm ds}[\![S]\!]$ may be defined for more arguments than $\mathcal{S}_{\rm sos}[\![S]\!]$. However, this is ruled out by the following lemma.

Lemma 5.57

For every statement S of **While**, we have $\mathcal{S}_{\rm ds}[\![S]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S]\!]$.

Proof: We proceed by structural induction on the statement S.

The case $x := a$: Clearly $\mathcal{S}_{\rm ds}[\![x := a]\!]s = \mathcal{S}_{\rm sos}[\![x := a]\!]s$. Note that this means that $\mathcal{S}_{\rm sos}$ satisfies the clause defining $\mathcal{S}_{\rm ds}$ in Table 5.1.

The case `skip`: Clearly $\mathcal{S}_{\rm ds}[\![\texttt{skip}]\!]s = \mathcal{S}_{\rm sos}[\![\texttt{skip}]\!]s$.

The case S_1 ; S_2: Recall that $\circ$ is monotone in both arguments (Lemma 5.45 and Exercise 5.46). We then have

$$\begin{aligned}
\mathcal{S}_{\rm ds}[\![S_1\ ;\ S_2]\!] &= \mathcal{S}_{\rm ds}[\![S_2]\!] \circ \mathcal{S}_{\rm ds}[\![S_1]\!]\\
&\sqsubseteq \mathcal{S}_{\rm sos}[\![S_2]\!] \circ \mathcal{S}_{\rm sos}[\![S_1]\!]
\end{aligned}$$

because the induction hypothesis applied to S_1 and S_2 gives $\mathcal{S}_{\rm ds}[\![S_1]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S_1]\!]$ and $\mathcal{S}_{\rm ds}[\![S_2]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S_2]\!]$. Furthermore, Exercise 2.21 gives that if $\langle S_1,\ s\rangle \Rightarrow^* s'$ then $\langle S_1\ ;\ S_2,\ s\rangle \Rightarrow^* \langle S_2,\ s'\rangle$ and hence

$$\mathcal{S}_{\rm sos}[\![S_2]\!] \circ \mathcal{S}_{\rm sos}[\![S_1]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S_1\ ;\ S_2]\!]$$

and this proves the result. Note that in this case $\mathcal{S}_{\rm sos}$ fulfils a weaker version of the clause defining $\mathcal{S}_{\rm ds}$ in Table 5.1.

The case `if` b `then` S_1 `else` S_2: Recall that cond is monotone in its second and third arguments (Lemma 5.43 and Exercise 5.44). We then have

$$\begin{array}{rcl}\mathcal{S}_{\rm ds}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm ds}[\![S_1]\!], \mathcal{S}_{\rm ds}[\![S_2]\!]) \\ & \sqsubseteq & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm sos}[\![S_1]\!], \mathcal{S}_{\rm sos}[\![S_2]\!])\end{array}$$

because the induction hypothesis applied to S_1 and S_2 gives $\mathcal{S}_{\rm ds}[\![S_1]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S_1]\!]$ and $\mathcal{S}_{\rm ds}[\![S_2]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S_2]\!]$. Furthermore, it follows from [$\text{if}^{\rm tt}_{\rm sos}$] and [$\text{if}^{\rm ff}_{\rm sos}$] that

$$\begin{array}{rcll}\mathcal{S}_{\rm sos}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s & = & \mathcal{S}_{\rm sos}[\![S_1]\!]s & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \\ \mathcal{S}_{\rm sos}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s & = & \mathcal{S}_{\rm sos}[\![S_2]\!]s & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}\end{array}$$

so that

$$\mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm sos}[\![S_1]\!], \mathcal{S}_{\rm sos}[\![S_2]\!]) = \mathcal{S}_{\rm sos}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]$$

and this proves the result. Note that in this case $\mathcal{S}_{\rm sos}$ fulfils the clause defining $\mathcal{S}_{\rm ds}$ in Table 5.1.

The case `while` b `do` S: We have

$$\mathcal{S}_{\rm ds}[\![\texttt{while } b \texttt{ do } S]\!] = \mathsf{FIX}\ F$$

where $F\ g = \mathsf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\rm ds}[\![S]\!], \text{id})$, and we recall that F is continuous. It is sufficient to prove that

$$F(\mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!]) \sqsubseteq \mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!]$$

because then Exercise 5.40 gives $\mathsf{FIX}\ F \sqsubseteq \mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!]$ as required. From Exercise 2.21, we get

$$\begin{array}{rcl}\mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!] & = & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm sos}[\![S \texttt{ ; while } b \texttt{ do } S]\!], \text{id}) \\ & \sqsupseteq & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!] \circ \mathcal{S}_{\rm sos}[\![S]\!], \text{id})\end{array}$$

The induction hypothesis applied to S gives $\mathcal{S}_{\rm ds}[\![S]\!] \sqsubseteq \mathcal{S}_{\rm sos}[\![S]\!]$, so using the monotonicity of $\circ$ and cond, we get

$$\begin{array}{rcl}\mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!] & \sqsupseteq & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!] \circ \mathcal{S}_{\rm sos}[\![S]\!], \text{id}) \\ & \sqsupseteq & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!] \circ \mathcal{S}_{\rm ds}[\![S]\!], \text{id}) \\ & = & F(\mathcal{S}_{\rm sos}[\![\texttt{while } b \texttt{ do } S]\!])\end{array}$$

Note that in this case $\mathcal{S}_{\rm sos}$ also fulfils a weaker version of the clause defining $\mathcal{S}_{\rm ds}$ in Table 5.1. □

The key technique used in the proof can be summarized as follows:

Proof Summary for While:

Equivalence of Operational and Denotational Semantics

1: Prove that $\mathcal{S}_{\text{sos}}[\![S]\!] \sqsubseteq \mathcal{S}_{\text{ds}}[\![S]\!]$ by first using *induction on the shape of derivation trees* to show that

– if a statement is executed *one step* in the structural operational semantics and does not terminate, then this does not change the meaning in the denotational semantics, and

– if a statement is executed *one step* in the structural operational semantics and does terminate, then the same result is obtained in the denotational semantics

and secondly by using *induction on the length of derivation sequences.*

2: Prove that $\mathcal{S}_{\text{ds}}[\![S]\!] \sqsubseteq \mathcal{S}_{\text{sos}}[\![S]\!]$ by showing that

– $\mathcal{S}_{\text{sos}}$ fulfils slightly weaker versions of the clauses defining $\mathcal{S}_{\text{ds}}$ in Table 5.1, that is, if

$$\mathcal{S}_{\text{ds}}[\![S]\!] = \Psi(\cdots\, \mathcal{S}_{\text{ds}}[\![S']\!]\, \cdots)$$

then $\mathcal{S}_{\text{sos}}[\![S]\!] \sqsupseteq \Psi(\cdots\, \mathcal{S}_{\text{sos}}[\![S']\!]\, \cdots)$

A proof by *structural induction* then gives that $\mathcal{S}_{\text{ds}}[\![S]\!] \sqsubseteq \mathcal{S}_{\text{sos}}[\![S]\!]$.

Exercise 5.58

Give a detailed argument showing that

$$\mathcal{S}_{\text{sos}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \sqsupseteq \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{sos}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] \circ \mathcal{S}_{\text{sos}}[\![S]\!], \text{id}).$$

Exercise 5.59

Extend the proof of Theorem 5.55 so that it applies to the language when augmented with `repeat` S `until` b. □

Exercise 5.60

Extend the proof of Theorem 5.55 so that it applies to the language when augmented with `for` x:=a_1 `to` a_2 `do` S. □

Exercise 5.61

Combining the results of Theorem 2.26 and Theorem 5.55, we get that $\mathcal{S}_{\text{ns}}[\![S]\!] = \mathcal{S}_{\text{ds}}[\![S]\!]$ holds for every statement S of **While**. Give a direct proof of this (that is, without using the two theorems). $\square$

6
More on Denotational Semantics

The previous chapter presented the foundations for denotational semantics, and we shall now study two extensions of the language **While**: first with blocks and procedures, much as in Chapter 3, and subsequently with exceptions. In doing so, we introduce some important semantic concepts:

– environments and stores and

– continuations.

For both language extensions, the semantic specifications are obtained by suitable modifications of Table 5.1. Although we shall not go into great detail about the foundational properties of the specifications in the present chapter, it is important to note that the fixed point theory of chain complete partially ordered sets and continous functions presented in the previous chapter also applies for the development performed here.

6.1 Environments and Stores

We shall first extend **While** with blocks declaring local variables and procedures. The new language is called **Proc**, and its syntax is

$$\begin{array}{lcl} S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\ & \mid & \texttt{while } b \texttt{ do } S \mid \texttt{begin } D_V \; D_P \; S \texttt{ end} \mid \texttt{call } p \\ D_V & ::= & \texttt{var } x := a;\, D_V \mid \varepsilon \\ D_P & ::= & \texttt{proc } p \texttt{ is } S;\, D_P \mid \varepsilon \end{array}$$

where D_V and D_P are meta-variables ranging over the syntactic categories $\mathbf{Dec}_\mathrm{V}$ of variable declarations and $\mathbf{Dec}_\mathrm{P}$ of procedure declarations, respectively, and p is a meta-variable ranging over the syntactic category **Pname** of procedure names. The idea is that variables and procedures are only known inside the block where they are declared. Procedures may or may not be recursive, and we shall emphasize the differences in the semantics to be specified below.

We shall adopt *static scope rules* rather than dynamic scope rules. Consider the following statement:

```
begin var x := 7; proc p is x := 0;
      begin var x := 5; call p end
end
```

Using static scope rules, the effect of executing `call p` in the inner block will be to modify the *global* variable `x`. Using dynamic scope rules, the effect will be to modify the *local* variable `x`.

To obtain static scope rules, we shall follow the approach of Section 3.2 and introduce the notion of *locations*: to each variable, we associate a unique location, and to each location we associate a value. This is in contrast to what we did in Table 5.1, where we employed a direct association between variables and values. The idea then is that whenever a new variable is declared it is associated with a new unused location and that it is the value of this location that is changed by assignment to the variable. With respect to the statement above, this means that the global variable `x` and the local variable `x` will have different locations. In the inner block, we can only directly access the location of the local variable, but the procedure body for `p` may only access the location of the global variable.

Stores and Variable Environments

So far, states in **State** have been used to associate values with variables. We shall now replace states with *stores* that map locations to values and with *variable environments* that map variables to locations. We introduce the domain

$$\mathbf{Loc} = \mathbf{Z}$$

of locations, which for the sake of simplicity has been identified with the integers. We shall need an operation

$$\text{new: } \mathbf{Loc} \to \mathbf{Loc}$$

on locations that given a location will give the next one; since **Loc** is **Z**, we may take 'new' to be the successor function on the integers.

We can now define a store, *sto*, as an element of

$$\mathbf{Store} = \mathbf{Loc} \cup \{\mathsf{next}\} \to \mathbf{Z}$$

where next is a special token used to hold the *next free location.* Note that since **Loc** is **Z** we have that *sto* next is a location.

A variable environment env_V is an element of

$$\mathbf{Env_V} = \mathbf{Var} \to \mathbf{Loc}$$

Thus the variable environment will assign a location to each variable.

So, rather than having a single mapping s from variables to values, we have split it into two mappings, env_V and *sto*, and the idea is that $s = sto \circ env_V$. This motivates defining the function 'lookup' by

$$\text{lookup } env_V \; sto = sto \circ env_V$$

so that 'lookup env_V' will transform a store to a state; that is,

$$\text{lookup: } \mathbf{Env_V} \to \mathbf{Store} \to \mathbf{State}$$

Having replaced a one-stage mapping with a two-stage mapping, we shall want to reformulate the semantic equations of Table 5.1 to use variable environments and stores. The new semantic function $\mathcal{S}'_{\text{ds}}$ has functionality

$$\mathcal{S}'_{\text{ds}}\text{: } \mathbf{Stm} \to \mathbf{Env_V} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

so that only the store is updated during the execution of statements. The clauses defining $\mathcal{S}'_{\text{ds}}$ are given in Table 6.1. Note that in the clause for assignment, the variable environment is consulted to determine the location of the variable, and this location is updated in the store. In the clauses for the conditional and the `while`-construct, we use the auxiliary function cond of functionality

$$\begin{aligned}\mathsf{cond}\text{: } &(\mathbf{Store} \to \mathbf{T}) \times (\mathbf{Store} \hookrightarrow \mathbf{Store}) \times (\mathbf{Store} \hookrightarrow \mathbf{Store}) \\ &\qquad \to (\mathbf{Store} \hookrightarrow \mathbf{Store})\end{aligned}$$

and its definition is as in Section 5.1.

Exercise 6.1

We have to make sure that the clauses of Table 6.1 define a well-defined function $\mathcal{S}'_{\text{ds}}$. To do so,

$$
\begin{aligned}
\mathcal{S}'_{\mathrm{ds}}[\![x{:=}a]\!]\mathit{env}_V\ \mathit{sto} &= \mathit{sto}[l{\mapsto}\mathcal{A}[\![a]\!](\mathrm{lookup}\ \mathit{env}_V\ \mathit{sto})] \\
&\quad \text{where } l = \mathit{env}_V\ x \\
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{skip}]\!]\mathit{env}_V &= \mathrm{id} \\
\mathcal{S}'_{\mathrm{ds}}[\![S_1\ ;\ S_2]\!]\mathit{env}_V &= (\mathcal{S}'_{\mathrm{ds}}[\![S_2]\!]\mathit{env}_V) \circ (\mathcal{S}'_{\mathrm{ds}}[\![S_1]\!]\mathit{env}_V) \\
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\mathit{env}_V & \\
&= \mathsf{cond}(\mathcal{B}[\![b]\!]{\circ}(\mathrm{lookup}\ \mathit{env}_V),\ \mathcal{S}'_{\mathrm{ds}}[\![S_1]\!]\mathit{env}_V,\ \mathcal{S}'_{\mathrm{ds}}[\![S_2]\!]\mathit{env}_V) \\
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\mathit{env}_V &= \mathsf{FIX}\ F \\
&\text{where } F\ g = \mathsf{cond}(\mathcal{B}[\![b]\!]{\circ}(\mathrm{lookup}\ \mathit{env}_V),\ g \circ (\mathcal{S}'_{\mathrm{ds}}[\![S]\!]\mathit{env}_V),\ \mathrm{id})
\end{aligned}
$$

Table 6.1 Denotational semantics for **While** using locations

– equip **Store** $\hookrightarrow$ **Store** with a partial ordering such that it becomes a ccpo,

– show that $\circ$ is continuous in both of its arguments and that cond is continuous in its second and third arguments, and

– show that the fixed point operation is only applied to continuous functions.

Conclude that $\mathcal{S}'_{\mathrm{ds}}$ is a well-defined function. □

Exercise 6.2 (*)

Prove that the two semantic functions $\mathcal{S}_{\mathrm{ds}}$ and $\mathcal{S}'_{\mathrm{ds}}$ satisfy

$$\mathcal{S}_{\mathrm{ds}}[\![S]\!] \circ (\mathrm{lookup}\ \mathit{env}_V) = (\mathrm{lookup}\ \mathit{env}_V) \circ (\mathcal{S}'_{\mathrm{ds}}[\![S]\!]\mathit{env}_V)$$

for all statements S of **While** and for all env_V such that env_V is an injective mapping. □

Exercise 6.3

Having replaced a one-stage mapping with a two-stage mapping, we might consider redefining the semantic functions $\mathcal{A}$ and $\mathcal{B}$. The new functionalities of $\mathcal{A}$ and $\mathcal{B}$ might be

$$\mathcal{A}'\colon \mathbf{Aexp} \to \mathbf{Env}_{\mathrm{V}} \to (\mathbf{Store} \to \mathbf{Z})$$

$$\mathcal{B}'\colon \mathbf{Bexp} \to \mathbf{Env}_{\mathrm{V}} \to (\mathbf{Store} \to \mathbf{T})$$

and the intended relationship is that

$$\begin{aligned}
&\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![\texttt{var}\ x := a;\ D_V]\!](env_V,\ sto) = \\
&\qquad \mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![D_V]\!](env_V[x\mapsto l],\ sto[l\mapsto v][\mathsf{next}\mapsto \text{new}\ l]) \\
&\qquad \text{where } l = sto\ \mathsf{next} \text{ and } v = \mathcal{A}[\![a]\!](\text{lookup}\ env_V\ sto) \\
&\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![\varepsilon]\!] = \text{id}
\end{aligned}$$

Table 6.2 Denotational semantics for variable declarations

$$\mathcal{A}'[\![a]\!]env_V = \mathcal{A}[\![a]\!] \circ (\text{lookup}\ env_V)$$
$$\mathcal{B}'[\![b]\!]env_V = \mathcal{B}[\![b]\!] \circ (\text{lookup}\ env_V)$$

Give a compositional definition of the functions $\mathcal{A}'$ and $\mathcal{B}'$ such that this is the case. □

Updating the Variable Environment

The variable environment is updated whenever we enter a block containing local declarations. To express this, we shall introduce a semantic function $\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}$ for the syntactic category of variable declarations. It has functionality

$$\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}\text{: } \mathbf{Dec}_{\mathrm{V}} \to (\mathbf{Env}_V \times \mathbf{Store}) \to (\mathbf{Env}_V \times \mathbf{Store})$$

The function $\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![D_V]\!]$ will take a pair as arguments: the first component of that pair will be the current variable environment and the second component the current store. The function will return the updated variable environment as well as the updated store. The function is defined by the semantic clauses of Table 6.2. Note that we process the declarations from left to right and that we update the value of the token next in the store.

In the case where there are *no* procedure declarations in a block, we can extend the semantic function $\mathcal{S}'_{\mathrm{ds}}$ of Table 6.1 with a clause such as

$$\begin{aligned}
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{begin}\ D_V\ S\ \texttt{end}]\!]env_V\ sto &= \mathcal{S}'_{\mathrm{ds}}[\![S]\!]env'_V\ sto' \\
&\text{where } \mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![D_V]\!](env_V,\ sto) = (env'_V,\ sto')
\end{aligned}$$

Thus we evaluate the body S in an updated variable environment and an updated store. We shall later modify the clause above to take the procedure declarations into account.

Exercise 6.4

Consider the following statement of **Proc**:

```
begin var y := 0; var x := 1;

      begin ¯var x := 7; x := x+1 end;

      y := x

end
```

Use $\mathcal{S}'_{\mathrm{ds}}$ and $\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}$ to show that the location for y is assigned the value **1** in the final store. □

Procedure Environments

To cater to procedures, we shall introduce the notion of a *procedure environment*. It will be a total function that will associate each procedure with the effect of executing its body. This means that a procedure environment, env_P, will be an element of

$$\mathbf{Env}_{\mathrm{P}} = \mathbf{Pname} \rightarrow (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

Remark This notion of procedure environment differs from that of the operational approach studied in Section 3.2. □

The procedure environment is updated using the semantic function $\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}$ for procedure declarations. It has functionality

$$\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}\colon \mathbf{Dec}_{\mathrm{P}} \rightarrow \mathbf{Env}_{\mathrm{V}} \rightarrow \mathbf{Env}_{\mathrm{P}} \rightarrow \mathbf{Env}_{\mathrm{P}}$$

So given the current variable environment and the current procedure environment, the function $\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}[\![D_P]\!]$ will return the updated procedure environment. The variable environment must be available because procedures must know the variables that have been declared so far. An example is the statement

```
begin   var x := 7; proc p is x := 0;

        begin var x := 5; call p end

end
```

where the body of p must know that a variable x has been declared in the outer block.

The semantic clauses defining $\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}$ in the case of *non-recursive procedures* are given in Table 6.3. In the clause for procedure declarations, we use the semantic function $\mathcal{S}_{\mathrm{ds}}$ for statements (defined below) to determine the meaning of the body of the procedure using that env_V and env_P are the environments at the point of declaration. The variables occurring in the body S of p will therefore be bound to the locations of the variables as known at the time of declaration, but the values of these locations will not be known until the time

$$\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}[\![\texttt{proc}\ p\ \texttt{is}\ S;\ D_P]\!]\,env_V\ env_P = \mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}[\![D_P]\!]\,env_V\ (env_P[p{\mapsto}g])$$
$$\text{where } g = \mathcal{S}_{\mathrm{ds}}[\![S]\!]\,env_V\ env_P$$
$$\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}[\![\varepsilon]\!]\,env_V = \mathrm{id}$$

Table 6.3 Denotational semantics for non-recursive procedure declarations

of the call. In this way, we ensure that we obtain static scope for variables. Also, an occurrence of `call` p' in the body of the procedure will refer to a procedure p' mentioned in env_P; that is, a procedure declared in an outer block or in the current block but preceding the present procedure. In this way, we obtain static scope for procedures. This will be illustrated in Exercise 6.6 below.

The Semantic Function $\mathcal{S}_{\mathrm{ds}}$ for Proc

The meaning of a statement depends on the variables and procedures that have been declared. Therefore the semantic function $\mathcal{S}_{\mathrm{ds}}$ for statements in **Proc** will have functionality

$$\mathcal{S}_{\mathrm{ds}}\colon \mathbf{Stm} \to \mathbf{Env}_{\mathrm{V}} \to \mathbf{Env}_{\mathrm{P}} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

The function is defined by the clauses of Table 6.4. In most cases, the definition of $\mathcal{S}_{\mathrm{ds}}$ is a straightforward modification of the clauses of $\mathcal{S}'_{\mathrm{ds}}$. Note that the meaning of a procedure call is obtained by simply consulting the procedure environment.

Example 6.5

This example shows how we obtain static scope rules for the variables. Consider the application of the semantic function $\mathcal{S}_{\mathrm{ds}}$ to the statement

```
begin var x := 7; proc p is x := 0;
      begin var x := 5; call p end
end
```

Assume that the initial environments are env_V and env_P and that the initial store sto has sto next $= \mathbf{12}$. Then the first step will be to update the variable environment with the declarations of the outer block:

$$\begin{aligned}\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![\texttt{var x := 7;}]\!](env_V,\ sto) &= \mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}[\![\varepsilon]\!](env_V[\mathsf{x}{\mapsto}\mathbf{12}],\ sto[\mathbf{12}{\mapsto}\mathbf{7}][\mathsf{next}{\mapsto}\mathbf{13}])\\ &= (env_V[\mathsf{x}{\mapsto}\mathbf{12}],\ sto[\mathbf{12}{\mapsto}\mathbf{7}][\mathsf{next}{\mapsto}\mathbf{13}])\end{aligned}$$

$$\mathcal{S}_{\rm ds}[\![x{:=}a]\!]\,env_V\ env_P\ sto = sto[l{\mapsto}\mathcal{A}[\![a]\!](\text{lookup}\ env_V\ sto)]$$
$$\text{where } l = env_V\ x$$

$$\mathcal{S}_{\rm ds}[\![\texttt{skip}]\!]\,env_V\ env_P = \text{id}$$

$$\mathcal{S}_{\rm ds}[\![S_1\ ;\ S_2]\!]\,env_V\ env_P = (\mathcal{S}_{\rm ds}[\![S_2]\!]\,env_V\ env_P) \circ (\mathcal{S}_{\rm ds}[\![S_1]\!]\,env_V\ env_P)$$

$$\mathcal{S}_{\rm ds}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\,env_V\ env_P =$$
$$\textsf{cond}(\mathcal{B}[\![b]\!]{\circ}(\text{lookup}\ env_V),\ \mathcal{S}_{\rm ds}[\![S_1]\!]\,env_V\ env_P,$$
$$\mathcal{S}_{\rm ds}[\![S_2]\!]\,env_V\ env_P)$$

$$\mathcal{S}_{\rm ds}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\,env_V\ env_P = \textsf{FIX}\ F$$
$$\text{where } F\ g = \textsf{cond}(\mathcal{B}[\![b]\!]{\circ}(\text{lookup}\ env_V),$$
$$g \circ (\mathcal{S}_{\rm ds}[\![S]\!]\,env_V\ env_P),\ \text{id})$$

$$\mathcal{S}_{\rm ds}[\![\texttt{begin}\ D_V\ D_P\ S\ \texttt{end}]\!]\,env_V\ env_P\ sto = \mathcal{S}_{\rm ds}[\![S]\!]\,env'_V\ env'_P\ sto'$$
$$\text{where } \mathcal{D}^{\rm V}_{\rm ds}[\![D_V]\!](env_V,\ sto) = (env'_V,\ sto')$$
$$\text{and } \mathcal{D}^{\rm P}_{\rm ds}[\![D_P]\!]\,env'_V\ env_P = env'_P$$

$$\mathcal{S}_{\rm ds}[\![\texttt{call}\ p]\!]\,env_V\ env_P = env_P\ p$$

Table 6.4 Denotational semantics for **Proc**

Next we update the procedure environment:

$$\begin{aligned}&\mathcal{D}^{\rm P}_{\rm ds}[\![\texttt{proc p is x := 0;}]\!](env_V[\texttt{x}{\mapsto}\mathbf{12}])\ env_P\\ &\quad = \mathcal{D}^{\rm P}_{\rm ds}[\![\varepsilon]\!](env_V[\texttt{x}{\mapsto}\mathbf{12}])\ (env_P[\texttt{p}{\mapsto}g])\\ &\quad = env_P[\texttt{p}{\mapsto}g]\end{aligned}$$

where

$$\begin{aligned} g\ sto &= \mathcal{S}_{\rm ds}[\![\texttt{x := 0}]\!](env_V[\texttt{x}{\mapsto}\mathbf{12}])\ env_P\ sto\\ &= sto[\mathbf{12}{\mapsto}\mathbf{0}]\end{aligned}$$

because `x` is to be found in location **12** according to the variable environment. Then we get

$$\begin{aligned}&\mathcal{S}_{\rm ds}[\![\texttt{begin var x := 7; proc p is x := 0;}\\ &\qquad\qquad \texttt{begin var x := 5; call p end end}]\!]\,env_V\ env_P\ sto\\ &= \mathcal{S}_{\rm ds}[\![\texttt{begin var x := 5; call p end}]\!]\ (env_V[\texttt{x}{\mapsto}\mathbf{12}])\ (env_P[\texttt{p}{\mapsto}g])\\ &\qquad\qquad (sto[\mathbf{12}{\mapsto}\mathbf{7}][\textsf{next}{\mapsto}\mathbf{13}])\end{aligned}$$

For the variable declarations of the inner block we have

$$\begin{aligned}
&\mathcal{D}^{\rm V}_{\rm ds}[\![\texttt{var x := 5;}]\!](env_V[\mathsf{x}\mapsto\mathbf{12}],\ sto[\mathbf{12}\mapsto\mathbf{7}][\mathsf{next}\mapsto\mathbf{13}])\\
&\qquad = \mathcal{D}^{\rm V}_{\rm ds}[\![\varepsilon]\!](env_V[\mathsf{x}\mapsto\mathbf{13}],\ sto[\mathbf{12}\mapsto\mathbf{7}][\mathbf{13}\mapsto\mathbf{5}][\mathsf{next}\mapsto\mathbf{14}])\\
&\qquad = (env_V[\mathsf{x}\mapsto\mathbf{13}],\ sto[\mathbf{12}\mapsto\mathbf{7}][\mathbf{13}\mapsto\mathbf{5}][\mathsf{next}\mapsto\mathbf{14}])
\end{aligned}$$

and

$$\mathcal{D}^{\rm P}_{\rm ds}[\![\varepsilon]\!](env_V[\mathsf{x}\mapsto\mathbf{13}])\ (env_P[\mathsf{p}\mapsto g]) = env_P[\mathsf{p}\mapsto g]$$

Thus we get

$$\begin{aligned}
&\mathcal{S}_{\rm ds}[\![\texttt{begin var x := 5; call p end}]\!]\ (env_V[\mathsf{x}\mapsto\mathbf{12}])\ (env_P[\mathsf{p}\mapsto g])\\
&\qquad\qquad (sto[\mathbf{12}\mapsto\mathbf{7}][\mathsf{next}\mapsto\mathbf{13}])\\
&\quad = \mathcal{S}_{\rm ds}[\![\texttt{call p}]\!](env_V[\mathsf{x}\mapsto\mathbf{13}])\ (env_P[\mathsf{p}\mapsto g])\\
&\qquad\qquad (sto[\mathbf{12}\mapsto\mathbf{7}][\mathbf{13}\mapsto\mathbf{5}][\mathsf{next}\mapsto\mathbf{14}])\\
&\quad = g\ (sto[\mathbf{12}\mapsto\mathbf{7}][\mathbf{13}\mapsto\mathbf{5}][\mathsf{next}\mapsto\mathbf{14}])\\
&\quad = sto[\mathbf{12}\mapsto\mathbf{0}][\mathbf{13}\mapsto\mathbf{5}][\mathsf{next}\mapsto\mathbf{14}]
\end{aligned}$$

so we see that in the final store the location for the local variable has the value **5** and the one for the global variable has the value **0**. ☐

Exercise 6.6

Consider the following statement in **Proc**:

```
begin var x := 0;
          proc p is x := x+1;
          proc q is call p;
          begin  proc p is x := 7;
                 call q
          end
end
```

Use the semantic clauses of **Proc** to illustrate that procedures have static scope; that is, show that the final store will associate the location of `x` with the value **1** (rather than **7**). ☐

$$\mathcal{D}_{\mathrm{ds}}^{\mathrm{P}}[\![\texttt{proc}\ p\ \texttt{is}\ S;\ D_P]\!]\mathit{env}_V\ \mathit{env}_P = \mathcal{D}_{\mathrm{ds}}^{\mathrm{P}}[\![D_P]\!]\mathit{env}_V\ (\mathit{env}_P[p{\mapsto}\mathsf{FIX}\ F])$$

$$\text{where } F\ g = \mathcal{S}_{\mathrm{ds}}[\![S]\!]\mathit{env}_V\ (\mathit{env}_P[p{\mapsto}g])$$

$$\mathcal{D}_{\mathrm{ds}}^{\mathrm{P}}[\![\varepsilon]\!]\mathit{env}_V = \mathrm{id}$$

Table 6.5 Denotational semantics for recursive procedure declarations

Recursive Procedures

In the case where procedures are allowed to be *recursive*, we shall be interested in a function g in **Store** $\hookrightarrow$ **Store** satisfying

$$g = \mathcal{S}_{\mathrm{ds}}[\![S]\!]\mathit{env}_V\ (\mathit{env}_P[p{\mapsto}g])$$

since this will ensure that the meaning of all the recursive calls is the same as that of the procedure being defined. For this we only need to change the clause for $\mathcal{D}_{\mathrm{ds}}^{\mathrm{P}}[\![\texttt{proc}\ p\ \texttt{is}\ S;\ D_P]\!]$ of Table 6.3, and the new clause is given in Table 6.5. We shall see in Exercise 6.8 that this is a permissible definition; that is, F of Table 6.5 is indeed continuous.

Remark Let us briefly compare the semantics above with the operational semantics given in Section 3.2 for the same language. In the operational semantics, the possibility of recursion is handled by updating the environment *each time the procedure is called* and, except for recording the declaration, no action takes place when the procedure is declared. In the denotational approach, the situation is very different. The possibility of recursion is handled *once and for all*, namely *when the procedure is declared.* □

Exercise 6.7

Consider the declaration of the factorial procedure

```
proc fac is begin  var z := x;
                   if x = 1 then skip
                   else (x := x − 1; call fac; y := z ⋆ y)
             end;
```

Assume that the initial environments are env_V and env_P and that $\mathit{env}_V\ \texttt{x} = l_{\mathrm{x}}$ and $\mathit{env}_V\ \texttt{y} = l_{\mathrm{y}}$. Determine the updated procedure environment. □

As for **While**, we must ensure that the semantic clauses define a total function $\mathcal{S}_{\mathrm{ds}}$. We leave the details to the exercise below.

Exercise 6.8 (**)

To ensure that the clauses for $\mathcal{S}_{\mathrm{ds}}$ define a total function, we must show that FIX is applied only to continuous functions. In the case of recursive procedures, this is a rather laborious task. First, one may use structural induction to show that $\mathcal{D}^{\mathrm{V}}_{\mathrm{ds}}$ is indeed a well-defined function. Second, one may define

$$env_P \sqsubseteq' env'_P \text{ if and only if } env_P\ p \sqsubseteq env'_P\ p \text{ for all } p \in \mathbf{Pname}$$

and show that $(\mathbf{Env}_{\mathrm{P}}, \sqsubseteq')$ is a ccpo. Finally, one may use Exercise 5.41 (with D being $\mathbf{Store} \hookrightarrow \mathbf{Store}$) to show that for all $env_V \in \mathbf{Env}_{\mathrm{V}}$ the clauses of Tables 6.2, 6.4, and 6.5 do define continuous functions

$$\mathcal{S}_{\mathrm{ds}}[\![S]\!]env_V\colon \mathbf{Env}_{\mathrm{P}} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

and

$$\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}[\![D_P]\!]env_V\colon \mathbf{Env}_{\mathrm{P}} \to \mathbf{Env}_{\mathrm{P}}$$

This is performed using mutual structural induction on statements S and declarations D_{P}. □

Exercise 6.9

Modify the syntax of procedures so that they take two *call-by-value* parameters:

$$\begin{array}{rcl} D_P & ::= & \texttt{proc}\ p(x_1,x_2)\ \texttt{is}\ S;\ D_P \mid \varepsilon \\ S & ::= & \cdots \mid \texttt{call}\ p(a_1,a_2) \end{array}$$

The meaning of a procedure will now depend upon the values of its parameters as well as the store in which it is executed. We therefore change the definition of $\mathbf{Env}_{\mathrm{P}}$ to be

$$\mathbf{Env}_{\mathrm{P}} = \mathbf{Pname} \to ((\mathbf{Z} \times \mathbf{Z}) \to (\mathbf{Store} \hookrightarrow \mathbf{Store}))$$

so that given a pair of values and a store we can determine the final store. Modify the definition of $\mathcal{S}_{\mathrm{ds}}$ to use this procedure environment. Also provide semantic clauses for $\mathcal{D}^{\mathrm{P}}_{\mathrm{ds}}$ in the case of non-recursive as well as recursive procedures. Finally, construct statements that illustrate how the new clauses are used. □

Exercise 6.10 (*)

Modify the semantics of **Proc** so that dynamic scope rules are employed for variables as well as procedures. □

6.2 Continuations

Another important concept from denotational semantics is that of *continuations*. To illustrate it, we shall consider an extension of **While** where exceptions can be raised and handled. The new language is called **Exc** and its syntax is

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1 \; ; \; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ S_1\ \texttt{handle}\ e\texttt{:}\ S_2\ \texttt{end} \mid \texttt{raise}\ e \end{array}$$

The meta-variable e ranges over the syntactic category **Exception** of exceptions. The statement `raise` e is a kind of jump instruction: when it is encountered, the execution of the encapsulating block is stopped and the flow of control is given to the statement declaring the exception e. An example is the statement

```
begin while true do  if x≤0
                     then raise exit
                     else x := x−1
      handle exit: y := 7
end
```

Assume that s_0 is the initial state and that s_0 `x` $>$ **0**. Then the false branch of the conditional will be chosen and the value of `x` decremented. Eventually, `x` gets the value **0** and the true branch of the conditional will raise the exception `exit`. This will cause the execution of the `while`-loop to be terminated and control will be transferred to the handler for `exit`. Thus the statement will terminate in a state where `x` has the value **0** and `y` the value **7**.

The meaning of an exception will be the effect of *executing the remainder of the program* starting from the handler. Consider a statement of the form

$$(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2)\ ;\ S_3$$

In the language **While** it is evident that independently of whether we execute S_1 or S_2 we have to continue with S_3. When we introduce exceptions this does not hold any longer: if one of the branches raises an exception not handled inside that branch, then we will certainly not execute S_3. It is therefore necessary to rewrite the semantics of **While** to make the "effect of executing the remainder of the program" more explicit.

Continuation Style Semantics for While

In a *continuation style semantics*, the continuations describe the *effect of executing the remainder of the program.* For us, a *continuation* c is an element of

$$\begin{array}{rcl}
\mathcal{S}'_{\text{cs}}[\![x{:=}a]\!]c\ s & = & c\ (s[x{\mapsto}\mathcal{A}[\![a]\!]s]) \\
\mathcal{S}'_{\text{cs}}[\![\texttt{skip}]\!] & = & \text{id} \\
\mathcal{S}'_{\text{cs}}[\![S_1\ ;\ S_2]\!] & = & \mathcal{S}'_{\text{cs}}[\![S_1]\!] \circ \mathcal{S}'_{\text{cs}}[\![S_2]\!] \\
\mathcal{S}'_{\text{cs}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]c & = & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}'_{\text{cs}}[\![S_1]\!]c, \mathcal{S}'_{\text{cs}}[\![S_2]\!]c) \\
\mathcal{S}'_{\text{cs}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] & = & \mathsf{FIX}\ G \\
\text{where } (G\ g)\ c & = & \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}'_{\text{cs}}[\![S]\!](g\ c),\ c)
\end{array}$$

Table 6.6 Continuation style semantics for **While**

the domain

$$\textbf{Cont} = \textbf{State} \hookrightarrow \textbf{State}$$

and is thus a partial function from **State** to **State**. Sometimes one uses partial functions from **State** to a “simpler” set **Ans** of answers, but in all cases the purpose of a continuation is to express the “outcome” of the remainder of the program when started in a given state.

Consider a statement of the form $\cdots ; S ; \cdots$ and let us explain the meaning of S in terms of the effect of executing the remainder of the program. The starting point will be the continuation c determining the effect of executing the part of the program *after* S; that is, $c\ s$ is the state obtained when the remainder of the program is executed from state s. We shall then determine the effect of executing S *and* the remainder of the program; that is, we shall determine a continuation c' such that $c'\ s$ is the state obtained when executing S and the part of the program following S from state s. Pictorially, from

$$\cdots \qquad ; \qquad S \qquad ; \underbrace{\qquad \cdots \qquad}_{c}$$

we want to obtain

$$\cdots \qquad ; \underbrace{\qquad S \qquad ; \qquad \cdots \qquad}_{c'}$$

We shall define a semantic function $\mathcal{S}'_{\text{cs}}$ for **While** that achieves this. It has functionality

$$\mathcal{S}'_{\text{cs}}\text{: } \textbf{Stm} \to (\textbf{Cont} \to \textbf{Cont})$$

and is defined in Table 6.6. The clauses for assignment and `skip` are straightforward; however, note that we now use id as the identity function on **Cont**; that is, id $c\ s = c\ s$. In the clause for composition, the order of the functional composition is *reversed* compared with the direct style semantics of Table 5.1.

Intuitively, the reason is that the continuations are "pulled backwards" through the two statements. So assuming that c is the continuation for the remainder of the program, we shall first determine a continuation for S_2 followed by the remainder of the program and next for S_1 followed by S_2 *and* the remainder of the program.

The clause for the conditional is straightforward as the continuation applies to both branches. In the clause for the `while`-construct, we use the fixed point operator as in the direct style semantics. If the test of `while` b `do` S evaluates to **ff**, then we return the continuation c for the remainder of the program. If the test evaluates to **tt**, then $g\ c$ denotes the effect of executing the remainder of the loop followed by the remainder of the program and is the continuation to be used for the first unfolding of the loop.

Example 6.11

Consider the statement `z := x; x := y; y := z` of Chapter 1. Let id be the identity function on **State**. Then we have

$$\begin{aligned}
&\mathcal{S}'_{\text{cs}}[\![\texttt{z := x; x := y; y := z}]\!]\text{id}\\
&= (\mathcal{S}'_{\text{cs}}[\![\texttt{z := x}]\!] \circ \mathcal{S}'_{\text{cs}}[\![\texttt{x := y}]\!] \circ \mathcal{S}'_{\text{cs}}[\![\texttt{y := z}]\!])\ \text{id}\\
&= (\mathcal{S}'_{\text{cs}}[\![\texttt{z := x}]\!] \circ \mathcal{S}'_{\text{cs}}[\![\texttt{x := y}]\!])\ g_1\\
&\qquad \text{where } g_1\ s \quad = \text{id}(s[\texttt{y}\mapsto(s\ \texttt{z})])\\
&\qquad\qquad\qquad\quad = \mathcal{S}'_{\text{cs}}[\![\texttt{z := x}]\!]g_2\\
&\qquad \text{where } g_2\ s \quad = g_1(s[\texttt{x}\mapsto(s\ \texttt{y})])\\
&\qquad\qquad\qquad\quad = \text{id}(s[\texttt{x}\mapsto(s\ \texttt{y})][\texttt{y}\mapsto(s\ \texttt{z})])\\
&= g_3\\
&\qquad \text{where } g_3\ s \quad = g_2(s[\texttt{z}\mapsto(s\ \texttt{x})])\\
&\qquad\qquad\qquad\quad = \text{id}(s[\texttt{z}\mapsto(s\ \texttt{x})][\texttt{x}\mapsto(s\ \texttt{y})][\texttt{y}\mapsto(s\ \texttt{x})])
\end{aligned}$$

Note that the semantic function is constructed in a "backwards" manner. □

As in the case of the direct style semantics, we must ensure that the semantic clauses define a total function $\mathcal{S}'_{\text{cs}}$. We leave the details to the exercise below.

Exercise 6.12 (**)

To ensure that the clauses for $\mathcal{S}'_{\text{cs}}$ define a total function, we must show that FIX is only applied to continuous functions. First, one may define

$$g_1 \sqsubseteq' g_2 \text{ if and only if } g_1\ c \sqsubseteq g_2\ c \text{ for all } c \in \mathbf{Cont}$$

and show that ($\mathbf{Cont} \to \mathbf{Cont}$, $\sqsubseteq'$) is a ccpo. Second, one may define

$$[\mathbf{Cont} \to \mathbf{Cont}] = \{\ g\colon \mathbf{Cont} \to \mathbf{Cont} \mid g \text{ is continuous } \}$$

and show that ($[\mathbf{Cont} \to \mathbf{Cont}]$, $\sqsubseteq'$) is a ccpo. Finally, one may use Exercise 5.41 (with $D = [\mathbf{Cont} \to \mathbf{Cont}]$) to show that the clauses of Table 6.6 define a function

$$\mathcal{S}'_{\text{cs}}\colon \mathbf{Stm} \to [\mathbf{Cont} \to \mathbf{Cont}]$$

using structural induction on S. □

Exercise 6.13 (*)

Prove that the two semantic functions $\mathcal{S}_{\text{ds}}$ and $\mathcal{S}'_{\text{cs}}$ satisfy

$$\mathcal{S}'_{\text{cs}}[\![S]\!]c = c \circ \mathcal{S}_{\text{ds}}[\![S]\!]$$

for all statements S of **While** and for all continuations c. □

Exercise 6.14

Extend the language **While** with the construct `repeat` S `until` b and give the new (compositional) clause for $\mathcal{S}'_{\text{cs}}$. □

The Semantic Function $\mathcal{S}_{\text{cs}}$ for Exc

In order to keep track of the exceptions that have been introduced, we shall use an *exception environment*. It will be an element, env_E, of

$$\mathbf{Env}_{\text{E}} = \mathbf{Exception} \to \mathbf{Cont}$$

Given an exception environment env_E and an exception e, the effect of executing the remainder of the program starting from the handler for e, will then be $env_E\ e$.

The semantic function $\mathcal{S}_{\text{cs}}$ for the statements of the language **Exc** has functionality

$$\mathcal{S}_{\text{cs}}\colon \mathbf{Stm} \to \mathbf{Env}_{\text{E}} \to (\mathbf{Cont} \to \mathbf{Cont})$$

The function is defined by the clauses of Table 6.7. Most of the clauses are straightforward extensions of those given for **While** in Table 6.6. The meaning of the block construct is to execute the body in the updated environment. Therefore the environment is updated so that e is bound to the effect of executing the remainder of the program starting from the handler for e and this is the continuation obtained by executing first S_2 and then the remainder of the

$$
\begin{aligned}
&\mathcal{S}_{\text{cs}}[\![x{:=}a]\!]\mathit{env}_E\ c\ s = c\ (s[x{\mapsto}\mathcal{A}[\![a]\!]s])\\
&\mathcal{S}_{\text{cs}}[\![\texttt{skip}]\!]\mathit{env}_E = \text{id}\\
&\mathcal{S}_{\text{cs}}[\![S_1\ ;\ S_2]\!]\mathit{env}_E = (\mathcal{S}_{\text{cs}}[\![S_1]\!]\mathit{env}_E) \circ (\mathcal{S}_{\text{cs}}[\![S_2]\!]\mathit{env}_E)\\
&\mathcal{S}_{\text{cs}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\mathit{env}_E\ c =\\
&\qquad \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\text{cs}}[\![S_1]\!]\mathit{env}_E\ c,\ \mathcal{S}_{\text{cs}}[\![S_2]\!]\mathit{env}_E\ c)\\
&\mathcal{S}_{\text{cs}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\mathit{env}_E = \mathsf{FIX}\ G\\
&\qquad \text{where } (G\ g)\ c = \mathsf{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}_{\text{cs}}[\![S]\!]\mathit{env}_E\ (g\ c),\ c)\\
&\mathcal{S}_{\text{cs}}[\![\texttt{begin}\ S_1\ \texttt{handle}\ e{:}\ S_2\ \texttt{end}]\!]\mathit{env}_E\ c =\\
&\qquad \mathcal{S}_{\text{cs}}[\![S_1]\!](\mathit{env}_E[e{\mapsto}\mathcal{S}_{\text{cs}}[\![S_2]\!]\mathit{env}_E\ c])\ c\\
&\mathcal{S}_{\text{cs}}[\![\texttt{raise}\ e]\!]\mathit{env}_E\ c = \mathit{env}_E\ e
\end{aligned}
$$

Table 6.7 Continuation style semantics for **Exc**

program; that is, $\mathcal{S}_{\text{cs}}[\![S_2]\!]\mathit{env}_E\ c$. Finally, in the clause for `raise` e, we *ignore* the continuation that is otherwise supplied. So rather than using c, we choose to use $\mathit{env}_E\ e$.

Example 6.15

Let env_E be an initial environment and assume that the initial continuation is the identity function, id. Then we have

$$
\begin{aligned}
&\mathcal{S}_{\text{cs}}[\![\ \texttt{begin while true do if x}{\le}\texttt{0 then raise exit else x := x}{-}\texttt{1}\\
&\qquad \texttt{handle exit: y := 7 end}]\!]\mathit{env}_E\ \text{id}\\
&=\quad (\mathsf{FIX}\ G)\ \text{id}
\end{aligned}
$$

where G is defined by

$$
\begin{aligned}
G\ g\ c\ s \quad &= \quad \mathsf{cond}(\ \mathcal{B}[\![\texttt{true}]\!],\\
&\qquad\qquad \mathsf{cond}(\mathcal{B}[\![\texttt{x}{\le}\texttt{0}]\!],\ c_{\text{exit}},\\
&\qquad\qquad\qquad S_{\text{cs}}[\![\texttt{x := x}{-}\texttt{1}]\!]\mathit{env}_E[\texttt{exit}{\mapsto}c_{\text{exit}}]\ (g\ c)),\\
&\qquad\qquad c)\ s\\
&= \begin{cases} c_{\text{exit}}\ s & \text{if } s\ \texttt{x} \le \mathbf{0}\\ (g\ c)\ (s[\texttt{x}{\mapsto}(s\ \texttt{x}){-}\mathbf{1}]) & \text{if } s\ \texttt{x} > \mathbf{0}\end{cases}
\end{aligned}
$$

and the continuation c_{exit} associated with the exception `exit` is given by

$$c_{\text{exit}}\ s = \text{id}\ (s[\mathtt{y}{\mapsto}\mathbf{7}]) = s[\mathtt{y}{\mapsto}\mathbf{7}]$$

Note that G will use the "default" continuation c or the continuation c_{exit} associated with the exception, as appropriate. We then get

$$(\mathsf{FIX}\ G)\ \text{id}\ s = \begin{cases} s[\mathtt{y}{\mapsto}\mathbf{7}] & \text{if } s\ \mathtt{x} \leq \mathbf{0} \\ s[\mathtt{x}{\mapsto}\mathbf{0}][\mathtt{y}{\mapsto}\mathbf{7}] & \text{if } s\ \mathtt{x} > \mathbf{0} \end{cases}$$

Exercise 6.16

Show that $\mathsf{FIX}\ G$ as specified in the example above is indeed the least fixed point; that is, construct the iterands $G^{\text{n}}\ \bot$ and show that their least upper bound is as specified. □

Exercise 6.17 (**)

Extend Exercise 6.12 to show the well-definedness of the function $\mathcal{S}_{\text{cs}}$ defined by the clauses of Table 6.7. □

Exercise 6.18

Suppose that there is a distinguished output variable `out` $\in$ **Var** and that only the final value of this variable is of interest. This might motivate defining

$$\mathbf{Cont} = \mathbf{State} \hookrightarrow \mathbf{Z}$$

Define the initial continuation $c_0 \in \mathbf{Cont}$. What changes to $\mathbf{Env}_{\text{E}}$, the functionality of $\mathcal{S}_{\text{cs}}$, and Table 6.7 are necessary? □

7
Program Analysis

The availability of powerful tools is crucial for the design, implementation, and maintenance of large programs. Advanced programming environments provide many such tools: syntax-directed editors, optimizing compilers, and debuggers — in addition to tools for transforming programs and estimating their performance. Program analyses play a major role in many of these tools: they are able to give useful information about the dynamic behaviour of programs without actually running them. Below we give a few examples.

In a *syntax-directed editor*, we may meet warnings like "variable `x` is used before it is initialised", "the part of the program starting at line 1298 and ending at line 1354 will never be executed", or "there is a reference outside the bounds of array `a`". Such information is the result of various program analyses. The first warning is the result of a *definition-use* analysis: at each point of a program where the value of a variable is used, the analysis will determine those points where the variable might have obtained its present value; if there are none, then clearly the variable is uninitialised. The second warning might be the result of a *constant propagation* analysis: at each point of a program where an expression is evaluated, the analysis will attempt to deduce that the expression always evaluates to a constant. So if the expression is the test of a conditional, we might deduce that only one of the branches can ever be taken. The third warning could be the result of an *interval* analysis: instead of determining a possible constant value, we determine upper and lower bounds of the value to which the expression may evaluate. So if the expression is an index into an array, then a comparison with the bounds of the array will suffice for issuing the third warning above.

Traditionally, program analyses have been developed for *optimizing compilers*. They are used at all levels in the compiler: some optimizations apply to the source program, others to the various intermediate representations used inside the compiler, and finally there are optimizations that exploit the architecture of the target machine and therefore directly improve the target code. The improvements facilitated by these analyses, and the associated transformations, may result in dramatic reductions of the running time. One example is the *available expressions* analysis: an expression E is available at a program point p if E has been computed previously and the values of the free variables of E have not changed since then. Clearly we can avoid recomputing the expression and instead use the previously computed value. This information is particularly useful at the intermediate level in a compiler for computing actual addresses into data structures with arrays as a typical example. Another example is *live variable* analysis: given a variable x and a point p in the program, will the value of x at p be used at a later stage? If so, then x is said to be live at p; otherwise it is said to be dead. Such information is useful when deciding how to use the registers of a machine: the values of dead variables need not be stored in memory when the registers in which they reside are reused for other purposes.

Many *program transformations* are only valid when certain conditions are fulfilled. As an example, it is only safe to move the computation of an expression outside a loop if its value is not affected by the computations in the remainder of the loop. Similarly, we may only replace an expression with a variable if we know that the expression already has been evaluated in a context where it gave the same result as here and where its value has been assigned to the variable. Such information may be collected by a slight extension of an *available expression* analysis: an expression E is available in x at a program point p if E has been evaluated and assigned to x on all paths leading to p and if the values of x and the free variables of E have not changed since then.

Properties and Property States

Program analyses give information about the *dynamic* behaviour of programs. The analyses are performed *statically*, meaning that the programs are *not* run on all possible inputs in order to find the result of the analysis. On the other hand, the analyses are *safe*, meaning that the result of the analysis describes all possible runs of the program. This effect is obtained by letting the analysis compute with *abstract properties* of the "real" values rather than with the "real" values themselves.

Let **P** denote the set of abstract properties. As an example, in the detection of signs analysis, **P** will contain the properties POS, ZERO, NEG, and ANY (and others), and in a live variable analysis it will contain the properties LIVE and DEAD. Since some properties are more discriminating than others, we shall equip **P** with a *partial ordering* $\sqsubseteq_P$. So, for example, in the detection of signs analysis, we have POS $\sqsubseteq_P$ ANY because it is more discriminating to know that a number is positive than it is to know that it can have any sign. Similarly, in the live variable analysis, we may take DEAD $\sqsubseteq_P$ LIVE because it is more discriminating to know that the value of a variable definitely is not used in the rest of the computation than it is to know that it might be used. (One might rightly feel that this intuition is somewhat at odds with the viewpoint of denotational semantics; nonetheless, the approach makes sense!) When specifying an analysis, we shall always make sure that

$$(\mathbf{P},\ \sqsubseteq_P) \text{ is a complete lattice}$$

as defined in Chapter 5. The lattice structure gives us a convenient method for *combining* properties: if some value has one of the two properties p_1 or p_2, then we can combine this to say that it has the property $\bigsqcup_P\{p_1, p_2\}$, where $\bigsqcup_P$ is the least upper bound operation on **P**. It is convenient to write $p_1 \sqcup_P p_2$ for $\bigsqcup_P\{p_1, p_2\}$.

Many analyses (for example, the detection of signs analysis) associate properties with the individual variables of the program, and here the function spaces are often used to model *property states*: in the standard semantics, states map variables to values, whereas in the analysis, the property states will map variables to their properties:

$$\mathbf{PState} = \mathbf{Var} \rightarrow \mathbf{P}$$

The property states inherit the ordering from the properties in a pointwise manner. This is in fact a corollary of a fairly general result, which can be expressed as follows.

Lemma 7.1

Assume that S is a non-empty set and that $(D,\ \sqsubseteq)$ is a partially ordered set. Let $\sqsubseteq'$ be the ordering on the set $S \rightarrow D$ defined by

$$f_1\ \sqsubseteq'\ f_2 \text{ if and only if } f_1\ x\ \sqsubseteq\ f_2\ x \text{ for all } x \in S$$

Then $(S \rightarrow D,\ \sqsubseteq')$ is a partially ordered set. Furthermore, $(S \rightarrow D,\ \sqsubseteq')$ is a ccpo if D is, and it is a complete lattice if D is. In both cases, we have

$$(\textstyle\bigsqcup' Y)\ x = \bigsqcup\{f\ x \mid f \in Y\}$$

so that least upper bounds are determined pointwise.

Proof: It is straightforward to verify that $\sqsubseteq'$ is a partial order, so we omit the details. We shall first prove the lemma in the case where D is a complete lattice, so let Y be a subset of $S \to D$. Then the formula

$$(\bigsqcup' Y)\ x = \bigsqcup\{f\ x \mid f \in Y\}$$

defines an element $\bigsqcup' Y$ of $S \to D$ because D being a complete lattice means that $\bigsqcup\{f\ x \mid f \in Y\}$ exists for all x of S. This shows that $\bigsqcup' Y$ is a *well-defined* element of $S \to D$. To see that $\bigsqcup' Y$ is an *upper bound* of Y, let $f_0 \in Y$, and we shall show that $f_0 \sqsubseteq' \bigsqcup' Y$. This amounts to considering an arbitrary x in S and showing

$$f_0\ x\ \sqsubseteq\ \bigsqcup\{f\ x \mid f \in Y\}$$

and this is immediate because $\bigsqcup$ is the least upper bound operation in D. To see that $\bigsqcup' Y$ is the *least* upper bound of Y, let f_1 be an upper bound of Y, and we shall show that $\bigsqcup' Y\ \sqsubseteq'\ f_1$. This amounts to showing

$$\bigsqcup\{f\ x \mid f \in Y\}\ \sqsubseteq\ f_1\ x$$

for an arbitrary $x \in S$. However, this is immediate because $f_1\ x$ must be an upper bound of $\{f\ x \mid f \in Y\}$ and because $\bigsqcup$ is the least upper bound operation in D.

To prove the other part of the lemma, assume that D is a ccpo and that Y is a chain in $S \to D$. The formula

$$(\bigsqcup' Y)\ x = \bigsqcup\{f\ x \mid f \in Y\}$$

defines an element $\bigsqcup' Y$ of $S \to D$: each $\{f\ x \mid f \in Y\}$ will be a chain in D because Y is a chain, and hence each $\bigsqcup\{f\ x \mid f \in Y\}$ exists because D is a ccpo. That $\bigsqcup' Y$ is the least upper bound of Y in $S \to D$ is as above. □

Side-stepping the Halting Problem

It is important to realize that *exact* answers to many of the program analyses we have mentioned above involve solving the Halting Problem! As an example, consider the program fragment

$$(\texttt{if} \cdots \texttt{then x := 1 else } (S;\ \texttt{x := 2}));\ \texttt{y := x}$$

and the constant propagation analysis: does `x` always evaluate to the constant 1 in the assignment to `y`? This is the case if and only if S never terminates (or if S ends with a run-time error). To allow for an implementable analysis, we allow constant propagation to provide *safe approximations* to the *exact* answers: in the example, we always deem that `x` does not evaluate to a constant at the assignment to `y`. In this way, the possible outcomes of the analysis are

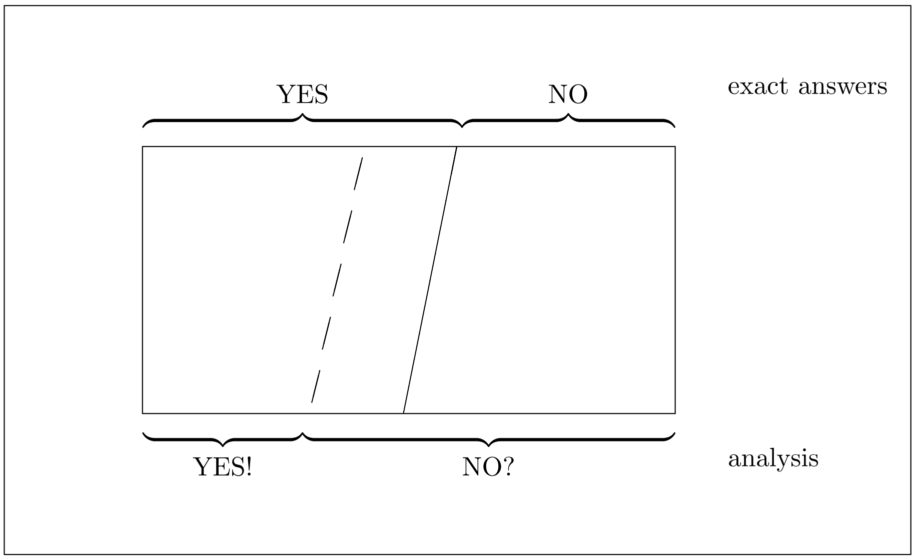


Figure 7.1 Safe approximations to exact answers

– YES!: the expression E *always* evaluates to the constant c or

– NO?: the expression E *might not always* evaluate to a constant

and where the second answer is *not* equivalent to saying that "the expression E *does not always* evaluate to a constant". To be useful, the second answer should not be produced too often: a useless but correct analysis is obtained by always producing the second answer. We shall see that this is a general phenomenon: one answer of the analysis will imply the exact answer, but the other answer will not; this is illustrated in Figure 7.1, and we say that the analysis *errs on the safe side.* To be more precise about this requires a study of the concrete analysis and the use to which we intend to put it. We thus return to the issue when dealing with the correctness of a given analysis.

7.1 Detection of Signs Analysis: Specification

The rules for computation with signs are well-known and the idea is now to turn them into an analysis of programs in the **While** language. The specification of the analysis falls into two parts. First we have to specify the properties with which we compute: in this case, properties of numbers and truth values. Next we specify the analysis itself for the three syntactic categories: arithmetic expressions, boolean expressions, and statements.

The detection of signs analysis is based on three basic properties of numbers:

– POS: the number is positive,

– ZERO: the number is zero, and

– NEG: the number is negative.

Although a given number will have one (and only one) of these properties it is obvious that we easily lose precision when calculating with signs: the subtraction of two positive numbers may give any number so the sign of the result cannot be described by one of the three basic properties. This is a common situation in program analysis, and the solution is to introduce extra properties that express *combinations* of the basic properties. For the detection of signs analysis, we may add the following properties:

– NON-NEG: the number is not negative,

– NON-ZERO: the number is not zero,

– NON-POS: the number is not positive, and

– ANY: the number can have any sign.

For each property, we can determine a set of numbers that are described by that property. When formulating the analysis, it is convenient to have a property corresponding to the *empty set* of numbers as well, and we therefore introduce the property

– NONE: the number belongs to $\emptyset$.

Now let **Sign** be the set

$$\{\text{NEG, ZERO, POS, NON-POS, NON-ZERO, NON-NEG, ANY, NONE}\}$$

We shall equip **Sign** with a partial ordering $\sqsubseteq_S$ reflecting the subset ordering on the underlying sets of numbers. The ordering is depicted by means of the Hasse diagram of Figure 7.2. So, for example, POS $\sqsubseteq_S$ NON-ZERO holds because $\{z \in \mathbf{Z} \mid z > 0\} \subseteq \{z \in \mathbf{Z} \mid z \neq 0\}$ and NONE $\sqsubseteq_S$ NEG holds because we have $\emptyset \subseteq \{z \in \mathbf{Z} \mid z < 0\}$.

Exercise 7.2 (Essential)

Show that (**Sign**, $\sqsubseteq_S$) is a complete lattice, and let $\bigsqcup_S$ be the associated least upper bound operation. For each pair p_1 and p_2 of elements from **Sign**, specify $p_1 \sqcup_S p_2$. □

Clearly we can associate a "best" property with each number. To formalise this, we define a function

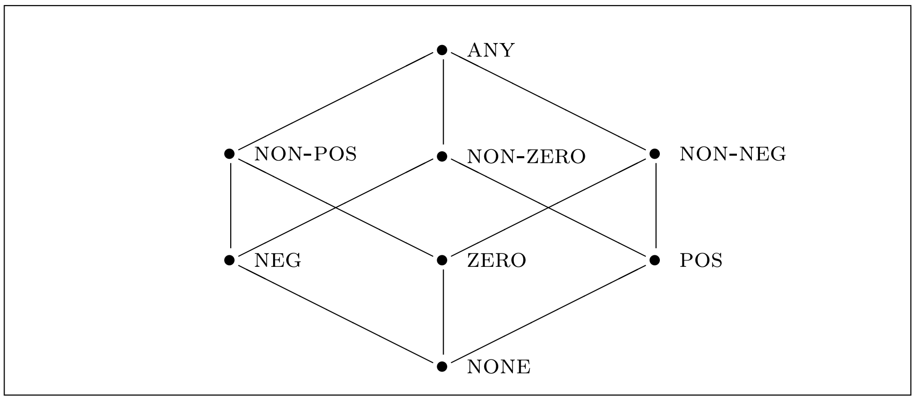


Figure 7.2 The complete lattice of signs $(\mathbf{Sign}, \sqsubseteq_S)$

$$\mathsf{abs}_Z\colon \mathbf{Z} \to \mathbf{Sign}$$

that will abstract a number into its sign:

$$\mathsf{abs}_Z\ z = \begin{cases} \text{NEG} & \text{if } z < 0 \\ \text{ZERO} & \text{if } z = 0 \\ \text{POS} & \text{if } z > 0 \end{cases}$$

In the detection of signs analysis, we define the set **PState** of property states by

$$\mathbf{PState} = \mathbf{Var} \to \mathbf{Sign}$$

and we shall use the meta-variable ps to range over **PState**. The operation abs_Z is lifted to states

$$\mathsf{abs}\colon \mathbf{State} \to \mathbf{PState}$$

by defining it in a componentwise manner: $(\mathsf{abs}\ s)\ x = \mathsf{abs}_Z\ (s\ x)$ for all variables x.

Corollary 7.3

Let $\sqsubseteq_{PS}$ be the ordering on **PState** defined by

$$ps_1 \ \sqsubseteq_{PS}\ ps_2 \text{ if and only if } ps_1\ x\ \sqsubseteq_S\ ps_2\ x \text{ for all } x \in \mathbf{Var}.$$

Then $(\mathbf{PState}, \sqsubseteq_{PS})$ is a complete lattice. In particular, the least upper bound $\bigsqcup_{PS} Y$ of a subset Y of **PState** is characterized by

$$(\textstyle\bigsqcup_{PS} Y)\ x = \bigsqcup_S \{ps\ x \mid ps \in Y\}$$

and is thus defined in a componentwise manner.

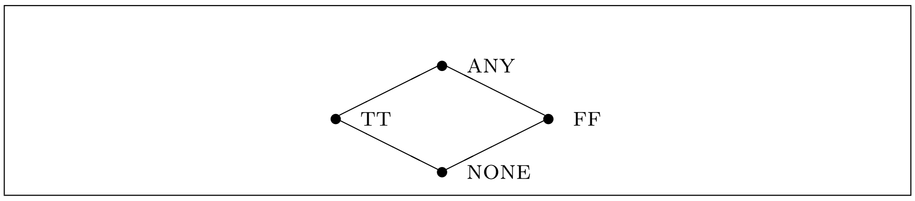


Figure 7.3 The complete lattice ($\mathbf{TT}$, $\sqsubseteq_T$)

Proof: An immediate consequence of Lemma 7.1. □

We shall write INIT for the *least element* of **PState**; that is, for the property state that maps all variables to NONE.

In the analysis, we compute with the properties of **Sign** rather than the numbers of **Z**. Similarly, we will have to replace the truth values **T** with some set of properties: although knowledge about the signs may enable us to predict the truth value of certain boolean expressions this need not always be the case. We shall therefore introduce four properties corresponding to the four subsets of the truth values:

– TT: corresponding to the set $\{\mathbf{tt}\}$,

– FF: corresponding to the set $\{\mathbf{ff}\}$,

– ANY: corresponding to the set $\{\mathbf{tt}, \mathbf{ff}\}$, and

– NONE: corresponding to the set $\emptyset$.

The set is equipped with an ordering $\sqsubseteq_T$ reflecting the subset ordering on the sets of truth values. This is depicted in Figure 7.3.

Exercise 7.4 (Essential)

Show that ($\mathbf{TT}$, $\sqsubseteq_T$) is a complete lattice, and let $\bigsqcup_T$ be the associated least upper bound operation. For each pair p_1 and p_2 of elements from $\mathbf{TT}$, specify $p_1 \sqcup_T p_2$. □

We shall also introduce an abstraction function for truth values. It has the functionality

$$\mathsf{abs}_T\colon \mathbf{T} \to \mathbf{TT}$$

and is defined by $\mathsf{abs}_T\ \mathbf{tt} = \text{TT}$ and $\mathsf{abs}_T\ \mathbf{ff} = \text{FF}$.

$$\begin{array}{rcl}
\mathcal{DA}[\![n]\!]ps & = & \mathsf{abs}_Z(\mathcal{N}[\![n]\!]) \\
\mathcal{DA}[\![x]\!]ps & = & ps\ x \\
\mathcal{DA}[\![a_1 + a_2]\!]ps & = & \mathcal{DA}[\![a_1]\!]ps\ +_S\ \mathcal{DA}[\![a_2]\!]ps \\
\mathcal{DA}[\![a_1\ \star\ a_2]\!]ps & = & \mathcal{DA}[\![a_1]\!]ps\ \star_S\ \mathcal{DA}[\![a_2]\!]ps \\
\mathcal{DA}[\![a_1 - a_2]\!]ps & = & \mathcal{DA}[\![a_1]\!]ps\ -_S\ \mathcal{DA}[\![a_2]\!]ps
\end{array}$$

Table 7.1 Detection of signs analysis of arithmetic expressions

Analysis of Expressions

Recall that the semantics of arithmetic expressions is given by a function

$$\mathcal{A}\colon \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Z}$$

In the analysis, we do not know the exact value of the variables but only their properties, and consequently we can only compute a property of the arithmetic expression. So the analysis will be given by a function

$$\mathcal{DA}\colon \mathbf{Aexp} \to \mathbf{PState} \to \mathbf{Sign}$$

whose defining clauses are given in Table 7.1.

In the clause for n, we use the function abs_Z to determine the property of the corresponding number. For variables, we simply consult the property state. For the composite constructs, we proceed in a compositional manner and rely on addition, multiplication, and subtraction operations defined on **Sign**. For addition, the operation $+_S$ is written in detail in Table 7.2. The multiplication and subtraction operations $\star_S$ and $-_S$ are only partly specified in that table.

The semantics of boolean expressions is given by a function

$$\mathcal{B} : \mathbf{Bexp}\ \to\ \mathbf{State}\ \to\ \mathbf{T}$$

As in the case of arithmetic expressions, the analysis will use property states rather than states. The truth values will be replaced by the set **TT** of properties of truth values so the analysis will be given by a function

$$\mathcal{DB} : \mathbf{Bexp}\ \to\ \mathbf{PState}\ \to\ \mathbf{TT}$$

whose defining clauses are given in Table 7.3.

The clauses for `true` and `false` should be straightforward. For the tests on arithmetic expressions, we rely on operations defined on the lattice **Sign** and giving results in **TT**; these operations are partly specified in Table 7.4. In the case of negation and conjunction, we need similar operations defined on **TT** and these are also specified in Table 7.4.

$+_S$	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE
NEG	NONE	NEG	NEG	ANY	NEG	ANY	ANY	ANY
ZERO	NONE	POS	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
POS	NONE	ANY	POS	POS	ANY	ANY	POS	ANY
NON-POS	NONE	NEG	NON-POS	ANY	NON-POS	ANY	ANY	ANY
NON-ZERO	NONE	ANY	NON-ZERO	ANY	ANY	ANY	ANY	ANY
NON-NEG	NONE	ANY	NON-NEG	POS	ANY	ANY	NON-NEG	ANY
ANY	NONE	ANY	ANY	ANY	ANY	ANY	ANY	ANY

$\star_S$	NEG	ZERO	POS
NEG	POS	ZERO	NEG
ZERO	ZERO	ZERO	ZERO
POS	NEG	ZERO	POS

$-_S$	NEG	ZERO	POS
NEG	ANY	NEG	NEG
ZERO	POS	ZERO	NEG
POS	POS	POS	ANY

Table 7.2 Operations on **Sign**

$$\begin{aligned}
\mathcal{DB}[\![\texttt{true}]\!]ps &= \text{TT} \\
\mathcal{DB}[\![\texttt{false}]\!]ps &= \text{FF} \\
\mathcal{DB}[\![a_1 = a_2]\!]ps &= \mathcal{DA}[\![a_1]\!]ps \ =_S \ \mathcal{DA}[\![a_2]\!]ps \\
\mathcal{DB}[\![a_1 \le a_2]\!]ps &= \mathcal{DA}[\![a_1]\!]ps \ \le_S \ \mathcal{DA}[\![a_2]\!]ps \\
\mathcal{DB}[\![\neg b]\!]ps &= \neg_T \ (\mathcal{DB}[\![b]\!]ps) \\
\mathcal{DB}[\![b_1 \wedge b_2]\!]ps &= \mathcal{DB}[\![b_1]\!]ps \ \wedge_T \ \mathcal{DB}[\![b_2]\!]ps
\end{aligned}$$

Table 7.3 Detection of signs analysis of boolean expressions

Example 7.5

We have $\mathcal{DB}[\![\neg(\texttt{x} = \texttt{1})]\!]ps = \neg_T(ps\ \texttt{x} =_S \text{POS})$, which can be represented by the following table:

ps `x`	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
$\mathcal{DB}[\![\neg(\texttt{x} = \texttt{1})]\!]ps$	NONE	TT	TT	ANY	TT	ANY	ANY	ANY

Thus, if `x` is positive, we cannot deduce anything about the outcome of the test, whereas if `x` is negative, then the test will always give true. □

$=_S$	NEG	ZERO	POS
NEG	ANY	FF	FF
ZERO	FF	TT	FF
POS	FF	FF	ANY

$\leq_S$	NEG	ZERO	POS
NEG	ANY	TT	TT
ZERO	FF	TT	TT
POS	FF	FF	ANY

$\neg_T$	
NONE	NONE
TT	FF
FF	TT
ANY	ANY

$\wedge_T$	NONE	TT	FF	ANY
NONE	NONE	NONE	NONE	NONE
TT	NONE	TT	FF	ANY
FF	NONE	FF	FF	FF
ANY	NONE	ANY	FF	ANY

Table 7.4 Operations on **Sign** and **TT**

$$
\begin{array}{rcl}
\mathcal{DS}[\![x := a]\!]ps & = & ps[x \mapsto \mathcal{DA}[\![a]\!]ps] \\
\mathcal{DS}[\![\texttt{skip}]\!] & = & \mathsf{id} \\
\mathcal{DS}[\![S_1;\ S_2]\!] & = & \mathcal{DS}[\![S_2]\!] \circ \mathcal{DS}[\![S_1]\!] \\
\mathcal{DS}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \mathsf{cond}_D(\mathcal{DB}[\![b]\!], \mathcal{DS}[\![S_1]\!], \mathcal{DS}[\![S_2]\!]) \\
\mathcal{DS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] & = & \mathsf{FIX}\ H \\
\text{where } H\ h & = & \mathsf{cond}_D(\mathcal{DB}[\![b]\!], h \circ \mathcal{DS}[\![S]\!], \mathsf{id})
\end{array}
$$

Table 7.5 Detection of signs analysis of statements

Analysis of Statements

In the denotational semantics, the meaning of statements is given by a function

$$\mathcal{S}_{ds}\colon \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

In the analysis, we perform two changes. First, we replace the states by property states so that given information about the signs of the variables *before* the statement is executed, we will obtain information about the signs of the variables *after* the execution has (possibly) terminated. Second, we replace partial functions by *total* functions; this is a crucial change in that the whole point of performing a program analysis is to trade precision for termination. So the analysis will be specified by a function

$$\mathcal{DS}\colon \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

whose clauses are given in Table 7.5.

At the surface, these clauses are exactly like those of the direct style denotational semantics in Chapter 5. However, the auxiliary function cond_D is different in that it has to take into account that the outcome of the test can be any of the *four* properties of **TT**. We shall define it by

$$\mathsf{cond}_D(f, h_1, h_2)ps = \begin{cases} \text{INIT} & \text{if } f\ ps = \text{NONE} \\ h_1\ ps & \text{if } f\ ps = \text{TT} \\ h_2\ ps & \text{if } f\ ps = \text{FF} \\ (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \text{ANY} \end{cases}$$

Here the operation $\sqcup_{PS}$ is the *binary* least upper bound operation of **PState**:

$$ps_1 \sqcup_{PS} ps_2 = \bigsqcup_{PS}\{ps_1, ps_2\}$$

Since (**PState**, $\sqsubseteq_{PS}$) is a complete lattice (Corollary 7.3), this is a well-defined operation. The idea behind the definition of cond_D is that if the outcome of the test is TT or FF, then we know exactly which branch will be taken when executing the statement. So we select the analysis of that branch to be the result of analysing the conditional. If the outcome of the test is ANY, then the execution of the conditional might result in **tt** as well as **ff**, so in the analysis we have to combine the results of the two branches. So if one branch says that `x` has the property POS and the other says that it has the property NEG, then we can only deduce that after execution of the conditional `x` will be either positive or negative, and this is exactly what is achieved by using the least upper bound operation. The case where the test gives NONE should only be possible in unreachable code, so in this case we return the property state INIT that does not describe any states.

In the clause for the `while`-loop we also use the function cond_D, and otherwise the clause is as in the direct style denotational semantics. In particular, we use the fixed point operation FIX as it corresponds to unfolding the `while`-loop a number of times — once for each time the *analysis* traverses the loop. As in Chapter 5, the fixed point is defined by

$$\mathsf{FIX}\ H = \bigsqcup\{H^{\mathrm{n}}\bot \mid \mathrm{n} \geq 0\}$$

where the functionality of H is

$$H\colon (\mathbf{PState} \to \mathbf{PState}) \to (\mathbf{PState} \to \mathbf{PState})$$

and where **PState** → **PState** is the set of total functions from **PState** to **PState**. In order for this to make sense, H must be a continuous function on a ccpo with least element $\bot$. We shall shortly verify that this is indeed the case.

Example 7.6

We are now in a position, where we can attempt the application of the analysis to the factorial statement

$$\mathcal{DS}[\![\mathtt{y := 1;\ while\ \neg(x = 1)\ do y := y \star x; x := x - 1)}]\!]$$

We shall apply this function to the property state ps_0 that maps `x` to $\textsc{pos}$ and all other variables (including `y`) to $\textsc{any}$. So we are interested in

$$(\mathsf{FIX}\ H)\ (ps_0[\mathtt{y} \mapsto \textsc{pos}])$$

where

$$H\ h = \mathsf{cond}_D(\mathcal{DB}[\![\neg(\mathtt{x} = \mathtt{1})]\!], h \circ h_{\text{fac}}, \mathsf{id})$$

and

$$h_{\text{fac}} = \mathcal{DS}[\![\mathtt{y} := \mathtt{y} \star \mathtt{x}; \mathtt{x} := \mathtt{x} - \mathtt{1}]\!]$$

Thus we have to compute the approximations $H^0\ \bot$, $H^1\ \bot$, $H^2\ \bot \cdots$. Below we shall show that

$$H^3\ \bot(ps_0[\mathtt{y} \mapsto \textsc{pos}])\mathtt{y} = \textsc{any}.$$

Since $H^3\ \bot \sqsubseteq \mathsf{FIX}\ H$, we have

$$H^3\ \bot(ps_0[\mathtt{y} \mapsto \textsc{pos}]) \sqsubseteq_{PS} (\mathsf{FIX}\ H)(ps_0[\mathtt{y} \mapsto \textsc{pos}])$$

and thereby

$$H^3\ \bot(ps_0[\mathtt{y} \mapsto \textsc{pos}])\mathtt{y} \sqsubseteq_P (\mathsf{FIX}\ H)(ps_0[\mathtt{y} \mapsto \textsc{pos}])\mathtt{y}$$

Thus $(\mathsf{FIX}\ H)(ps_0[\mathtt{y} \mapsto \textsc{pos}])\mathtt{y} = \textsc{any}$ must be the case. Thus, even though we start with the assumption that `x` is positive, the analysis cannot deduce that the factorial of `x` is positive. We shall remedy this shortly.

Using the definition of h_{fac} and $\mathcal{DB}[\![\neg(\mathtt{x} = \mathtt{1})]\!]$ (as tabulated in Example 7.5), we get

$$\begin{aligned}
&H^3\ \bot\ (ps_0[\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = H(H^2\ \bot)\ (ps_0[\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = (H^2\ \bot)\ (h_{\text{fac}}(ps_0[\mathtt{y} \mapsto \textsc{pos}]))\\
&\qquad \sqcup_{PS}\ \mathsf{id}\ (ps_0[\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = H(H^1\ \bot)\ (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}])\\
&\qquad \sqcup_{PS}\ (ps_0[\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = (H^1\ \bot)\ (h_{\text{fac}}\ (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}]))\\
&\qquad \sqcup_{PS}\ \mathsf{id}\ (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}]) \sqcup_{PS} (ps_0[\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = H(H^0\ \bot)\ (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{any}])\\
&\qquad \sqcup_{PS}\ (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = (H^0\ \bot)\ (h_{\text{fac}}\ (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{any}]))\\
&\qquad \sqcup_{PS}\ \mathsf{id}(ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{any}]) \sqcup_{PS} (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}])\\
&\quad = \textsc{init} \sqcup_{PS} (ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{any}])\\
&\quad = ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{any}]
\end{aligned}$$

Thus we see that the "mistake" was made when we applied h_{fac} to the property state $ps_0[\texttt{x} \mapsto \textsc{any}][\texttt{y} \mapsto \textsc{pos}]$. □

Remark A more precise analysis may be performed if we change the definition of $\mathsf{cond}_D(f, h_1, h_2)$ in the case where $f\ ps = \textsc{any}$. To do so, we first fix a set $X \subseteq \mathbf{Var}$ of variables and introduce the notion of an X-*atomic* property state: ps is X-atomic if

$$\forall x\ \in\ X : ps\ x\ \in\ \{\textsc{neg}, \textsc{zero}, \textsc{pos}\}$$

Next we introduce the notion of an X-*proper* property state: ps is X-proper if

$$\forall x_1, x_2 \in X : (ps\ x_1 = \textsc{none} \quad \Leftrightarrow \quad ps\ x_2 = \textsc{none})$$

Clearly, a property state need not be X-atomic nor X-proper, but all X-proper property states can be described as the least upper bound of a set of X-atomic property states:

$$ps = \bigsqcup_{PS}\{ps' \mid ps' \sqsubseteq_{PS} ps \text{ and } ps' \text{ is } X\text{-atomic}\}$$

holds for all X-proper property states ps. Throughout the remainder of this remark, we shall restrict our attention to X-proper property states.

Returning to the definition of $\mathsf{cond}_D(f, h_1, h_2)\,ps$, if $f\ ps = \textsc{tt}$, then we know that only the true branch may be taken; if $f\ ps = \textsc{ff}$, then only the false branch may be taken; and only in the case where $f\ ps = \textsc{any}$ do we need to take both possibilities into account. So we define the two sets $\mathsf{filter}_T^X(f, ps)$ and $\mathsf{filter}_F^X(f, ps)$ by

$$\mathsf{filter}_T^X(f, ps) = \{ps' \mid ps' \sqsubseteq_{PS} ps,\ ps' \text{ is } X\text{-atomic},\ \textsc{tt} \sqsubseteq_T f\ ps'\}$$

and

$$\mathsf{filter}_F^X(f, ps) = \{ps' \mid ps' \sqsubseteq_{PS} ps,\ ps' \text{ is } X\text{-atomic},\ \textsc{ff} \sqsubseteq_T f\ ps'\}$$

We can now define $\mathsf{cond}_D^X(f, h_1, h_2)\,ps$ analogously to $\mathsf{cond}_D(f, h_1, h_2)\,ps$ except that we replace $(h_1\ ps) \sqcup_{PS} (h_2\ ps)$ by

$$(h_1\ (\bigsqcup_{PS}\ \mathsf{filter}_T^X(f, ps))) \sqcup_{PS} (h_2\ (\bigsqcup_{PS}\ \mathsf{filter}_F^X(f, ps)))$$

Here $\bigsqcup_{PS}$ is used to combine the set of property states into a single property state before applying the analysis of the particular branch to it. The correctness of this definition is due to the restriction to X-proper property states.

We note in passing that an even more precise analysis might result if the use of $\bigsqcup_{PS}$ was postponed until after h_i had been applied pointwise to the property states in the corresponding set.

Example 7.7

To be able to obtain useful information from the analysis of the factorial statement, we need to do two things:

– replace cond_D by $\mathsf{cond}_D^{\{\mathtt{x},\mathtt{y}\}}$ as given in the previous remark, and

– rewrite the test of the factorial statement to use $\neg(\mathtt{x} \leq \mathtt{1})$ instead of $\neg(\mathtt{x} = \mathtt{1})$.

With these amendments, we are interested in

$$\mathcal{DS}[\![\mathtt{y} := \mathtt{1}; \mathtt{while} \neg(\mathtt{x}{\leq}\mathtt{1}) \mathtt{do} (\mathtt{y} := \mathtt{y}{\star}\mathtt{x}; \mathtt{x} := \mathtt{x} - \mathtt{1})]\!]\ ps_0$$

where ps_0 $\mathtt{x} = \textsc{pos}$ and ps_0 maps all other variables to $\textsc{any}$. So we are interested in

$$(\mathsf{FIX}\ H)\ (ps_0[\mathrm{y} \mapsto \textsc{pos}])$$

where

$$H\ h = \mathsf{cond}_D^{\{\mathtt{x},\mathtt{y}\}}(\mathcal{DB}[\![\neg(\mathtt{x} \leq \mathtt{1})]\!], h\ \circ\ h_{\mathrm{fac}}, \mathsf{id})$$

and

$$h_{\mathrm{fac}} = \mathcal{DS}[\![\mathtt{y} := \mathtt{y}{\star}\mathtt{x}; \mathtt{x} := \mathtt{x} - \mathtt{1}]\!]$$

In the case where ps $\mathtt{x} = p \in \{\textsc{pos}, \textsc{any}\}$ and ps $\mathtt{y} = \textsc{pos}$, we have

$$\begin{array}{rcl} H\ h\ ps & = & (h \circ h_{\mathrm{fac}})(\bigsqcup_{PS} \mathsf{filter}_T^{\{\mathtt{x},\mathtt{y}\}}(\mathcal{DB}[\![\neg(\mathtt{x} \leq \mathtt{1})]\!], ps)) \\ & & \quad \sqcup_{PS}\mathsf{id}(\bigsqcup_{PS} \mathsf{filter}_F^{\{\mathtt{x},\mathtt{y}\}}(\mathcal{DB}[\![\neg(\mathtt{x} \leq \mathtt{1})]\!], ps)) \\ & = & (h \circ h_{\mathrm{fac}})(ps[\mathtt{x} \mapsto \textsc{pos}]) \\ & & \quad \sqcup_{PS}(ps[\mathtt{x} \mapsto p]) \\ & = & h(ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} ps \end{array}$$

We can now compute the iterands $H^n \perp ps$ as follows when ps $\mathtt{x} = p \in \{\textsc{pos}, \textsc{any}\}$ and ps $\mathtt{y} = \textsc{pos}$:

$$\begin{array}{rcl} H^0 \perp ps & = & \textsc{init} \\ H^1 \perp ps & = & H^0 \perp (ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} (ps[\mathtt{x} \mapsto p]) \\ & = & ps \\ H^2 \perp ps & = & H^1 \perp (ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} (ps[\mathtt{x} \mapsto p]) \\ & = & ps[\mathtt{x} \mapsto \textsc{any}] \\ H^3 \perp ps & = & H^2 \perp (ps[\mathtt{x} \mapsto \textsc{any}]) \sqcup_{PS} (ps[\mathtt{x} \mapsto p]) \\ & = & ps[\mathtt{x} \mapsto \textsc{any}] \end{array}$$

One can show that for all n ≥ 2

$$H^{\mathrm{n}} \perp ps = ps[\mathtt{x} \mapsto \textsc{any}]$$

when $ps\ \mathtt{x} \in \{\textsc{pos}, \textsc{any}\}$ and $ps\ \mathtt{y} = \textsc{pos}$. It then follows that

$$(\mathsf{FIX}\ H)(ps_0[\mathtt{y} \mapsto \textsc{pos}]) = ps_0[\mathtt{x} \mapsto \textsc{any}][\mathtt{y} \mapsto \textsc{pos}]$$

So the analysis tells us that if `x` is positive in the initial state, then `y` will be positive in the final state (provided that the program terminates). □

Exercise 7.8 (Essential)

Show that if H: $(\mathbf{PState} \rightarrow \mathbf{PState}) \rightarrow (\mathbf{PState} \rightarrow \mathbf{PState})$ is continuous (or just monotone) and

$$H^{\mathrm{n}}\ \bot = H^{\mathrm{n}+1}\ \bot$$

for some n, then $H^{\mathrm{n}+\mathrm{m}}\ \bot = H^{1+\mathrm{m}}\ \bot$ for all $\mathrm{m} > 0$. Conclude that

$$H^{\mathrm{n}}\ \bot = H^{\mathrm{m}}\ \bot$$

for $\mathrm{m} \geq \mathrm{n}$ and therefore $\mathsf{FIX}\ H = H^{\mathrm{n}}\ \bot$.

Show by means of an example that $H^{\mathrm{n}}\ \bot\ ps_0 = H^{\mathrm{n}+1}\ \bot\ ps_0$ for some $ps_0 \in \mathbf{PState}$ does not necessarily imply that $\mathsf{FIX}\ H\ ps_0 = H^{\mathrm{n}}\ \bot\ ps_0$. □

Exercise 7.9

A *constant propagation* analysis attempts to predict whether arithmetic and boolean expressions always evaluate to constant values. For natural numbers, the following properties are of interest:

– $\textsc{any}$: it can be any number,

– z: it is definitely the number $z \in \mathbf{Z}$ and

– $\textsc{none}$: the number belongs to $\emptyset$

Let now $\mathbf{Const} = \mathbf{Z} \cup \{\textsc{any}, \textsc{none}\}$ and let $\sqsubseteq_C$ be the ordering defined by

– $\textsc{none} \sqsubseteq_C p$ for all $p \in \mathbf{Const}$, and

– $p \sqsubseteq_C \textsc{any}$ for all $p \in \mathbf{Const}$

All other elements are incomparable. Draw the Hasse diagram for $(\mathbf{Const}, \sqsubseteq_C)$.

Let $\mathbf{PState} = \mathbf{Var} \rightarrow \mathbf{Const}$ be the set of property states and let $\mathbf{TT}$ be the properties of the truth values. Specify the constant propagation analysis by defining the functionals

$$\mathcal{CA}\text{: } \mathbf{Aexp} \rightarrow \mathbf{PState} \rightarrow \mathbf{Const}$$

$$\mathcal{CB}\text{: } \mathbf{Bexp} \rightarrow \mathbf{PState} \rightarrow \mathbf{TT}$$

$$\mathcal{CS}\text{: } \mathbf{Stm} \rightarrow \mathbf{PState} \rightarrow \mathbf{PState}$$

Be careful to specify the auxiliary operations in detail. Give a couple of examples illustrating the power of the analysis. □

7.2 Detection of Signs Analysis: Existence

Having specified the detection of signs analysis, we shall now show that it is indeed well-defined. We proceed in three stages:

– First we introduce a partial order on **PState** → **PState** such that it becomes a ccpo.

– Then we show that certain auxiliary functions used in the definition of $\mathcal{DS}$ are continuous.

– Finally we show that the fixed point operator is applied only to continuous functions.

The ccpo

Our first task is to define a partial order on **PState** → **PState**, and for this we use the approach developed in Lemma 7.1. Instantiating the non-empty set S to the set **PState** and the partially ordered set $(D, \sqsubseteq)$ to $(\mathbf{PState}, \sqsubseteq_{DS})$, we get the following corollary.

Corollary 7.10

Let $\sqsubseteq$ be the ordering on **PState** → **PState** defined by

$$h_1 \ \sqsubseteq\ h_2 \text{ if and only if } h_1\ ps\ \sqsubseteq_{PS}\ h_2\ ps \text{ for all } ps\ \in \mathbf{PState}.$$

Then $(\mathbf{PState} \to \mathbf{PState}, \sqsubseteq)$ is a complete lattice, and hence a ccpo, and the formula for least upper bounds is

$$(\bigsqcup Y)\ ps = \bigsqcup_{PS} \{h\ ps \mid h \in Y\}$$

for all subsets Y of **PState** → **PState**.

Auxiliary Functions

Our second task is to ensure that the function H used in Table 7.5 is a continuous function from **PState** → **PState** to **PState** → **PState**. For this, we follow the approach of Section 5.3 and show that cond_D is continuous in its second argument and later that composition is continuous in its first argument.

Lemma 7.11

Let f: $\mathbf{PState} \to \mathbf{TT}$, h_0: $\mathbf{PState} \to \mathbf{PState}$ and define

$$H\ h = \mathsf{cond}_D(f,\ h,\ h_0)$$

Then H: $(\mathbf{PState} \to \mathbf{PState}) \to (\mathbf{PState} \to \mathbf{PState})$ is a continuous function.

Proof: We shall first prove that H is *monotone*, so let h_1 and h_2 be such that $h_1 \sqsubseteq h_2$; that is, $h_1\ ps \sqsubseteq_{PS} h_2\ ps$ for all property states ps. We then have to show that $\mathsf{cond}_D(f,\ h_1,\ h_0)\ ps \sqsubseteq_{PS} \mathsf{cond}_D(f,\ h_2,\ h_0)\ ps$ for all property states ps. The proof is by cases on the value of $f\ ps$. If $f\ ps = \textsc{none}$, then the result trivially holds. If $f\ ps = \textsc{tt}$, then the result follows from the assumption

$$h_1\ ps \sqsubseteq_{PS} h_2\ ps$$

If $f\ ps = \textsc{ff}$, then the result trivially holds. If $f\ ps = \textsc{any}$, then the result follows from

$$(h_1\ ps \sqcup_{PS} h_0\ ps) \sqsubseteq_{PS} (h_2\ ps \sqcup_{PS} h_0\ ps)$$

which again follows from the assumption $h_1\ ps \sqsubseteq_{PS} h_2\ ps$.

To see that H is *continuous*, let Y be a non-empty chain in $\mathbf{PState} \to \mathbf{PState}$. Using the characterization of least upper bounds in $\mathbf{PState}$ given in Corollary 7.10, we see that we must show that

$$(H(\bigsqcup Y))\ ps = \bigsqcup_{PS} \{(H\ h)ps \mid h \in Y\}$$

for all property states ps in $\mathbf{PState}$. The proof is by cases on the value of $f\ ps$. If $f\ ps = \textsc{none}$, then we have $(H\ (\bigsqcup Y))\ ps = \textsc{init}$ and

$$\begin{aligned}\bigsqcup_{PS} \{(H\ h)ps \mid h \in Y\} &= \bigsqcup_{PS} \{\textsc{init} \mid h \in Y\}\\ &= \textsc{init}\end{aligned}$$

Thus we have proved the required result in this case. If $f\ ps = \textsc{tt}$, then we have

$$\begin{aligned}(H(\bigsqcup Y))ps &= (\bigsqcup Y)ps\\ &= (\bigsqcup_{PS}\{h\ ps \mid h \in Y\})\end{aligned}$$

using the characterization of least upper bounds and furthermore

$$\bigsqcup_{PS}\{(H\ h)ps \mid h \in Y\} = \bigsqcup_{PS}\{h\ ps \mid h \in Y\}$$

and the result follows. If $f\ ps = \textsc{ff}$, then we have

$$(H(\bigsqcup Y))ps = h_0\ ps$$

and

$$\begin{aligned}\textstyle\bigsqcup_{PS}\{(H\ h)ps \mid h \in Y\} &= \textstyle\bigsqcup_{PS}\{h_0\ ps \mid h \in Y\}\\ &= h_0\ ps\end{aligned}$$

where the last equality follows because Y is non-empty. If $f\ ps$ = ANY, then the characterization of least upper bounds in **PState** gives

$$\begin{aligned}(H(\bigsqcup Y))ps &= ((\bigsqcup Y)ps) \sqcup_{PS} (h_0\ ps)\\ &= (\textstyle\bigsqcup_{PS}\{h\ ps \mid h \in Y\}) \sqcup_{PS} (h_0\ ps)\\ &= \textstyle\bigsqcup_{PS}\{h\ ps \mid h \in Y \cup \{h_0\}\}\end{aligned}$$

and

$$\begin{aligned}\textstyle\bigsqcup_{PS}\{(H\ h)ps \mid h \in Y\} &= \textstyle\bigsqcup_{PS}\{(h\ ps) \sqcup_{PS} (h_0\ ps) \mid h \in Y\}\\ &= \textstyle\bigsqcup_{PS}\{h\ ps \mid h \in Y \cup \{h_0\}\}\end{aligned}$$

where the last equality follows because Y is non-empty. Thus we have established that H is continuous. □

Exercise 7.12

Let f: **PState** → **TT**, h_0: **PState** → **PState**, and define

$$H\ h = \mathsf{cond}_D(f,\ h_0,\ h)$$

Show that H: (**PState** → **PState**) → (**PState** → **PState**) is a continuous function. □

Lemma 7.13

Let h_0: **PState** → **PState** and define

$$H\ h = h\ \circ\ h_0$$

Then H: (**PState** → **PState**) → (**PState** → **PState**) is a continuous function.

Proof: We shall first show that H is *monotone*, so let h_1 and h_2 be such that $h_1 \sqsubseteq h_2$; that is, $h_1\ ps \sqsubseteq_{PS} h_2\ ps$ for all property states ps. Clearly we then have $h_1(h_0\ ps) \sqsubseteq_{PS} h_2(h_0\ ps)$ for all property states ps and thereby we have proved the monotonicity of H.

To prove the *continuity* of H, let Y be a non-empty chain in **PState** → **PState**. We must show that

$$(H(\bigsqcup Y))ps = (\bigsqcup\{H\ h \mid h \in Y\})ps$$

for all property states ps. Using the characterization of least upper bounds given in Corollary 7.10, we get

$$\begin{aligned}(H(\bigsqcup Y))ps &= ((\bigsqcup Y)\ \circ\ h_0)ps\\ &= (\bigsqcup Y)(h_0\ ps)\\ &= \bigsqcup_{PS}\{h(h_0\ ps) \mid h \in Y\}\end{aligned}$$

and

$$\begin{aligned}(\bigsqcup\{H\ h \mid h \in Y\})ps &= \bigsqcup_{PS}\{(H\ h)ps \mid h \in Y\}\\ &= \bigsqcup_{PS}\{(h\ \circ\ h_0)ps \mid h \in Y\}\end{aligned}$$

Hence the result follows. □

Exercise 7.14

Show that there exists h_0: **PState** $\rightarrow$ **PState** such that H defined by $H\ h = h_0\ \circ\ h$ is *not even* a monotone function from **PState** $\rightarrow$ **PState** to **PState** $\rightarrow$ **PState**. □

Remark The example of the exercise above indicates a major departure from the secure world of Chapter 5. Luckily an insurance policy can be arranged. The premium is to replace all occurrences of

PState $\rightarrow$ **PState**, **PState** $\rightarrow$ **Sign**, and **PState** $\rightarrow$ **TT**

by

[**PState** $\rightarrow$ **PState**], [**PState** $\rightarrow$ **Sign**], and [**PState** $\rightarrow$ **TT**]

where $[D\ \rightarrow\ E] = \{f : D\ \rightarrow\ E \mid f \text{ is continuous}\}$. One can then show that $[D\ \rightarrow\ E]$ is a ccpo if D and E are and that the characterization of least upper bounds given in Lemma 7.1 still holds. Furthermore, the entire development in this section still carries through although there are additional proof obligations to be carried out. In this setting, one gets that if h_0: [**PState** $\rightarrow$ **PState**], then H defined by $H\ h = h_0\ \circ\ h$ is a continuous function from [**PState** $\rightarrow$ **PState**] to [**PState** $\rightarrow$ **PState**].

Well-definedness

First we note that the equations of Tables 7.1 and 7.3 define total functions $\mathcal{DA}$ and $\mathcal{DB}$ in **Aexp** $\rightarrow$ **PState** $\rightarrow$ **Sign** and **Bexp** $\rightarrow$ **PState** $\rightarrow$ **TT**, respectively. For the well-definedness of $\mathcal{DS}$, we have the following proposition.

Proposition 7.15

The function $\mathcal{DS}$: **Stm** → **PState** → **PState** of Table 7.5 is a well-defined function.

Proof: We prove by structural induction on S that $\mathcal{DS}[\![S]\!]$ is well-defined and only the case of the `while`-loop is interesting. We note that the function H used in Table 7.5 is given by

$$H = H_1 \circ H_2$$

where

$$H_1\ h = \mathsf{cond}_D(\mathcal{DB}[\![b]\!], h, \mathsf{id})$$

and

$$H_2\ h = h \circ \mathcal{DS}[\![S]\!]$$

Since H_1 and H_2 are continuous functions by Lemmas 7.11 and 7.13, we have that H is a continuous function by Lemma 5.35. Hence $\mathsf{FIX}\ H$ is well-defined and this completes the proof. □

To summarize, the well-definedness of $\mathcal{DS}$ relies on the following results established above:

Proof Summary for While:

Well-definedness of Program Analysis

1: The set **PState** → **PState** equipped with an appropriate ordering $\sqsubseteq$ is a ccpo (Corollary 7.10).

2: Certain functions Ψ: (**PState** → **PState**) → (**PState** → **PState**) are continuous (Lemmas 7.11 and 7.13).

3: In the definition of $\mathcal{DS}$, we only apply the fixed point operation to continuous functions (Proposition 7.15).

Exercise 7.16

Extend **While** with the statement `repeat` S `until` b and give the new (compositional) clause for $\mathcal{DS}$. Validate the well-definedness of the extended version of $\mathcal{DS}$. □

Exercise 7.17

Show that the constant propagation analysis specified in Exercise 7.9 is indeed well-defined. □

7.3 Safety of the Analysis

In this section, we shall show that the analysis functions $\mathcal{DA}$, $\mathcal{DB}$, and $\mathcal{DS}$ are safe with respect to the semantic functions $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{S}_{ds}$. We begin with the rather simple case of expressions.

Expressions

Let g: **State** $\to$ **Z** be a function, perhaps of the form $\mathcal{A}[\![a]\!]$ for some arithmetic expression $a \in$ **Aexp**, and let h: **PState** $\to$ **Sign** be another function, perhaps of the form $\mathcal{DA}[\![a]\!]$ for some arithmetic expression $a \in$ **Aexp**. We shall introduce a relation

$$g \ \mathsf{safe}_A \ h$$

for expressing when the analysis h is *safe with respect to* the semantics g. It is defined by

$$\mathsf{abs}(s) \sqsubseteq_{PS} \ ps \qquad \text{implies} \qquad \mathsf{abs}_Z(g\ s) \sqsubseteq_S \ h\ ps$$

for all states s and property states ps. Thus the predicate says that if ps describes the sign of the variables of s, then the sign of $g\ s$ will be described by $h\ ps$.

Exercise 7.18 (Essential)

Prove that for all $a \in$ **Aexp** we have $\mathcal{A}[\![a]\!]$ $\mathsf{safe}_A \mathcal{DA}[\![a]\!]$. □

To express the safety of the analysis of boolean expressions, we shall introduce a relation

$$g \ \mathsf{safe}_B \ h$$

for expressing when the analysis h: **PState** $\to$ **TT** is *safe with respect to* the semantics g: **State** $\to$ **T**. It is defined by

$$\mathsf{abs}(s) \sqsubseteq_{PS} \ ps \qquad \text{implies} \qquad \mathsf{abs}_T(g\ s) \sqsubseteq_T \ h\ ps$$

for all states s and property states ps. We have the following exercise.

Exercise 7.19 (Essential)

Prove that for all $b \in \mathbf{Bexp}$ we have $\mathcal{B}[\![b]\!]\ \mathsf{safe}_B\ \mathcal{DB}[\![b]\!]$. $\square$

Statements

The safety of the analysis of statements will express that if the signs of the initial state are described by some property state, then the signs of the final state (if ever reached) will be described by the property state obtained by applying the analysis to the initial property state. We shall formalize this by defining a relation

$$g\ \mathsf{safe}_S\ h$$

between a function g: $\mathbf{State} \hookrightarrow \mathbf{State}$, perhaps of the form $\mathcal{S}_{ds}[\![S]\!]$ for some $S \in \mathbf{Stm}$, and another function h: $\mathbf{PState} \to \mathbf{PState}$, perhaps of the form $\mathcal{DS}[\![S]\!]$ for some $S \in \mathbf{Stm}$. The formal definition amounts to

$$\mathsf{abs}(s) \sqsubseteq_{PS} ps \text{ and } g\ s \neq \underline{\text{undef}} \qquad \text{imply} \qquad \mathsf{abs}(g\ s) \sqsubseteq_{PS} h\ ps$$

for all states $s \in \mathbf{State}$ and all property states $ps \in \mathbf{PState}$.

We may then formulate the desired relationship between the semantics and the analysis as follows.

Theorem 7.20

For all statements S of **While**: $\mathcal{S}_{ds}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{DS}[\![S]\!]$.

Before conducting the proof, we need to establish some properties of the auxiliary operations composition and conditional.

Lemma 7.21

Let g_1, g_2: $\mathbf{State} \hookrightarrow \mathbf{State}$ and h_1, h_2: $\mathbf{PState} \to \mathbf{PState}$. Then

$$g_1\ \mathsf{safe}_S\ h_1 \text{ and } g_2\ \mathsf{safe}_S\ h_2 \quad \text{imply} \quad g_2 \circ g_1\ \mathsf{safe}_S\ h_2 \circ h_1$$

Proof: Let s and ps be such that

$$\mathsf{abs}(s) \sqsubseteq_{PS} ps \text{ and } (g_2 \circ g_1)s \neq \underline{\text{undef}}$$

Then $g_1\ s \neq \underline{\text{undef}}$ must be the case and from the assumption g_1 safe_S h_1 we then get

$$\mathsf{abs}(g_1\ s) \sqsubseteq_{PS}\ h_1\ ps$$

Since $g_2\ (g_1\ s) \neq \underline{\text{undef}}$, we use the assumption g_2 safe_S h_2 to get

$$\mathsf{abs}(g_2\ (g_1\ s)) \sqsubseteq_{PS}\ h_2(h_1\ ps)$$

and we have completed the proof. □

Lemma 7.22

Assume that g_1, g_2: **State** $\hookrightarrow$ **State** and g: **State** $\rightarrow$ **T**, and that h_1, h_2: **PState** $\rightarrow$ **PState** and f: **PState** $\rightarrow$ **TT**. Then

$$g\ \mathsf{safe}_B\ f,\ g_1\ \mathsf{safe}_S\ h_1 \text{ and } g_2\ \mathsf{safe}_S\ h_2 \quad \text{imply}$$
$$\mathsf{cond}(g,\ g_1,\ g_2)\ \mathsf{safe}_S\ \mathsf{cond}_D(f,\ h_1,\ h_2)$$

Proof: Let s and ps be such that

$$\mathsf{abs}(s) \sqsubseteq_{PS}\ ps \text{ and } \mathsf{cond}(g, g_1, g_2)\ s \neq \underline{\text{undef}}$$

We shall now proceed by a case analysis on $g\ s$. First assume that $g\ s = \mathbf{tt}$. It must be the case that $g_1\ s \neq \underline{\text{undef}}$, so from g_1 safe_S h_1 we get

$$\mathsf{abs}(g_1\ s) \sqsubseteq_{PS}\ h_1\ ps$$

From g safe_B f, we get that

$$\mathsf{abs}_T(g\ s) \sqsubseteq_T\ f\ ps$$

and thereby $\text{TT} \sqsubseteq_T f\ ps$. Since $h_1\ ps\ \sqsubseteq_{PS}\ h_1\ ps \sqcup_{PS} h_2\ ps$, we get the required result both when $f\ ps = \text{TT}$ and when $f\ ps = \text{ANY}$. The case where $g\ s = \mathbf{ff}$ is similar. □

We now have the apparatus needed to show the safety of $\mathcal{DS}$.

Proof of Theorem 7.20: We shall show that $\mathcal{S}_{ds}[\![S]\!]$ safe_S $\mathcal{DS}[\![S]\!]$, and we proceed by structural induction on the statement S.

The case $x := a$: Let s and ps be such that

$$\mathsf{abs}(s)\ \sqsubseteq_{PS}\ ps \text{ and } \mathcal{S}_{ds}[\![x := a]\!]s \neq \underline{\text{undef}}$$

We have to show that

$$\mathsf{abs}(\mathcal{S}_{ds}[\![x := a]\!]s)\ \sqsubseteq_{PS}\ \mathcal{DS}[\![x := a]\!]ps$$

that is,

$$\mathsf{abs}_Z((\mathcal{S}_{ds}[\![x := a]\!]s)y) \;\sqsubseteq_S\; (\mathcal{DS}[\![x := a]\!]ps)y$$

for all $y \in$ **Var**. If $y \neq x$, then it is immediate from the assumption $\mathsf{abs}(s) \sqsubseteq_{PS} ps$. If $y = x$, then we use that Exercise 7.18 gives

$$\mathsf{abs}(\mathcal{A}[\![a]\!]s) \;\sqsubseteq_S\; \mathcal{DA}[\![a]\!]ps$$

Hence we have the required relationship.

The case `skip`: Straightforward.

The case $S_1;S_2$: The induction hypothesis applied to S_1 and S_2 gives

$$\mathcal{S}_{ds}[\![S_1]\!] \ \mathsf{safe}_S\ \mathcal{DS}[\![S_1]\!]$$

and

$$\mathcal{S}_{ds}[\![S_2]\!] \ \mathsf{safe}_S\ \mathcal{DS}[\![S_2]\!]$$

The desired result

$$\mathcal{S}_{ds}[\![S_2]\!] \;\circ\; \mathcal{S}_{ds}[\![S_1]\!] \ \mathsf{safe}_S\ \mathcal{DS}[\![S_2]\!] \;\circ\; \mathcal{DS}[\![S_1]\!]$$

follows directly from Lemma 7.21.

The case `if` b `then` S_1 `else` S_2: From Exercise 7.19, we have

$$\mathcal{B}[\![b]\!] \ \mathsf{safe}_B\ \mathcal{DB}[\![b]\!]$$

and the induction hypothesis applied to S_1 and S_2 gives

$$\mathcal{S}_{ds}[\![S_1]\!] \ \mathsf{safe}_S\ \mathcal{DS}[\![S_1]\!]$$

and

$$\mathcal{S}_{ds}[\![S_2]\!] \ \mathsf{safe}_S\ \mathcal{DS}[\![S_2]\!]$$

The desired result

$$\mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!]) \quad \mathsf{safe}_S \quad \mathsf{cond}_D(\mathcal{DB}[\![b]\!], \mathcal{DS}[\![S_1]\!], \mathcal{DS}[\![S_2]\!])$$

then follows directly from Lemma 7.22.

The case `while` b `do` S: We must prove that

$$\mathsf{FIX}(G) \ \mathsf{safe}_S\ \mathsf{FIX}(H)$$

where

$$G\ g = \mathsf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], \mathsf{id})$$

and

$$H\ h = \mathsf{cond}_D(\mathcal{DB}[\![b]\!], h \circ \mathcal{DS}[\![S]\!], \mathsf{id})$$

To do this, we recall from Chapter 5 the definition of the least fixed points:

$$\mathsf{FIX}\ G = \bigsqcup \{\, G^{\mathrm{n}}\, g_0 \mid \mathrm{n}\ \geq 0\} \text{ where } g_0\ s = \underline{\mathrm{undef}} \text{ for all } s$$

and

$$\mathsf{FIX}\ H = \bigsqcup\{\, H^{\rm n}\, h_0 \mid {\rm n}\ \geq 0\} \text{ where } h_0\ ps = \textsc{init} \text{ for all } ps$$

The proof proceeds in two stages. We begin by proving that

$$G^{\rm n}\ g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H \text{ for all n} \tag{*}$$

and then continue by proving that

$$\mathsf{FIX}\ G\ \mathsf{safe}_S\ \mathsf{FIX}\ H \tag{**}$$

We prove (*) by induction on n. For the base case, we observe that

$$g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

holds trivially since $g_0\ s = \underline{\rm undef}$ for all states s. For the induction step, we assume that

$$G^{\rm n}\ g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

and we shall prove the result for ${\rm n}+1$. We have

$$\mathcal{B}[\![b]\!]\ \mathsf{safe}_B\ \mathcal{DB}[\![b]\!]$$

from Exercise 7.19 and

$$\mathcal{S}_{ds}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{DS}[\![S]\!]$$

from the induction hypothesis applied to the body of the `while`-loop, and it is clear that

$$\mathsf{id}\ \mathsf{safe}_S\ \mathsf{id}$$

We then obtain

$$\mathsf{cond}(\mathcal{B}[\![b]\!], (G^{\rm n}\ g_0)\circ\mathcal{S}_{ds}[\![S]\!], \mathsf{id}) \quad \mathsf{safe}_S \quad \mathsf{cond}_D(\mathcal{DB}[\![b]\!], (\mathsf{FIX}\ H)\circ\mathcal{DS}[\![S]\!], \mathsf{id})$$

from Lemmas 7.21 and 7.22, and this is indeed the desired result since the right-hand side amounts to H ($\mathsf{FIX}\ H$), which equals $\mathsf{FIX}\ H$.

Finally, we must show (**). This amounts to showing

$$(\bigsqcup Y)\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

where $Y = \{G^{\rm n} g_0 \mid {\rm n}\ \geq 0\}$. So assume that

$$\mathsf{abs}(s)\ \sqsubseteq_{PS}\ ps \text{ and } (\bigsqcup Y)\ s \neq\ \underline{\rm undef}$$

By Lemma 5.25, we have

$$\mathsf{graph}(\bigsqcup Y) = \bigcup\{\,\mathsf{graph}\ g \mid g\in Y\}$$

and $(\bigsqcup Y)\ s \neq\ \underline{\rm undef}$ therefore shows the existence of an element g in Y such that $g\ s \neq\ \underline{\rm undef}$ and $(\bigsqcup Y)s = g\ s$. Since $g\ \mathsf{safe}_S\ \mathsf{FIX}\ H$ holds for all $g \in Y$, by (*) we get that

$$\mathsf{abs}(g\ s) \sqsubseteq_{PS} (\mathsf{FIX}\ H)\ ps$$

and therefore $\mathsf{abs}((\bigsqcup Y)s) \sqsubseteq_{PS} (\mathsf{FIX}\ H)\ ps$ as required. □

The proof of safety of the analysis can be summarized as follows:

Proof Summary for While:

Safety of Program Analysis

1: Define a relation safe_S expressing the relationship between the functions of **State** $\hookrightarrow$ **State** and **PState** $\rightarrow$ **PState**.

2: Show that the relation is preserved by certain pairs of auxiliary functions used in the denotational semantics and the static analysis (Lemmas 7.21 and 7.22).

3: Use *structural induction* on the statements S to show that the relation holds between the semantics and the analysis of S.

Exercise 7.23

Extend the proof of Theorem 7.20 to incorporate the safety of the analysis developed for `repeat` S `until` b in Exercise 7.16. □

Exercise 7.24

Prove that the constant propagation analysis specified in Exercise 7.9 is safe. That is, show that

$$\mathcal{A}[\![a]\!]\ \mathsf{safe}_A\ \mathcal{CA}[\![a]\!]$$

$$\mathcal{B}[\![b]\!]\ \mathsf{safe}_B\ \mathcal{CB}[\![b]\!]$$

$$\mathcal{S}_{ds}[\![a]\!]\ \mathsf{safe}_S\ \mathcal{CS}[\![S]\!]$$

for appropriately defined predicates safe_A, safe_B, and safe_S. □

7.4 Program Transformation

The detection of signs analysis can be used to predict the values of tests in conditionals and loops and thereby may be used to facilitate certain program transformations. An example program transformation is

Replace	$\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$
by	S_1
when	$\mathcal{DB}[\![b]\!]\ \textsc{top} = \textsc{tt}$

where we have written TOP for the property state that maps all variables to ANY. The condition $\mathcal{DB}[\![b]\!]\ \textsc{top} = \textsc{tt}$ will only be satisfied when $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ for all states s, so the transformation can be used rather seldom. Other transformations take the *context* into account, and we may use the property states to describe the contexts. So we may extend the format above to:

In context	ps
replace	$\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$
by	S_1
when	$\mathcal{DB}[\![b]\!]ps = \textsc{tt}$

We shall formalise these transformations as a transition system. The transitions have the form

$$ps \vdash S \ \rightsquigarrow\ S'$$

meaning that in the context described by ps the statement S can be replaced by S'. So the transformation above can be formulated as

$$ps \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \ \rightsquigarrow\ S_1, \qquad \text{if } \mathcal{DB}[\![b]\!]ps = \textsc{tt}$$

where the side condition expresses when the transformation is applicable. The dual transformation is

$$ps \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \ \rightsquigarrow\ S_2, \qquad \text{if } \mathcal{DB}[\![b]\!]ps = \textsc{ff}$$

We might also want to transform inside composite statements such as $S_1; S_2$. This is expressed by the rule

$$\frac{ps \vdash S_1 \ \rightsquigarrow\ S_1', \quad (\mathcal{DS}[\![S_1]\!]ps) \vdash S_2 \ \rightsquigarrow\ S_2'}{ps \vdash S_1; S_2 \ \rightsquigarrow\ S_1'; S_2'}$$

Combined with the trivial transformation

$$ps \vdash S \ \rightsquigarrow\ S$$

this opens up for the possibility of only transforming parts of the statement.

In general, a transformation $ps \vdash S \ \rightsquigarrow\ S'$ is (weakly) *valid* if

$$\mathsf{abs}(s) \sqsubseteq_{PS} ps \text{ and } \mathcal{S}_{ds}[\![S]\!]s \neq \underline{\text{undef}} \quad \text{imply} \quad \mathcal{S}_{ds}[\![S]\!]s = \mathcal{S}_{ds}[\![S']\!]s$$

for all states s. This is a *weak* notion of validity because a transformation such as

$$\textsc{top} \vdash \texttt{while true do skip} \ \rightsquigarrow\ \texttt{skip}$$

is valid even though it allows us to replace a looping statement with one that always terminates.

Lemma 7.25

The transformation

$$ps \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \ \rightsquigarrow\ S_1$$

is valid provided that $\mathcal{DB}[\![b]\!]ps = \text{TT}$.

Proof

Assume that

$$\mathsf{abs}(s) \ \sqsubseteq_{PS}\ ps \text{ and } \mathcal{S}_{ds}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]s \neq \underline{\text{undef}}.$$

From $\mathcal{DB}[\![b]\!]ps = \text{TT}$, $\mathsf{abs}(s) \sqsubseteq_{PS} ps$, and Exercise 7.19, we get that

$$\mathsf{abs}_T(\mathcal{B}[\![b]\!]s) \sqsubseteq_T \text{TT}$$

and thereby $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ since $\mathcal{B}[\![b]\!]$ is a total function. From the definition of $\mathcal{S}_{ds}$ in Chapter 5, we get

$$\mathcal{S}_{ds}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]s = \mathcal{S}_{ds}[\![S_1]\!]s$$

and this is the required result. □

Exercise 7.26

Prove that the transformation rule for $S_1; S_2$ is valid. □

Exercise 7.27

Prove that the transformation rule

$$\frac{ps \vdash S_1 \ \rightsquigarrow\ S_1', \qquad (\mathcal{DS}[\![S_1']\!]ps) \vdash S_2 \ \rightsquigarrow\ S_2'}{ps \vdash S_1; S_2 \ \rightsquigarrow\ S_1'; S_2'}$$

is valid. Note that it differs from the rule given earlier in the second premise, where the transformed statement is analysed rather than the original. □

Exercise 7.28

Suggest a transformation for assignment and prove it is valid. □

Exercise 7.29

Suggest a transformation rule that allows transformations inside the branches of a conditional and prove that it is valid. □

Exercise 7.30 (**)

Try to develop a transformation rule that allows transformations inside the body of a `while`-loop. □

8
More on Program Analysis

Program analyses are often classified in two respects:

– *what* kind of properties do they compute with, and

– *how* do they compute with them.

Basically, there are two kinds of properties:

– properties of *values* and

– properties of *relationships* between values.

The first class of analyses is often called *first-order analyses*, as they compute with direct properties of the values. The second class of analyses is called *second-order analyses*, as they compute with properties derived from relationships between values.

The *detection of signs* analysis of Chapter 7 is a first-order analysis. The properties of interest are the signs of values, so rather than computing with numbers, we compute with the sign properties POS, ZERO, NEG, etc. Constant propagation and interval analysis are also first-order analyses.

A *live variables analysis* (see Chapter 7) is a second-order analysis. Here the properties associated with the variables are LIVE and DEAD, and obviously this does not say much about the "real" value of the variables. However, it expresses a property of the relationship between "real" values: if the property is DEAD, then the variable could have any value whatsoever — the result of the computation would be the same since the value will never be used. On the other hand, if the property is LIVE, then the "real" value of the variable might influence the final outcome of the computation. Detection of common subexpressions, available subexpression analysis, and the security analysis of Section 8.2 are other second-order analyses.

The other classification is concerned with *how* the analyses compute with the properties. Again there are basically two approaches (although mixtures exist):

– forward analyses and

– backward analyses.

In a *forward analysis*, the computation proceeds much as in the direct style denotational semantics: given the properties of the input, the analysis will compute properties of the output. Clearly the *detection of signs analysis* of Chapter 7 proceeds in this way.

As the name suggests, a *backward analysis* performs the computation the other way around: given properties of the output of the computation, it will predict the properties that the input should have. Here *live variable analysis* is a classical example: the output certainly should have the property LIVE, and the idea is then to calculate backward to see which parts of the input (and which parts of the intermediate results) really are needed in order to compute the output.

The two classifications can be freely mixed so, for example, the *detection of signs analysis* is a forward first-order analysis and *live variables analysis* is a backward second-order analysis. An example of a backward first-order analysis is an *error detection analysis*: we might want to ensure that the program does not result in an error (say, from an attempt to divide by zero), and the analysis could then provide information about inputs that definitely would prevent this from happening. This kind of information will be crucial for the design of safety-critical systems. An example of a forward second-order analysis is the *security analysis* to be developed in Section 8.2. Properties LOW and HIGH indicate whether data should be regarded as public (i.e., having low confidentiality) or private (i.e., having high confidentiality). The aim of the analysis is to ensure that the computation of public data does not (either *directly* or *indirectly*) depend on private data; if this were to be the case, the program would exhibit a "covert channel" and should be considered insecure. This kind of information will be crucial for the design of secure systems.

8.1 Data Flow Frameworks

Many forward program analyses can be seen as a slight variation of the detection of signs analysis: the properties and the way we compute with them may be different but the overall approach will be the same. In this section, we shall extract the overall approach and present it as a *general framework* where only a few parameters need to be specified in order to obtain the desired analysis. Similar remarks apply to backward analyses.

In general, the *specification* of a program analysis falls into two parts. First we introduce the properties with which the analysis computes, and next we specify the actual analysis for the three syntactic categories of **While**.

Properties and Property States

The *basic properties* are those of the numbers and the truth values. So the first step in specifying an analysis will be to define

– P1: a complete lattice of properties of $\mathbf{Z}$: $(\mathbf{P}_Z, \sqsubseteq_Z)$

– P2: a complete lattice of properties of $\mathbf{T}$: $(\mathbf{P}_T, \sqsubseteq_T)$

Example 8.1

In the case of the detection of signs analysis, we have $\mathbf{P}_Z = \mathbf{Sign}$ and $\mathbf{P}_T = \mathbf{TT}$. $\square$

The *property states* will then be defined as

$$\mathbf{PState} = \mathbf{Var} \to \mathbf{P}_Z$$

independently of the choice of $\mathbf{P}_Z$. The property states inherit the ordering of $\mathbf{P}_Z$ as indicated in Lemma 7.1 and will thus form a complete lattice. In particular, **PState** will have a least element, which we call $\textsc{init}$.

Forward Analysis

In a forward analysis, the computation proceeds much as in the direct style denotational semantics: given properties of the input, the analysis will compute properties of the output. Thus the idea will be to replace the semantic functions

$$\begin{aligned}
\mathcal{FA}\llbracket n \rrbracket ps &= \mathcal{FZ}\llbracket n \rrbracket ps \\
\mathcal{FA}\llbracket x \rrbracket ps &= ps\ x \\
\mathcal{FA}\llbracket a_1 + a_2 \rrbracket ps &= \mathsf{add}_F(\mathcal{FA}\llbracket a_1 \rrbracket ps, \mathcal{FA}\llbracket a_2 \rrbracket ps) \\
\mathcal{FA}\llbracket a_1 \star a_2 \rrbracket ps &= \mathsf{mult}_F(\mathcal{FA}\llbracket a_1 \rrbracket ps, \mathcal{FA}\llbracket a_2 \rrbracket ps) \\
\mathcal{FA}\llbracket a_1 - a_2 \rrbracket ps &= \mathsf{sub}_F(\mathcal{FA}\llbracket a_1 \rrbracket ps, \mathcal{FA}\llbracket a_2 \rrbracket ps) \\[1ex]
\mathcal{FB}\llbracket \texttt{true} \rrbracket ps &= \mathcal{FT}\llbracket \texttt{true} \rrbracket ps \\
\mathcal{FB}\llbracket \texttt{false} \rrbracket ps &= \mathcal{FT}\llbracket \texttt{false} \rrbracket ps \\
\mathcal{FB}\llbracket a_1 = a_2 \rrbracket ps &= \mathsf{eq}_F(\mathcal{FA}\llbracket a_1 \rrbracket ps, \mathcal{FA}\llbracket a_2 \rrbracket ps) \\
\mathcal{FB}\llbracket a_1 \leq a_2 \rrbracket ps &= \mathsf{leq}_F(\mathcal{FA}\llbracket a_1 \rrbracket ps, \mathcal{FA}\llbracket a_2 \rrbracket ps) \\
\mathcal{FB}\llbracket \neg b \rrbracket ps &= \mathsf{neg}_F(\mathcal{FB}\llbracket b \rrbracket ps) \\
\mathcal{FB}\llbracket b_1 \wedge b_2 \rrbracket ps &= \mathsf{and}_F(\mathcal{FB}\llbracket b_1 \rrbracket ps, \mathcal{FB}\llbracket b_2 \rrbracket ps)
\end{aligned}$$

Table 8.1 Forward analysis of expressions

$$\begin{aligned}
\mathcal{A} : \mathbf{Aexp} &\to \mathbf{State} \to \mathbf{Z} \\
\mathcal{B} : \mathbf{Bexp} &\to \mathbf{State} \to \mathbf{T} \\
\mathcal{S}_{ds} : \mathbf{Stm} &\to \mathbf{State} \hookrightarrow \mathbf{State}
\end{aligned}$$

with semantic functions that compute with properties rather than values:

$$\begin{aligned}
\mathcal{FA} : \mathbf{Aexp} &\to \mathbf{PState} \to \mathbf{P}_Z \\
\mathcal{FB} : \mathbf{Bexp} &\to \mathbf{PState} \to \mathbf{P}_T \\
\mathcal{FS} : \mathbf{Stm} &\to \mathbf{PState} \to \mathbf{PState}
\end{aligned}$$

The semantic functions $\mathcal{FA}$ and $\mathcal{FB}$ are defined in Table 8.1. Whenever the direct style semantics performs computations involving numbers or truth values, the analysis has to do something analogous depending on the actual choice of properties. We shall therefore assume that we have functions

– F1: add_F: $\mathbf{P}_Z \times \mathbf{P}_Z \to \mathbf{P}_Z$

– F2: mult_F: $\mathbf{P}_Z \times \mathbf{P}_Z \to \mathbf{P}_Z$

– F3: sub_F: $\mathbf{P}_Z \times \mathbf{P}_Z \to \mathbf{P}_Z$

– F4: eq_F: $\mathbf{P}_Z \times \mathbf{P}_Z \to \mathbf{P}_T$

– F5: leq_F: $\mathbf{P}_Z \times \mathbf{P}_Z \to \mathbf{P}_T$

– F6: neg_F: $\mathbf{P}_T \to \mathbf{P}_T$

– F7: and_F: $\mathbf{P}_T \times \mathbf{P}_T \rightarrow \mathbf{P}_T$

describing how the analysis proceeds for the operators of arithmetic and boolean operators. Furthermore, we need a way of turning numbers and truth values into properties:

– F8: $\mathcal{FZ}$: $\mathbf{Num} \rightarrow \mathbf{PState} \rightarrow \mathbf{P}_Z$

– F9: $\mathcal{FT}$: $\{\mathtt{true}, \mathtt{false}\} \rightarrow \mathbf{PState} \rightarrow \mathbf{P}_T$

Example 8.2

For the detection of signs analysis we have

$$\begin{aligned}
\mathsf{add}_F\ (p_1, p_2) &= p_1\ +_S\ p_2\\
\mathsf{mult}_F\ (p_1, p_2) &= p_1\ \star_S\ p_2\\
\mathsf{sub}_F\ (p_1, p_2) &= p_1\ -_S\ p_2\\
\mathsf{eq}_F\ (p_1, p_2) &= p_1\ =_S\ p_2\\
\mathsf{leq}_F\ (p_1, p_2) &= p_1\ \leq_S\ p_2\\
\mathsf{neg}_F\ p &= \neg_T\ p\\
\mathsf{and}_F\ (p_1, p_2) &= p_1\ \wedge_T\ p_2
\end{aligned}$$

where $+_S$, $\star_S$, $-_S$, $-_S$, $=_S$, $\leq_S$, $\neg_T$, and $\wedge_T$ are defined in Tables 7.2 and 7.4. Furthermore, we have

$$\begin{aligned}
\mathcal{FZ}[\![n]\!]\ ps &= \mathsf{abs}_Z(\mathcal{N}[\![n]\!])\\
\mathcal{FT}[\![\mathtt{true}]\!]\ ps &= \text{TT}\\
\text{and } \mathcal{FT}[\![\mathtt{false}]\!]\ ps &= \text{FF}
\end{aligned}$$

so the property states are ignored when determining the properties of numbers and truth values. □

The forward analysis of a statement will be defined by a function $\mathcal{FS}$ of functionality:

$$\mathcal{FS}: \mathbf{Stm} \rightarrow \mathbf{PState} \rightarrow \mathbf{PState}$$

The idea is that if ps is a property of the initial state of S, then $\mathcal{FS}[\![S]\!]\ ps$ is a property of the final state obtained by executing S from the initial state. The totality of $\mathcal{FS}[\![S]\!]$ reflects that we shall be able to analyse *all* statements of **While**, including statements that loop in the direct style semantics. The definition of $\mathcal{FS}$ is given by the clauses of Table 8.2, and they are parameterised on the definition of cond_F:

$$\begin{array}{rcl}
\mathcal{FS}[\![x := a]\!]ps & = & ps[x \mapsto \mathcal{FA}[\![a]\!]ps] \\
\mathcal{FS}[\![\texttt{skip}]\!] & = & \mathsf{id} \\
\mathcal{FS}[\![S_1; S_2]\!] & = & \mathcal{FS}[\![S_2]\!] \circ \mathcal{FS}[\![S_1]\!] \\
\mathcal{FS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & = \mathsf{cond}_F(\mathcal{FB}[\![b]\!], \mathcal{FS}[\![S_1]\!], \mathcal{FS}[\![S_2]\!]) \\
\mathcal{FS}[\![\texttt{while } b \texttt{ do } S]\!] & = & \mathsf{FIX}\ H \\
\text{where } H\ h & = & \mathsf{cond}_F(\mathcal{FB}[\![b]\!], h \circ \mathcal{FS}[\![S]\!], \mathsf{id})
\end{array}$$

Table 8.2 Forward analysis of statements

– F10: cond_F: $((\mathbf{PState} \to \mathbf{P}_T) \times (\mathbf{PState} \to \mathbf{PState}) \times (\mathbf{PState} \to \mathbf{PState})) \to (\mathbf{PState} \to \mathbf{PState})$

specifying how to analyse the conditional.

Example 8.3

For the detection of signs analysis, we have

$$\mathsf{cond}_F(f, h_1, h_2)ps = \begin{cases} \text{INIT} & \text{if } f\ ps = \text{NONE} \\ h_1\ ps & \text{if } f\ ps = \text{TT} \\ h_2\ ps & \text{if } f\ ps = \text{FF} \\ (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \text{ANY} \end{cases}$$

In summary, to specify a forward program analysis of **While**, we only have to provide definitions of the lattices of **P1** and **P2** and to define the functions of **F1** – **F10**.

Backward Analysis

In a backward analysis, the computation is performed in the *opposite direction* of the direct style semantics: given properties of the output of the computation, the analysis will predict the properties the input should have. Thus the idea will be to replace the semantics functions

$$\mathcal{A}\colon \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Z}$$

$$\mathcal{B}\colon \mathbf{Bexp} \to \mathbf{State} \to \mathbf{T}$$

$$\mathcal{S}_{ds}\colon \mathbf{Stm} \to \mathbf{State} \hookrightarrow \mathbf{State}$$

$$\begin{array}{rcl}
\mathcal{BA}[\![n]\!]p & = & \mathcal{BZ}[\![n]\!]p \\
\mathcal{BA}[\![x]\!]p & = & \textsc{init}[x \mapsto p] \\
\mathcal{BA}[\![a_1 + a_2]\!]p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{add}_B\, p) \\
\mathcal{BA}[\![a_1 \star a_2]\!]p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{mult}_B\, p) \\
\mathcal{BA}[\![a_1 - a_2]\!]p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{sub}_B\, p) \\
\\
\mathcal{BB}[\![\mathtt{true}]\!]p & = & \mathcal{BT}[\![\mathtt{true}]\!]p \\
\mathcal{BB}[\![\mathtt{false}]\!]p & = & \mathcal{BT}[\![\mathtt{false}]\!]p \\
\mathcal{BB}[\![a_1 = a_2]\!]p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{eq}_B\, p) \\
\mathcal{BB}[\![a_1 \le a_2]\!]p & = & \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{leq}_B\, p) \\
\mathcal{BB}[\![\neg b]\!]p & = & \mathcal{BB}[\![b]\!](\mathsf{neg}_B\, p) \\
\mathcal{BB}[\![b_1 \wedge b_2]\!]p & = & \mathsf{join}(\mathcal{BB}[\![b_1]\!], \mathcal{BB}[\![b_2]\!])(\mathsf{and}_B\, p)
\end{array}$$

Table 8.3 Backward analysis of expressions

with semantic functions that not only compute with properties rather than values but also invert the function arrows:

$$\mathcal{BA}\text{: } \mathbf{Aexp} \to \mathbf{P}_Z \to \mathbf{PState}$$

$$\mathcal{BB}\text{: } \mathbf{Bexp} \to \mathbf{P}_T \to \mathbf{PState}$$

$$\mathcal{BS}\text{: } \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

For an arithmetic expression a, the idea is that given a property p of the result of computing a, $\mathcal{BA}[\![a]\!]p$ will be a property state telling which properties the variables of a should have in order for the result of computing a to have property p. So, for the result of evaluating a variable x to have property p, we simply assume that the variable has that property and we have no assumptions about the other variables. As another example, consider the analysis of the expression $a_1 + a_2$ and assume that the result of evaluating it should be p; we will then determine which properties the variables of $a_1 + a_2$ should have in order for this to be the case. The first step of the analysis will be to determine which properties the result of evaluating the subexpressions a_1 and a_2 should have in order for the result of $a_1 + a_2$ to be p. Let them be p_1 and p_2. We can now analyse the subexpressions: the analysis of a_i from p_i will find out which properties the variables of a_i should have initially in order for the result of a_i to have property p_i. The last step will be to combine the information from the two subexpressions, and this will often be a least upper bound operation. Thus the analysis can be specified as

$$\mathcal{BA}[\![a_1 + a_2]\!]p = \mathsf{join}(\mathcal{BA}[\![a_1]\!], \mathcal{BA}[\![a_2]\!])(\mathsf{add}_B p)$$

where

$$\mathsf{join}(h_1, h_2)(p_1, p_2) = (h_1\ p_2)\ \sqcup_{PS}\ (h_2\ p_2)$$

and $\sqcup_{PS}$ is the least upper bound on property states.

Similar remarks hold for the backward analysis of booleans expressions. The clauses are given in Table 8.3, and as was the case for the forward analysis, they are parameterised on the auxiliary functions specifying how the arithmetic and boolean operators should be analysed:

– B1: add_B: $\mathbf{P}_Z \to \mathbf{P}_Z \times \mathbf{P}_Z$

– B2: mult_B: $\mathbf{P}_Z \to \mathbf{P}_Z \times \mathbf{P}_Z$

– B3: sub_B: $\mathbf{P}_Z \to \mathbf{P}_Z \times \mathbf{P}_Z$

– B4: eq_B: $\mathbf{P}_T \to \mathbf{P}_Z \times \mathbf{P}_Z$

– B5: leq_B: $\mathbf{P}_T \to \mathbf{P}_Z \times \mathbf{P}_Z$

– B6: neg_B: $\mathbf{P}_T \to \mathbf{P}_T$

– B7: and_B: $\mathbf{P}_T \to \mathbf{P}_T \times \mathbf{P}_T$

We need a way of turning numbers and truth values into property states. Given a number n and a property p, the analysis $\mathcal{BA}$ has to determine a property state that will ensure that the result of evaluating n will have property p. However, it might be the case that n cannot have the property p at all and in this case the property state returned by $\mathcal{BA}[\![n]\!]p$ should differ from that returned if n can have the property p. Therefore we shall assume that we have functions

– B8: $\mathcal{BZ}$: $\mathbf{Num} \to \mathbf{P}_Z \to \mathbf{PState}$

– B9: $\mathcal{BT}$: $\{\mathtt{true}, \mathtt{false}\} \to \mathbf{P}_T \to \mathbf{PState}$

that take the actual property into account when turning numbers and truth values into property states.

Turning to statements, we shall specify their analysis by a function $\mathcal{BS}$ of functionality:

$$\mathcal{BS}\text{: } \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

Again the totality of $\mathcal{BS}[\![S]\!]$ reflects that we shall be able to analyse *all* statements, including those that loop in the direct style semantics. Since $\mathcal{BS}$ is a backward analysis, the argument *ps* of $\mathcal{BS}[\![S]\!]$ will be a property state corresponding to the result of executing S, and $\mathcal{BS}[\![S]\!]$ *ps* will be the property state corresponding to the initial state from which S is executed. The definition of $\mathcal{BS}$ is given in Table 8.4. We shall assume that we have functions

$$
\begin{array}{rcl}
\mathcal{BS}[\![x := a]\!]ps & = & \mathsf{update}_B(ps, x, \mathcal{BA}[\![a]\!]) \\
\mathcal{BS}[\![\texttt{skip}]\!] & = & \mathsf{id} \\
\mathcal{BS}[\![S_1; S_2]\!] & = & \mathcal{BS}[\![S_1]\!] \circ \mathcal{BS}[\![S_2]\!] \\
\mathcal{BS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathsf{cond}_B(\mathcal{BB}[\![b]\!], \mathcal{BS}[\![S_1]\!], \mathcal{BS}[\![S_2]\!]) \\
\mathcal{BS}[\![\texttt{while } b \texttt{ do } S]\!] & = & \mathsf{FIX}\ H \\
\text{where } Hh & = & \mathsf{cond}_B(\mathcal{BB}[\![b]\!], \mathcal{BS}[\![S]\!] \circ h, \mathsf{id})
\end{array}
$$

Table 8.4 Backward analysis of statements

- B10: update_B: $\mathbf{PState} \times \mathbf{Var} \times (\mathbf{P}_Z \to \mathbf{PState}) \to \mathbf{PState}$
- B11: cond_B: $((\mathbf{P}_T \to \mathbf{PState}) \times (\mathbf{PState} \to \mathbf{PState}) \times (\mathbf{PState} \to \mathbf{PState})) \to (\mathbf{PState} \to \mathbf{PState})$

specifying how to analyse the assignment and the conditional.

Exercise 8.4 (**)

Extend **While** with division and develop a specification of an error detection analysis. How would you prove it correct? □

8.2 Security Analysis

We now present a *forward second-order analysis.* As already mentioned, this takes the form of a *security analysis* using properties LOW and HIGH. The intention is that LOW describes public data that may be visible to everybody, whereas HIGH describes private data that may be visible only to the owner of the program. For a computation

$$\mathcal{S}_{ds}[\![S]\!]\ s_1 = s_2$$

we will classify the initial values (in s_1) of program variables as public (LOW) or private (HIGH), and the security analysis should compute a similar classification of the final values (in s_2) of program variables. Intuitively, correctness of the analysis boils down to ensuring that:

- whenever private data (HIGH) are used for computing the new value to be assigned to a variable, then the variable must be classified as private (HIGH), and

– whenever a boolean expression in an `if`-construct or a `while`-loop involves private data (HIGH), then all variables assigned in the body must be classified as private (HIGH).

The first condition takes care of *direct flows* and is fairly obvious; the second condition takes care of *indirect flows* and is sometimes forgotten. Failure to deal correctly with the observations one may make upon private data gives rise to a "covert channel", and the resulting program should be considered insecure.

Example 8.5

Consider a user of a Microsoft Windows operating system. Occasionally programs crash and error reports are produced to be sent back to some external company. Typically a user is asked whether or not he or she will accept that the report be sent; to convince the user to accept, there often is a statement saying that no private data have been included in the public error report. For the user to trust this information, the program constructing the error report must be validated using a security analysis like the one developed here. □

Example 8.6

As a very simple example, in the factorial program

```
y := 1; while ¬(x = 1) do (y := y ⋆ x; x := x − 1)
```

we shall classify the initial value of `y` as private (HIGH) and the initial value of `x` as public (LOW). We may then ask whether the final value of `y` may be classified as public (LOW). This is clearly the case for the program above but is not the case for all programs. An example is

```
while ¬(x = 1) do (y := y ⋆ x; x := x − 1)
```

where we still assume that the initial value of `y` is private (HIGH) and the initial value of `x` is public (LOW). Since the initial value of `y` is not overwritten as before, the final value of `y` also will depend on the initial value of `y`. Hence, the final value of `y` must be classified as private (HIGH). □

The *specification* of the analysis falls into two parts. First we introduce the properties on which the analysis operates, and next we specify the actual analysis for the three syntactic categories.

Properties and Property States

For the security analysis, we shall be interested in two properties of the relationship between values (numbers or truth values):

– LOW, meaning that the values *may* be classified as *public*, and

– HIGH, meaning that the values *must* be classified as *private*.

We shall write

$$\mathbf{P} = \{\textsc{low}, \textsc{high}\}$$

for this set of properties, and we use p as a meta-variable ranging over $\mathbf{P}$. It is less restrictive to know that an expression has the property LOW than HIGH. As a record of this, we define a partial order $\sqsubseteq_P$ on $\mathbf{P}$:

$$\textsc{low} \sqsubseteq_P \textsc{high}, \quad \textsc{low} \sqsubseteq_P \textsc{low}, \quad \textsc{high} \sqsubseteq_P \textsc{high}$$

which may be depicted as

$$\begin{tikzcd}\bullet\ \textsc{high} \arrow[d, dash] \\ \bullet\ \textsc{low}\end{tikzcd}$$

Thus the less restrictive property is at the bottom of the ordering.

Fact 8.7

$(\mathbf{P}, \sqsubseteq_P)$ is a complete lattice. If Y is a subset of $\mathbf{P}$, then

$$\bigsqcup_P Y = \textsc{high} \quad \text{if and only if} \quad \textsc{high} \in Y$$

Following the approach of the previous analysis, we introduce a *property state* mapping variables to properties. As we shall see, this takes care of *direct flows*, and in order to take care of *indirect flows* also we introduce a special token history that captures the "flow of control"; it acts like an "extended program counter". The set **PState** of property states over which the meta-variable *ps* ranges, is then defined by

$$\mathbf{PState} = (\mathbf{Var} \cup \{\mathsf{history}\}) \to \mathbf{P}$$

The idea is that if history is mapped to HIGH, then we may be in the body of a private (HIGH) boolean expression; if it is mapped to LOW, this is guaranteed not to be the case. For a property state $ps \in \mathbf{PState}$, we define the set

$$\mathrm{LOW}(ps) = \{\, x \in \mathbf{Var} \cup \{\mathsf{history}\} \mid ps\ x = \textsc{low}\,\}$$

of "variables" mapped to LOW and we say that

$$ps \text{ is } \textit{proper} \quad \text{if and only if} \quad ps(\mathsf{history}) = \text{LOW}.$$

If *ps* is not proper, we shall sometimes say that it is *improper.*

Our next task will be to endow **PState** with some partially ordered structure. In Lemma 7.1, we instantiate S to be $\mathbf{Var} \cup \{\mathsf{history}\}$ and D to be $\mathbf{P}$ and we get the following corollary.

Corollary 8.8

Let $\sqsubseteq_{PS}$ be the ordering on **PState** defined by

$$ps_1 \sqsubseteq_{PS} ps_2$$

if and only if

$$ps_1\ x \sqsubseteq_P\ ps_2\ x \text{ for all } x \in\ \mathbf{Var} \cup \{\mathsf{history}\}$$

Then $(\mathbf{PState}, \sqsubseteq_{PS})$ is a complete lattice. In particular, the least upper bound $\bigsqcup_{PS} Y$ of a subset Y of **PState** is characterized by

$$(\bigsqcup_{PS} Y)\ x = \bigsqcup_{PS} \{\, ps\ x \mid ps \in Y\, \}$$

We shall write LOST for the property state *ps* that maps all variables to HIGH and that maps history to HIGH. Similarly, we shall write INIT for the property state that maps all variables to LOW and that maps history to LOW. Note that INIT is the *least element* of **PState**.

Example 8.9

To motivate the need to keep track of the "flow of control" (that is, the need for history), consider the statement S given by

```
if x = 1 then x := 1 else x := 2
```

Assume first that we do *not* keep track of the flow of control. The analysis of each of the two branches will give rise to a function that maps *ps* to $ps[\mathtt{x}\mapsto\text{LOW}]$, so it would be natural to expect the analysis of S to do the same. However, this will *not always be correct.* To see this, suppose that *ps* `x` = HIGH, indicating that the initial value of `x` is private. Then surely the resulting value of `x` will convey some information about private data and hence cannot be characterised as public (LOW).

The token history is used to solve this dilemma: if the analysis of the test gives HIGH, then history will be mapped to HIGH in the resulting property state; otherwise it is mapped to LOW. We shall return to this example in more detail when we have specified the analysis for statements. □

Exercise 8.10 (Essential)

Show that

$$ps_1 \sqsubseteq_{PS} ps_2 \quad \text{if and only if} \quad \text{LOW}(ps_1) \supseteq \text{LOW}(ps_2)$$

Next show that

$$\text{LOW}(\bigsqcup_{PS} Y) = \bigcap \{\, \text{LOW}(ps) \mid ps \in Y \,\}$$

whenever Y is a non-empty subset of **PState**. □

Analysis of Expressions

The analysis of an arithmetic expression a will be specified by a (total) function $\mathcal{SA}[\![a]\!]$ from property states to properties:

$$\mathcal{SA}\colon \textbf{Aexp} \to \textbf{PState} \to \textbf{P}$$

Similarly, the analysis of a boolean expression b will be defined by a (total) function $\mathcal{SB}[\![b]\!]$ from property states to properties:

$$\mathcal{SB}\colon \textbf{Bexp} \to \textbf{PState} \to \textbf{P}$$

The defining clauses are given in Table 8.5.

The overall idea is that once ps history has the value HIGH, then all results produced should be HIGH. This is reflected directly in the clauses for the basic constructs n, x, `true`, and `false`. For the composite expression, such as for example $a_1 + a_2$, the idea is that it can only have the property LOW if both subexpressions have that property. This is ensured by the binary operation $\sqcup_P$.

The functions $\mathcal{SA}[\![a]\!]$ and $\mathcal{SB}[\![b]\!]$ are closely connected with the sets of free variables defined in Chapter 1.

Exercise 8.11 (Essential)

Prove that, for every arithmetic expression a, we have

$$\mathcal{SA}[\![a]\!]ps = \text{LOW} \quad \text{if and only if} \quad \text{FV}(a) \cup \{\mathsf{history}\} \subseteq \text{LOW}(ps)$$

Formulate and prove a similar result for boolean expressions. Deduce that for all a of **Aexp** we get $\mathcal{SA}[\![a]\!]ps = \text{HIGH}$ if ps is improper, and that for all b of **Bexp** we get $\mathcal{SB}[\![b]\!]ps = \text{HIGH}$ if ps is improper. □

$$
\begin{array}{rcl}
\mathcal{SA}[\![n]\!]ps & = & \begin{cases} \text{LOW} & \text{if ps } \mathsf{history} = \text{LOW} \\ \text{HIGH} & \text{otherwise} \end{cases} \\
\mathcal{SA}[\![x]\!]ps & = & \begin{cases} \text{ps x} & \text{if ps } \mathsf{history} = \text{LOW} \\ \text{HIGH} & \text{otherwise} \end{cases} \\
\mathcal{SA}[\![a_1 + a_2]\!]ps & = & (\mathcal{SA}[\![a_1]\!]ps) \sqcup_P (\mathcal{SA}[\![a_2]\!]ps) \\
\mathcal{SA}[\![a_1 \star a_2]\!]ps & = & (\mathcal{SA}[\![a_1]\!]ps) \sqcup_P (\mathcal{SA}[\![a_2]\!]ps) \\
\mathcal{SA}[\![a_1 - a_2]\!]ps & = & (\mathcal{SA}[\![a_1]\!]ps) \sqcup_P (\mathcal{SA}[\![a_2]\!]ps) \\
\mathcal{SB}[\![\texttt{true}]\!]ps & = & \begin{cases} \text{LOW} & \text{if ps } \mathsf{history} = \text{LOW} \\ \text{HIGH} & \text{otherwise} \end{cases} \\
\mathcal{SB}[\![\texttt{false}]\!]ps & = & \begin{cases} \text{LOW} & \text{if ps } \mathsf{history} = \text{LOW} \\ \text{HIGH} & \text{otherwise} \end{cases} \\
\mathcal{SB}[\![a_1 = a_2]\!]ps & = & (\mathcal{SA}[\![a_1]\!]ps) \sqcup_P (\mathcal{SA}[\![a_2]\!]ps) \\
\mathcal{SB}[\![a_1 \le a_2]\!]ps & = & (\mathcal{SA}[\![a_1]\!]ps) \sqcup_P (\mathcal{SA}[\![a_2]\!]ps) \\
\mathcal{SB}[\![\neg b]\!]ps & = & \mathcal{SB}[\![b]\!]ps \\
\mathcal{SB}[\![b_1 \wedge b_2]\!]ps & = & (\mathcal{SB}[\![b_1]\!]ps) \sqcup_P (\mathcal{SB}[\![b_2]\!]ps)
\end{array}
$$

Table 8.5 Security analysis of expressions

Analysis of Statements

Turning to statements, we shall specify their analysis by a function $\mathcal{SS}$ of functionality:

$$\mathcal{SS}\text{: } \mathbf{Stm} \to \mathbf{PState} \to \mathbf{PState}$$

The totality of $\mathcal{SS}[\![S]\!]$ reflects that we shall be able to analyse *all* statements, including a statement such as `while true do skip` that loops. The definition of $\mathcal{SS}$ is given in Table 8.6.

The clauses for assignment, `skip`, and composition are much as in the direct style denotational semantics of Chapter 5. In the clause for `if` b `then` S_1 `else` S_2 we use the auxiliary function cond_S defined by

$$\mathsf{cond}_S(f,\, h_1,\, h_2)\ ps = \begin{cases} (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \text{LOW} \\ \text{LOST} & \text{if } f\ ps = \text{HIGH} \end{cases}$$

$$\begin{array}{rcl}
\mathcal{SS}\llbracket x := a\rrbracket ps & = & ps[x \mapsto \mathcal{SA}\llbracket a\rrbracket ps] \\
\mathcal{SS}\llbracket \texttt{skip}\rrbracket & = & \mathsf{id} \\
\mathcal{SS}\llbracket S_1 ; S_2\rrbracket & = & \mathcal{SS}\llbracket S_2\rrbracket \circ \mathcal{SS}\llbracket S_1\rrbracket \\
\mathcal{SS}\llbracket \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_s\rrbracket & = & \mathsf{cond}_S(\mathcal{SB}\llbracket b\rrbracket, \mathcal{SS}\llbracket S_1\rrbracket, \mathcal{SS}\llbracket S_2\rrbracket) \\
\mathcal{SS}\llbracket \texttt{while } b \texttt{ do } S\rrbracket & = & \mathsf{FIX}\ H \\
\text{where } H\ h & = & \mathsf{cond}_S(\mathcal{SB}\llbracket b\rrbracket, h \circ \mathcal{SS}\llbracket S\rrbracket, \mathsf{id})
\end{array}$$

Table 8.6 Security analysis of statements

First consider the case where the condition depends on public data only; that is, where $f\ ps = \text{LOW}$. For each variable x, we can determine the result of analysing each of the branches, namely $(h_1\ ps)\ x$ for the true branch and $(h_2\ ps)\ x$ for the false branch. The least upper bound of these two results will be the new property bound to x; that is, the new property state will map x to

$$((h_1\ ps)\ x) \sqcup_P ((h_2\ ps)\ x)$$

If the condition depends on private data (that is, $f\ ps = \text{HIGH}$), then the analysis of the conditional will have to produce an improper property state and we shall therefore use the property state LOST.

Example 8.12

Returning to Example 8.9 we see that if $ps\ \mathtt{x} = \text{HIGH}$ then

$$\mathcal{SB}\llbracket \mathsf{x} = 1\rrbracket ps = \text{HIGH}$$

We therefore get

$$\mathcal{SS}\llbracket \texttt{if x} = 1 \texttt{ then x} := 1 \texttt{ else x} := 2\rrbracket ps = \text{LOST}$$

using the definition of cond_S above. □

Example 8.13

Consider now the statement

```
if x = x then x := 1 else x := y
```

First assume that $ps\ \mathtt{x} = \text{LOW}$, $ps\ \mathtt{y} = \text{HIGH}$, and $ps\ \mathsf{history} = \text{LOW}$. Then $\mathcal{SA}\llbracket \mathsf{x} = \mathsf{x}\rrbracket ps = \text{LOW}$ and we get

$$\begin{aligned}
&\mathcal{SS}[\![\texttt{if x = x then x := 1 else x := y}]\!]ps\ \texttt{x}\\
&\qquad = \mathsf{cond}_S(\mathcal{SB}[\![\texttt{x = x}]\!], \mathcal{SS}[\![\texttt{x := 1}]\!], \mathcal{SS}[\![\texttt{x := y}]\!])ps\ \texttt{x}\\
&\qquad = (\mathcal{SS}[\![\texttt{x := 1}]\!]ps \sqcup_{PS}\ \mathcal{SS}[\![\texttt{x := y}]\!]ps)\ \texttt{x}\\
&\qquad = \textsc{high}
\end{aligned}$$

because $\mathcal{SB}[\![\texttt{x = x}]\!]ps = \textsc{low}$, $(\mathcal{SS}[\![\texttt{x := 1}]\!]ps)\ \texttt{x} = \textsc{low}$ but $(\mathcal{SS}[\![\texttt{x := y}]\!]ps)\ \texttt{x} = \textsc{high}$. So even though the false branch never will be executed, it will influence the result obtained by the analysis.

Next assume that $ps\ \texttt{x} = \textsc{high}$, $ps\ \texttt{y} = \textsc{low}$, and $ps\ \mathsf{history} = \textsc{low}$. Then $\mathcal{SA}[\![\texttt{x = x}]\!]ps = \textsc{high}$ and we get

$$\begin{aligned}
&\mathcal{SS}[\![\texttt{if x = x then x := 1 else x := y}]\!]ps\\
&\qquad = \mathsf{cond}_S(\mathcal{SB}[\![\texttt{x = x}]\!], \mathcal{SS}[\![\texttt{x := 1}]\!], \mathcal{SS}[\![\texttt{x := y}]\!])ps\\
&\qquad = \textsc{lost}
\end{aligned}$$

because $\mathcal{SB}[\![\texttt{x = x}]\!]ps = \textsc{high}$. So even though the test always evaluates to true for all states, this is not captured by the analysis. More complex analyses could do better (for example by trying to predict the outcome of tests). ☐

In the clause for the `while`-loop, we also use the function cond_S and otherwise the clause is as in the direct style denotational semantics of Chapter 5. In particular, we use the fixed point operation FIX as it corresponds to unfolding the `while`-loop any number of times. As in Chapter 5, the fixed point is defined by

$$\mathsf{FIX}\ H = \bigsqcup \{\ H^n\ \bot \mid n \geq 0\ \}$$

where the functionality of H is

$$H\colon (\mathbf{PState} \to \mathbf{PState}) \to (\mathbf{PState} \to \mathbf{PState})$$

and where $\mathbf{PState} \to \mathbf{PState}$ is the set of total functions from **PState** to **PState**. In order for this to make sense H must be a continuous function on a ccpo with least element $\bot$.

We may summarise how to use the security analysis as follows:

INPUT: a statement S of **While**

a set $L_I \subseteq \mathbf{Var}$ of (input) variables whose initial values are regarded as public

a set $L_O \subseteq \mathbf{Var}$ of (output) variables whose final values should be regarded as public

OUTPUT: YES!, if we can prove this to be safe

NO?, if we cannot prove this to be safe

METHOD: let ps_I be determined by $\text{LOW}(ps_I) = L_I \cup \{\textsf{history}\}$

let $ps_O = \mathcal{SS}[\![S]\!]\ ps_I$

output YES! if $\text{LOW}(ps_O) \supseteq L_O \cup \{\textsf{history}\}$

output NO? otherwise

Example 8.14

We are now in a position where we can attempt the application of the analysis to the factorial statement:

$$\mathcal{SS}[\![\texttt{y := 1; while}\ \neg(\texttt{x = 1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}]\!]$$

We shall apply this function to the *proper* property state ps_0 that maps `x` to LOW and all other variables (including `y`) to HIGH, as this corresponds to viewing `x` as a public input variable of the statement.

To do so, we use the clauses of Tables 8.5 and 8.6 and get

$$\begin{aligned}&\mathcal{SS}[\![\texttt{y := 1; while}\ \neg(\texttt{x = 1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}]\!]ps_0\\ &\qquad = (\textsf{FIX}\ H)\ (ps_0[\texttt{y}\mapsto\text{LOW}])\end{aligned}$$

where

$$H\ h = \textsf{cond}_S(\mathcal{SB}[\![\neg(\texttt{x = 1})]\!], h\circ\mathcal{SS}[\![\texttt{y := y}\star\texttt{x; x := x}-\texttt{1}]\!], id)$$

We first simplify H and obtain

$$(H\ h)\ ps = \begin{cases}\text{LOST} & \text{if } ps\ \textsf{history} = \text{HIGH or } ps\ \texttt{x} = \text{HIGH}\\ (h\ ps)\sqcup_{PS} ps & \text{if } ps\ \textsf{history} = \text{LOW and } ps\ \texttt{x} = \text{LOW}\end{cases}$$

At this point, we shall exploit the result of Exercise 7.8:

$$\begin{aligned}&\text{if } H^{\text{n}}\ \bot = H^{\text{n}+1}\ \bot \text{ for some n}\\ &\text{then } \textsf{FIX}\ H = H^{\text{n}}\ \bot\end{aligned}$$

where $\bot$ is the function $\bot\ ps = \text{INIT}$ for all ps. We can now calculate the iterands $H^0\ \bot$, $H^1\ \bot$, $H^2\ \bot$, $\cdots$. We obtain

$$(H^0\ \bot)\ ps = \text{INIT}$$

$$(H^1\ \bot)\ ps = \begin{cases}\text{LOST} & \text{if } ps\ \texttt{x} = \text{HIGH or } ps \text{ not proper}\\ ps & \text{if } ps\ \texttt{x} = \text{LOW and } ps \text{ proper}\end{cases}$$

$$(H^2\ \bot)\ ps = \begin{cases}\text{LOST} & \text{if } ps\ \texttt{x} = \text{HIGH or } ps \text{ not proper}\\ ps & \text{if } ps\ \texttt{x} = \text{LOW and } ps \text{ proper}\end{cases}$$

where ps is an arbitrary property state. Since $H^1 \perp = H^2 \perp$, our assumption above ensures that we have found the least fixed point for H:

$$(\mathsf{FIX}\ H)\ ps = \begin{cases} \textsc{lost} & \text{if } ps\ \mathtt{x} = \textsc{high} \text{ or } ps \text{ not proper} \\ ps & \text{if } ps\ \mathtt{x} = \textsc{low} \text{ and } ps \text{ proper} \end{cases}$$

It is now straightforward to verify that $(\mathsf{FIX}\ H)\ (ps_0[\mathsf{y}\mapsto\textsc{low}])\ \mathtt{y} = \textsc{low}$ and that $(\mathsf{FIX}\ H)(ps_0[\mathsf{y}\mapsto\textsc{low}])$ is proper. We conclude that the statement implements a service `y:=` $\cdots$ `x` $\cdots$ (actually `y:= x!`) that maps public data to public data and hence can be considered secure. □

Exercise 8.15

Extend **While** with the statement `repeat` S `until` b and give the new (compositional) clause for $\mathcal{SS}$. Motivate your extension. □

Exercise 8.16

Prove that the security analysis as specified by $\mathcal{SA}$, $\mathcal{SB}$, and $\mathcal{SS}$ does indeed exist. □

Exercise 8.17 (Essential)

Show that for every statement S

$$ps\ \mathsf{history} \sqsubseteq (\mathcal{SS}[\![S]\!]ps)\ \mathsf{history}$$

so that ps must be proper if $\mathcal{SS}[\![S]\!]ps$ is. In the case of `while` b `do` S, you should first prove that for all $\mathrm{n} \geq 1$

$$ps\ \mathsf{history} \sqsubseteq ((H^{\mathrm{n}}\ \perp)\ ps)\ \mathsf{history}$$

where $\perp ps' = \textsc{init}$ for all ps' and $H\ h = \mathsf{cond}_S(\mathcal{SB}[\![b]\!],\ h \circ \mathcal{SS}[\![S]\!],\ \mathrm{id})$. □

Exercise 8.18 (**)

The results developed in Appendix B for computing fixed points are applicable to a minor variation of the security analysis:

– show that the analysis is in the completely additive framework (that is, that the functions $\mathcal{SS}[\![S]\!]$ are strict and additive for all statements S of **While**), and

– show that the functionals H obtained for `while`-loops can be written in iterative form (provided that a minor modification is made in the definition of cond_S).

Conclude, using Theorem B.14, that in a program with p variables at most p+1 iterations are needed to compute the fixed point. □

Exercise 8.19 (*)

Formulate a *dependency analysis* as follows. Given a statement, you should designate a subset of the variables as *input* variables and another subset as *output* variables. Then the statement is intended to compute an *output* (being the final values of the output variables) only depending on the *input* (being the initial values of the input variables). Introduce properties OK for indicating that variables only depend on the input and D? for indicating that values may depend on more than the input and develop the corresponding dependency analysis. (Hint: Consider how OK and D? should be compared with LOW and HIGH. Will there be any major differences between the analyses?) □

8.3 Safety of the Analysis

In this section, we prove the safety of the security analysis with respect to the denotational semantics. Since it is a second-order analysis, the properties of interest do not directly describe values. Instead we employ a simulation approach where we view properties as describing relationships between values. The main idea will be that LOW demands the values to be equal, whereas HIGH places no demands on the values.

Second-Order Properties

As a first step, we shall define parameterized relations for arithmetic and boolean values:

$$\mathsf{rel}_A\colon \mathbf{P} \to (\mathbf{Z} \times \mathbf{Z} \to \mathbf{T})$$
$$\mathsf{rel}_B\colon \mathbf{P} \to (\mathbf{T} \times \mathbf{T} \to \mathbf{T})$$

For arithmetic values, the relation is defined by

$$\mathsf{rel}_A(p)(v_1,\, v_2) = \begin{cases} \mathbf{tt} & p = \text{HIGH or } v_1 = v_2 \\ \mathbf{ff} & \text{otherwise} \end{cases}$$

and similarly for boolean values:

$$\mathsf{rel}_B(p)(v_1,\, v_2) = \begin{cases} \mathbf{tt} & p = \textsc{high} \text{ or } v_1 = v_2 \\ \mathbf{ff} & \text{otherwise} \end{cases}$$

We shall often omit the subscript when no confusion is likely to result. Each of the relations takes a property and two values as parameters. Intuitively, the property expresses how much the two values are allowed to differ. Thus HIGH puts no requirements on the values, whereas LOW requires that the two values be equal. As an aid to readability, we shall often write

$$v_1 \equiv v_2 \ \mathsf{rel}\ p$$

instead of $\mathsf{rel}(p)(v_1,\, v_2)$ and we shall say that v_1 *and* v_2 *are equal as far as* p *is concerned* (or relative to p).

The next step is to introduce a parameterized relation for states:

$$\mathsf{rel}_S\text{: } \mathbf{PState} \to (\mathbf{State} \times \mathbf{State} \to \mathbf{T})$$

It is defined by

$$\mathsf{rel}_S(ps)(s_1,\, s_2) = \begin{cases} \mathbf{tt} & \text{if } ps\ \mathsf{history} = \textsc{high} \\ & \text{or } \forall x \in \mathbf{Var} \cap \text{LOW}(ps) : s_1\ x = s_2\ x \\ \mathbf{ff} & \text{otherwise} \end{cases}$$

and again we may omit the subscript when no confusion is likely to occur. The relation expresses the extent to which two states are allowed to differ as far as a given property state is concerned. If ps is not proper, then $\mathsf{rel}(ps)$ will hold on any two states. However, if ps is proper then, $\mathsf{rel}(ps)$ will hold on two states if they are equal on the variables in $\text{LOW}(ps)$. Phrased differently, we may view ps as a pair of glasses that only allows us to see part of the states, and $\mathsf{rel}(ps)(s_1,\, s_2)$ means that s_1 and s_2 look the same when viewed through that pair of glasses. Again we shall write

$$s_1 \equiv s_2 \ \mathsf{rel}\ ps$$

for $\mathsf{rel}(ps)(s_1,\, s_2)$.

Example 8.20

Let s_1, s_2, and ps be given by

$s_1\ \mathtt{x} = \mathbf{1}$ and $s_1\ y = \mathbf{0}$ for $y \in \mathbf{Var}\backslash\{\mathtt{x}\}$

$s_2\ \mathtt{x} = \mathbf{2}$ and $s_2\ y = \mathbf{0}$ for $y \in \mathbf{Var}\backslash\{\mathtt{x}\}$

$ps\ \mathtt{x} = \textsc{high}$ and $ps\ y = \textsc{low}$ for $y \in (\mathbf{Var} \cup \{\mathsf{history}\})\backslash\{\mathtt{x}\}$

Then $s_1 \equiv s_2\ \mathsf{rel}\ ps$. □

Example 8.21

It is instructive to reconsider the need for keeping track of the "flow of control" as discussed in Examples 8.9 and 8.12. Consider the following statements:

$$S_1 \equiv \texttt{x} := \texttt{1}$$

$$S_2 \equiv \texttt{x} := \texttt{2}$$

It would be natural to expect that the analysis of S_1 will map any property state *ps* to the property state $ps[\texttt{x}\mapsto\text{LOW}]$ since a constant value cannot convey any information about private data. A similar argument holds for S_2. Now consider the statements

$$S_{11} \equiv \texttt{if x = 1 then } S_1 \texttt{ else } S_1$$

$$S_{12} \equiv \texttt{if x = 1 then } S_1 \texttt{ else } S_2$$

Again we may expect that the analysis of S_{11} will map any property state *ps* to the property state $ps[\texttt{x}\mapsto\text{LOW}]$ since S_{11} is semantically equivalent to S_1.

Concerning S_{12}, it will not always be correct for the analysis to map a property state *ps* to $ps[\texttt{x}\mapsto\text{LOW}]$. For an example, suppose that *ps*, s_1, and s_2 are such that

$$ps\ \texttt{x} = \text{HIGH and } ps\ y = \text{LOW for } y \in (\mathbf{Var} \cup \{\mathsf{history}\}) \backslash \{\texttt{x}\}$$

$$s_1\ \texttt{x} = \mathbf{1} \text{ and } s_1\ y = \mathbf{0} \text{ for } y \in \mathbf{Var} \backslash \{\texttt{x}\}$$

$$s_2\ \texttt{x} = \mathbf{2} \text{ and } s_2\ y = \mathbf{0} \text{ for } y \in \mathbf{Var} \backslash \{\texttt{x}\}$$

Then Example 8.20 gives

$$s_1 \equiv s_2 \ \mathsf{rel}\ ps$$

but $\mathcal{S}_{\text{ds}}[\![S_{12}]\!]s_1 \equiv \mathcal{S}_{\text{ds}}[\![S_{12}]\!]s_2 \ \mathsf{rel}\ ps[\texttt{x}\mapsto\text{LOW}]$ *fails* because $\mathcal{S}_{\text{ds}}[\![S_{12}]\!]s_1 = s_1$ and $\mathcal{S}_{\text{ds}}[\![S_{12}]\!]s_2 = s_2$ and $s_1\ \texttt{x} \neq s_2\ \texttt{x}$.

However, from the point of view of the *analysis*, there is no difference between S_1 and S_2 because neither value `1` nor `2` conveys information about private data. Since the analysis is compositionally defined, this means that there can be no difference between S_{11} and S_{12} from the point of view of the analysis. Therefore we have to accept that the analysis of S_{11} also should *not* allow mapping of an arbitrary property state *ps* to $ps[\texttt{x}\mapsto\text{LOW}]$.

The difference between S_1 and S_2 arises when the "flow of control" depends on private data, and it is for this that we need the special token history. We shall transform a property state into an improper one, by mapping history to HIGH, whenever the "flow of control" depends on private data. Thus, if $ps\ \texttt{x} = \text{HIGH}$, then it is the test, $\texttt{x = 1}$, in S_{11} and S_{12} that will be responsible for mapping *ps* into the improper property state LOST, and then the effect of analysing S_1 and S_2 does not matter (provided, of course, that no improper property state is mapped into a proper property state). $\square$

Properties of rel

To study the properties of the parameterized relation rel, we need a notion of an equivalence relation. A relation

$$R\colon E \times E \to \mathbf{T}$$

is an *equivalence relation* on a set E if and only if

$R(e_1, e_1)$	(reflexivity)
$R(e_1, e_2)$ and $R(e_2, e_3)$ imply $R(e_1, e_3)$	(transitivity)
$R(e_1, e_2)$ implies $R(e_2, e_1)$	(symmetry)

for all e_1, e_2, and e_3 of E.

Exercise 8.22 (Essential)

Show that $\mathsf{rel}_A(p)$, $\mathsf{rel}_B(p)$ and $\mathsf{rel}_S(ps)$ are equivalence relations for all choices of $p \in \mathbf{P}$ and $ps \in \mathbf{PState}$. $\square$

Each of rel_A, rel_B, and rel_S are examples of parameterized (equivalence) relations. In general, a *parameterized relation* is of the form

$$\mathcal{R}\colon D \to (E \times E \to \mathbf{T})$$

where $(D, \sqsubseteq)$ is a partially ordered set, E is a set, and each $\mathcal{R}(d)$ is a relation. We shall say that a parameterized relation $\mathcal{R}$ is a *Kripke relation* if

$$d_1 \sqsubseteq d_2 \text{ implies that for all } e_1, e_2 \in E\colon$$
$$\text{if } \mathcal{R}(d_1)(e_1, e_2) \text{ then } \mathcal{R}(d_2)(e_1, e_2)$$

Note that this is a kind of monotonicity property.

Lemma 8.23

rel_S is a Kripke relation.

Proof: Let ps_1 and ps_2 be such that $ps_1 \sqsubseteq_{\mathrm{PS}} ps_2$, and assume that

$$s_1 \equiv s_2 \ \mathsf{rel}\ ps_1$$

holds for all states s_1 and s_2. We must show

$$s_1 \equiv s_2 \ \mathsf{rel}\ ps_2$$

If ps_2 $\mathsf{history}$ = HIGH, this is immediate from the definition of rel_S. So assume that ps_2 $\mathsf{history}$ = LOW. In this case, we must show

$$\forall x \in \mathrm{LOW}(ps_2) \cap \mathbf{Var}\colon s_1\ x = s_2\ x$$

Since $ps_1 \sqsubseteq_{\mathrm{PS}} ps_2$ and ps_2 $\mathsf{history}$ = LOW, it must be the case that ps_1 $\mathsf{history}$ is LOW. From $s_1 \equiv s_2 \ \mathsf{rel}\ ps_1$, we therefore get

$$\forall x \in \mathrm{LOW}(ps_1) \cap \mathbf{Var}\colon s_1\ x = s_2\ x$$

From Exercise 8.10 and the assumption $ps_1 \sqsubseteq_{\mathrm{PS}} ps_2$, we get $\mathrm{LOW}(ps_1) \supseteq \mathrm{LOW}(ps_2)$ and thereby we get the desired result. □

Exercise 8.24 (Essential)

Show that rel_A and rel_B are Kripke relations. □

Safety of Expressions

Let g: **State** → **Z** be a function, perhaps of the form $\mathcal{A}[\![a]\!]$ for some arithmetic expression $a \in$ **Aexp**, and let h: **PState** → **P** be another function, perhaps of the form $\mathcal{SA}[\![a]\!]$ for some arithmetic expression $a \in$ **Aexp**. We shall introduce a relation

$$g\ \mathsf{safe}_A\ h$$

for expressing when the analysis h is correct with respect to the semantics g. It is defined by

$$s_1 \equiv s_2\ \mathsf{rel}_S\ ps \quad \text{implies} \quad g\ s_1 \equiv g\ s_2\ \mathsf{rel}_A\ h\ ps$$

for all states s_1 and s_2 and property states ps. This condition says that the results of g will be suitably related provided that the arguments are. It is perhaps more intuitive when rephrased as

$$(s_1 \equiv s_2\ \mathsf{rel}_S\ ps) \text{ and } (h\ ps = \textsc{low}) \quad \text{imply} \quad g\ s_1 = g\ s_2$$

The safety of the analysis $\mathcal{SA}$ is then expressed by the following fact.

Fact 8.25

For all arithmetic expressions $a \in$ **Aexp**, we have

$$\mathcal{A}[\![a]\!]\ \mathsf{safe}_A\ \mathcal{SA}[\![a]\!]$$

Proof: This is a consequence of Lemma 1.12 and Exercise 8.11. □

The analysis $\mathcal{SB}$ of boolean expressions is safe in the following sense.

Exercise 8.26 (Essential)

Repeat the development for boolean expressions; that is, define a relation safe_B and show that

$$\mathcal{B}[\![b]\!] \ \mathsf{safe}_B \ \mathcal{SB}[\![b]\!]$$

for all boolean expressions $b \in \mathbf{Bexp}$. □

Safety of Statements

Our key tool will be the relation $s_1 \equiv s_2$ rel ps, and we shall show that if this relationship holds before the statement is executed and analysed, then either the statement will loop on both states or the same relationship will hold between the final states and the final property state (provided that the analysis does not get "lost"). We shall formalize this by defining a relation

$$g \ \mathsf{safe}_S \ h$$

between a function g: **State** $\hookrightarrow$ **State**, perhaps of the form $\mathcal{S}_{\mathrm{ds}}[\![S]\!]$ for some S in **Stm**, and another function h: **PState** $\rightarrow$ **PState**, perhaps of the form $\mathcal{SS}[\![S]\!]$ for some S in **Stm**. The formal definition is

$$(s_1 \equiv s_2 \ \mathsf{rel}\ ps) \text{ and } (h\ ps \text{ is proper}) \quad \text{imply}$$
$$(g\ s_1 = \underline{\text{undef}} \text{ and } g\ s_2 = \underline{\text{undef}}) \quad \text{or}$$
$$(g\ s_1 \neq \underline{\text{undef}} \text{ and } g\ s_2 \neq \underline{\text{undef}} \text{ and } g\ s_1 \equiv g\ s_2 \ \mathsf{rel}\ h\ ps)$$

for all states s_1, $s_2 \in$ **State** and all property states $ps \in$ **PState**. To explain this definition, consider two states s_1 and s_2 that are equal relative to ps. If ps is proper, this means that $s_1\ x = s_2\ x$ for all variables x in LOW(ps). The analysis of the statement may get "lost", in which case $h\ ps$ is not proper and we cannot deduce anything about the behaviour of the statement. Alternatively, it may be the case that $h\ ps$ is proper, and in that case the statement must behave in the same way whether executed from s_1 or from s_2. In particular

– the statement may enter a loop when executed from s_1 and s_2 (that is, $g\ s_1 = \underline{\text{undef}}$ and $g\ s_2 = \underline{\text{undef}}$) or

– the statement does not enter a loop when executed from s_1 and s_2 (that is, $g\ s_1 \neq \underline{\text{undef}}$ and $g\ s_2 \neq \underline{\text{undef}}$).

In the latter case, the two final states $g\ s_1$ and $g\ s_2$ must be equal relative to the resulting property state $h\ ps$; that is, $(g\ s_1)\ x = (g\ s_2)\ x$ for all variables x in LOW($h\ ps$). We may then formulate the desired relationship between the semantics and the analysis as follows.

Theorem 8.27

For all statements S of **While**, we have $\mathcal{S}_{\mathrm{ds}}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S]\!]$.

For the proof we need some properties of the auxiliary operations.

Lemma 8.28

Let g_1, g_2: **State** $\hookrightarrow$ **State** and h_1, h_2: **PState** $\to$ **PState** and assume that

$$ps\ \mathsf{history} \sqsubseteq_{\mathrm{P}} (h_2\ ps)\ \mathsf{history} \qquad (*)$$

holds for all $ps \in$ **PState**. Then

$$g_1\ \mathsf{safe}_S\ h_1 \text{ and } g_2\ \mathsf{safe}_S\ h_2 \quad \text{imply} \quad g_2 \circ g_1\ \mathsf{safe}_S\ h_2 \circ h_1$$

Proof: Let s_1, s_2, and ps be such that

$$s_1 \equiv s_2\ \mathsf{rel}\ ps \text{ and } (h_2 \circ h_1)\ ps \text{ is proper}$$

Using that $h_2\ (h_1\ ps)$ is proper, we get from (*) that $h_1\ ps$ must be proper as well (by taking ps to be $h_1\ ps$). So, from the assumption $g_1\ \mathsf{safe}_S\ h_1$, we get

$$g_1\ s_1 = \underline{\mathrm{undef}} \text{ and } g_1\ s_2 = \underline{\mathrm{undef}}$$

or

$$g_1\ s_1 \neq \underline{\mathrm{undef}} \text{ and } g_1\ s_2 \neq \underline{\mathrm{undef}} \text{ and } g_1\ s_1 \equiv g_1\ s_2\ \mathsf{rel}\ h_1\ ps$$

In the first case, we are finished since it follows that $(g_2 \circ g_1)\ s_1 = \underline{\mathrm{undef}}$ and that $(g_2 \circ g_1)\ s_2 = \underline{\mathrm{undef}}$. In the second case, we use that

$$g_1\ s_1 \equiv g_1\ s_2\ \mathsf{rel}\ h_1\ ps \text{ and } h_2(h_1\ ps) \text{ is proper}$$

The assumption $g_2\ \mathsf{safe}_S\ h_2$ then gives

$$g_2\ (g_1\ s_1) = \underline{\mathrm{undef}} \text{ and } g_2\ (g_1\ s_2) = \underline{\mathrm{undef}}$$

or

$$g_2\ (g_1\ s_1) \neq \underline{\mathrm{undef}} \text{ and } g_2\ (g_1\ s_2) \neq \underline{\mathrm{undef}} \text{ and}$$

$$g_2(g_1\ s_1) \equiv g_2(g_1\ s_2)\ \mathsf{rel}\ h_2(h_1\ ps)$$

In both cases we have completed the proof. □

Lemma 8.29

Assume that g_1, g_2: **State** $\hookrightarrow$ **State** and g: **State** $\to$ **T**, and that h_1, h_2: **PState** $\to$ **PState** and f: **PState** $\to$ **P**. Then

$$g \ \mathsf{safe}_B\ f,\ g_1\ \mathsf{safe}_S\ h_1 \text{ and } g_2\ \mathsf{safe}_S\ h_2 \quad \text{imply}$$

$$\mathsf{cond}(g,\ g_1,\ g_2)\ \mathsf{safe}_S\ \mathsf{cond}_S(f,\ h_1,\ h_2)$$

Proof: Let s_1, s_2, and ps be such that

$$s_1 \equiv s_2\ \mathsf{rel}\ ps \text{ and } \mathsf{cond}_S(f,\ h_1,\ h_2)\ ps \text{ is proper}$$

First assume that $f\ ps = \text{HIGH}$. This case turns out to be impossible since then $\mathsf{cond}_S(f,\ h_1,\ h_2)\ ps = \text{LOST}$ so $\mathsf{cond}_S(f,\ h_1,\ h_2)\ ps$ cannot be proper.

So we know that $f\ ps = \text{LOW}$. From $g\ \mathsf{safe}_B\ f$, we then get $g\ s_1 = g\ s_2$. We also get that $\mathsf{cond}_S(f,\ h_1,\ h_2)\ ps = (h_1\ ps) \sqcup_{\text{PS}} (h_2\ ps)$. Thus $h_1\ ps$ as well as $h_2\ ps$ must be proper since otherwise $\mathsf{cond}_S(f,\ h_1,\ h_2)\ ps$ cannot be proper. Now let 'i' denote the branch chosen by the test g. We then have

$$s_1 \equiv s_2\ \mathsf{rel}\ ps \text{ and } h_{\text{i}}\ ps \text{ is proper}$$

From the assumption $g_{\text{i}}\ \mathsf{safe}_S\ h_{\text{i}}$, we therefore get

$$g_{\text{i}}\ s_1 = \underline{\text{undef}} \text{ and } g_{\text{i}}\ s_2 = \underline{\text{undef}}$$

or

$$g_{\text{i}}\ s_1 \neq \underline{\text{undef}} \text{ and } g_{\text{i}}\ s_2 \neq \underline{\text{undef}} \text{ and } g_{\text{i}}\ s_1 \equiv g_{\text{i}}\ s_2\ \mathsf{rel}\ h_{\text{i}}\ ps$$

In the first case, we get

$$\mathsf{cond}(g,\ g_1,\ g_2)\ s_1 = \underline{\text{undef}} \text{ and } \mathsf{cond}(g,\ g_1,\ g_2)\ s_2 = \underline{\text{undef}}$$

and we are finished. In the second case, we get

$$\mathsf{cond}(g,\ g_1,\ g_2)\ s_1 \neq \underline{\text{undef}} \text{ and } \mathsf{cond}(g,\ g_1,\ g_2)\ s_2 \neq \underline{\text{undef}}$$

Furthermore, we have

$$\mathsf{cond}(g,\ g_1,\ g_2)\ s_1 \equiv \mathsf{cond}(g,\ g_1,\ g_2)\ s_2\ \mathsf{rel}\ h_{\text{i}}\ ps$$

Clearly $h_{\text{i}}\ ps \sqsubseteq h_1\ ps \sqcup_{\text{PS}} h_2\ ps$, and using the definition of cond_S and Lemma 8.23 we get

$$\mathsf{cond}(g,\ g_1,\ g_2)\ s_1 \equiv \mathsf{cond}(g,\ g_1,\ g_2)\ s_2\ \mathsf{rel}\ \mathsf{cond}_S(f,\ h_1,\ h_2)\ ps$$

as required. □

We now have the apparatus needed to show the safety of $\mathcal{SS}$ as expressed in Theorem 8.27.

Proof: We shall show that $\mathcal{S}_{\text{ds}}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S]\!]$, and we proceed by structural induction on the statement S.

The case $x := a$: Let s_1, s_2, and ps be given such that

$$s_1 \equiv s_2 \;\mathsf{rel}\; ps \text{ and } \mathcal{SS}[\![x := a]\!]ps \text{ is proper}$$

It then follows from Exercise 8.17 that ps is proper because $\mathcal{SS}[\![x := a]\!]ps$ is. Also, both $\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_1$ and $\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_2$ will be defined so we only have to show that

$$(\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_1)\; y = (\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_2)\; y$$

for all $y \in \mathbf{Var} \cap \mathrm{LOW}(\mathcal{SS}[\![x := a]\!]ps)$. Suppose first that $y \neq x$ and y is in $\mathrm{LOW}(\mathcal{SS}[\![x := a]\!]ps)$. Then $y \in \mathrm{LOW}(ps)$ and it is immediate from the definition of $\mathcal{S}_{\mathrm{ds}}$ that $(\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_1)\; y = (\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_2)\; y$. Suppose next that $y = x$ and x is in $\mathrm{LOW}(\mathcal{SS}[\![x := a]\!]ps)$. Then we use the assumption $s_1 \equiv s_2$ rel ps together with $(\mathcal{SS}[\![x := a]\!]ps)\; x = \textsc{low}$ to get

$$\mathcal{A}[\![a]\!]s_1 = \mathcal{A}[\![a]\!]s_2$$

by Fact 8.25. Hence $(\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_1)\; y = (\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s_2)\; y$ follows also in this case. This proves the required relationship.

The case `skip`: Straightforward.

The case $S_1;S_2$: The induction hypothesis applied to S_1 and S_2 gives

$$\mathcal{S}_{\mathrm{ds}}[\![S_1]\!] \;\mathsf{safe}_S\; \mathcal{SS}[\![S_1]\!] \text{ and } \mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \;\mathsf{safe}_S\; \mathcal{SS}[\![S_2]\!]$$

It follows from Exercise 8.17 that ps $\mathsf{history}$ $\sqsubseteq_{\mathrm{P}}$ $(\mathcal{SS}[\![S_2]\!]ps)$ $\mathsf{history}$ holds for all property states ps. The desired result

$$\mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!] \;\mathsf{safe}_S\; \mathcal{SS}[\![S_2]\!] \circ \mathcal{SS}[\![S_1]\!]$$

then follows from Lemma 8.28.

The case `if` b `then` S_1 `else` S_2: From Exercise 8.26, we have

$$\mathcal{B}[\![b]\!] \;\mathsf{safe}_B\; \mathcal{SB}[\![b]\!]$$

and the induction hypothesis applied to S_1 and S_2 gives

$$\mathcal{S}_{\mathrm{ds}}[\![S_1]\!] \;\mathsf{safe}_S\; \mathcal{SS}[\![S_1]\!]$$

and

$$\mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \;\mathsf{safe}_S\; \mathcal{SS}[\![S_2]\!]$$

The desired result

$$\mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\mathrm{ds}}[\![S_1]\!], \mathcal{S}_{\mathrm{ds}}[\![S_2]\!]) \;\mathsf{safe}_S\; \mathsf{cond}_S(\mathcal{SB}[\![b]\!], \mathcal{SS}[\![S_1]\!], \mathcal{SS}[\![S_2]\!])$$

then follows from Lemma 8.29.

The case `while` b `do` S: We must prove that

$$\mathsf{FIX}\ G\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

where

$$G\ g = \mathsf{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!],\ \mathrm{id})$$

and

$$H\ h = \mathsf{cond}_S(\mathcal{SB}[\![b]\!],\ h \circ \mathcal{SS}[\![S]\!],\ \mathrm{id})$$

To do this, we recall the definition of the least fixed points:

$$\mathsf{FIX}\ G = \bigsqcup\{G^{\mathrm{n}}\ g_0 \mid \mathrm{n} \geq 0\} \text{ where } g_0\ s = \underline{\mathrm{undef}} \text{ for all } s$$

and

$$\mathsf{FIX}\ H = \bigsqcup\{H^{\mathrm{n}}\ h_0 \mid \mathrm{n} \geq 0\} \text{ where } h_0\ ps = \textsc{init} \text{ for all } ps$$

The proof proceeds in two stages. We begin by proving that

$$G^{\mathrm{n}}\ g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H \text{ for all n} \tag{*}$$

and then

$$\mathsf{FIX}\ G\ \mathsf{safe}_S\ \mathsf{FIX}\ H \tag{**}$$

We prove (*) by induction on n. For the base case, we observe that

$$g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

holds trivially since $g_0\ s = \underline{\mathrm{undef}}$ for all states s. For the induction step, we assume that

$$G^{\mathrm{n}}\ g_0\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

and we shall prove the result for n+1. We have

$$\mathcal{B}[\![b]\!]\ \mathsf{safe}_B\ \mathcal{SB}[\![b]\!]$$

from Exercise 8.26,

$$\mathcal{S}_{\mathrm{ds}}[\![S]\!]\ \mathsf{safe}_S\ \mathcal{SS}[\![S]\!]$$

from the induction hypothesis applied to the body of the `while`-loop, and it is clear that

$$\mathrm{id}\ \mathsf{safe}_S\ \mathrm{id}$$

By Exercise 8.17, we also have

$$ps\ \mathsf{history} \sqsubseteq_{\mathrm{P}} ((\mathsf{FIX}\ H)\ ps)\ \mathsf{history}$$

for all property states ps. We then obtain

$$\mathsf{cond}(\mathcal{B}[\![b]\!], (G^{\mathrm{n}}\ g_0)\circ\mathcal{S}_{\mathrm{ds}}[\![S]\!], \mathrm{id})\ \mathsf{safe}_S\ \mathsf{cond}_S(\mathcal{SB}[\![b]\!], (\mathsf{FIX}\ H)\circ\mathcal{SS}[\![S]\!], \mathrm{id})$$

from Lemmas 8.28 and 8.29, and this is indeed the desired result since the right-hand side amounts to H ($\mathsf{FIX}\ H$), which equals $\mathsf{FIX}\ H$.

Finally, we must show (**). This amounts to showing

$$\bigsqcup Y\ \mathsf{safe}_S\ \mathsf{FIX}\ H$$

where $Y = \{G^{\mathrm{n}}\ g_0 \mid \mathrm{n} \geq 0\}$. So assume that

$$s_1 \equiv s_2\ \mathsf{rel}\ ps \text{ and } (\mathsf{FIX}\ H)\ ps \text{ is proper}$$

Since $g\ \mathsf{safe}_S\ \mathsf{FIX}\ H$ holds for all $g \in Y$ by (*), we get that either

$$g\ s_1 = \underline{\mathrm{undef}} \text{ and } g\ s_2 = \underline{\mathrm{undef}}$$

or

$$g\ s_1 \neq \underline{\mathrm{undef}} \text{ and } g\ s_2 \neq \underline{\mathrm{undef}} \text{ and } g\ s_1 \equiv g\ s_2\ \mathsf{rel}\ (\mathsf{FIX}\ H)\ ps$$

If $(\bigsqcup Y)\ s_1 = \underline{\mathrm{undef}}$, then $g\ s_1 = \underline{\mathrm{undef}}$ for all $g \in Y$ and thereby $g\ s_2 = \underline{\mathrm{undef}}$ for all $g \in Y$ so that $(\bigsqcup Y)\ s_2 = \underline{\mathrm{undef}}$. Similarly, $(\bigsqcup Y)\ s_2 = \underline{\mathrm{undef}}$ will imply that $(\bigsqcup Y)\ s_1 = \underline{\mathrm{undef}}$. So consider now the case where $(\bigsqcup Y)\ s_1 \neq \underline{\mathrm{undef}}$ as well as $(\bigsqcup Y)\ s_2 \neq \underline{\mathrm{undef}}$, and let $x \in \mathbf{Var} \cap \mathrm{LOW}((\mathsf{FIX}\ H)\ ps)$. By Lemma 5.25, we have

$$\mathrm{graph}(\bigsqcup Y) = \bigcup\{\,\mathrm{graph}\ g \mid g \in Y\}$$

and $(\bigsqcup Y)\ s_{\mathrm{i}} \neq \underline{\mathrm{undef}}$ therefore shows the existence of an element g_{i} in Y such that $g_{\mathrm{i}}\ s_{\mathrm{i}} \neq \underline{\mathrm{undef}}$ and $(\bigsqcup Y)\ s_{\mathrm{i}} = g_{\mathrm{i}}\ s_{\mathrm{i}}$ (for i = 1, 2). Since Y is a chain, either $g_1 \sqsubseteq g_2$ or $g_2 \sqsubseteq g_1$, so let g be the larger of the two. We then have

$$\begin{aligned}
((\textstyle\bigsqcup Y)\ s_1)\ x &= (g_1\ s_1)\ x && \text{as } (\textstyle\bigsqcup Y)\ s_1 = g_1\ s_1\\
&= (g\ s_1)\ x && \text{as } g_1 \sqsubseteq g \text{ and } g_1\ s_1 \neq \underline{\mathrm{undef}}\\
&= (g\ s_2)\ x && \text{as } g\ s_1 \equiv g\ s_2\ \mathsf{rel}\ (\mathsf{FIX}\ H)\ ps\\
&= (g_2\ s_2)\ x && \text{as } g_2 \sqsubseteq g \text{ and } g_2\ s_2 \neq \underline{\mathrm{undef}}\\
&= ((\textstyle\bigsqcup Y)\ s_2)\ x && \text{as } (\textstyle\bigsqcup Y)\ s_2 = g_2\ s_2
\end{aligned}$$

as required. This finishes the proof of the theorem. □

Exercise 8.30

Extend the proof of the theorem to incorporate the analysis developed for `repeat` S `until` b in Exercise 8.15. □

Exercise 8.31

When specifying $\mathcal{SS}$ in the previous section, we rejected the possibility of using

$$\mathsf{cond}'_S(f,\ h_1,\ h_2)\ ps = (h_1\ ps) \sqcup_{\mathrm{PS}} (h_2\ ps)$$

rather than cond_S. Formally show that the analysis obtained by using cond'_S rather than cond_S cannot be correct in the sense of Theorem 8.27. Hint: Consider the statement S_{12} of Example 8.9. □

Exercise 8.32

In the exercise above we saw that cond_S could not be simplified so as to ignore the test for whether the condition is dubious or not. Now consider the following remedy:

$$\mathsf{cond}'_S(f,\ h_1,\ h_2)\ ps$$
$$= \begin{cases} (h_1\ ps) \sqcup_{\mathrm{PS}} (h_2\ ps) \quad \text{if } f\ ps = \mathrm{LOW} \\ ((h_1\ ps[\mathsf{history}{\mapsto}\mathrm{HIGH}]) \sqcup_{\mathrm{PS}} (h_2\ ps[\mathsf{history}{\mapsto}\mathrm{HIGH}]))[\mathsf{history}{\mapsto}\mathrm{ps}\ \mathsf{history}] \\ \qquad\qquad\qquad \text{if } f\ ps = \mathrm{HIGH} \end{cases}$$

Give an example statement where cond'_S is preferable to cond_S. Does the safety proof carry through when cond_S is replaced by cond'_S? If not, suggest how to weaken the safety predicate such that another safety result can be proved to hold. □

Exercise 8.33 (*)

Prove the correctness of the dependency analysis of Exercise 8.19. □

Exercise 8.34 (**)

Write down a specification of a live variables analysis and prove it correct. □

9 Axiomatic Program Verification

The kinds of semantics we have seen so far specify the meaning of programs, although they may also be used to prove that given programs possess certain properties. We may distinguish among several classes of properties: *partial correctness properties* are properties expressing that *if* a given program terminates, *then* there will be a certain relationship between the initial and the final values of the variables. Thus a partial correctness property of a program need *not* ensure that it terminates. This is contrary to *total correctness properties*, which express that the program *will* terminate *and* that there will be a certain relationship between the initial and the final values of the variables. Thus we have

$$\text{partial correctness} + \text{termination} = \text{total correctness}$$

Yet another class of properties is concerned with the *resources* used when executing the program; an example is the *time* used to execute the program on a particular machine. In this chapter, we shall focus on the partial correctness properties; the following chapter will return to the properties that take resources into account.

9.1 Direct Proofs of Program Correctness

In this section, we shall give some examples that prove partial correctness of statements based directly on the operational and denotational semantics. We shall prove that the factorial statement

$$\texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}$$

is partially correct; that is, *if* the statement terminates, *then* the final value of `y` will be the factorial of the initial value of `x`.

Natural Semantics

Using *natural semantics* of Section 2.1, the partial correctness of the factorial statement can be formalized as follows:

For all states s and s', if

$$\langle \texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)},\ s\rangle \to s'$$

then $s'\ \texttt{y} = (s\ \texttt{x})!$ and $s\ \texttt{x} > \mathbf{0}$

This is indeed a partial correctness property because the statement does not terminate if the initial value s `x` of `x` is non-positive.

The proof proceeds in three stages:

Stage 1: We prove that the body of the `while` loop satisfies:

$$\begin{array}{l}\text{if } \langle \texttt{y := y}\star\texttt{x; x := x}-\texttt{1},\ s\rangle \to s'' \text{ and } s''\ \texttt{x} > \mathbf{0}\\ \text{then } (s\ \texttt{y}) \star (s\ \texttt{x})! = (s''\ \texttt{y}) \star (s''\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}\end{array} \qquad (*)$$

Stage 2: We prove that the `while` loop satisfies:

$$\begin{array}{l}\text{if } \langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)},\ s\rangle \to s''\\ \text{then } (s\ \texttt{y}) \star (s\ \texttt{x})! = s''\ \texttt{y} \text{ and } s''\ \texttt{x} = \mathbf{1} \text{ and } s\ \texttt{x} > \mathbf{0}\end{array} \qquad (**)$$

Stage 3: We prove the partial correctness property for the complete program:

$$\begin{array}{l}\text{if } \langle \texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)},\ s\rangle \to s'\\ \text{then } s'\ \texttt{y} = (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}\end{array} \qquad (***)$$

In each of the three stages, the derivation tree of the given transition is inspected in order to prove the property.

In the *first stage*, we consider the transition

$$\langle \texttt{y := y}\star\texttt{x; x := x}-\texttt{1},\ s\rangle \to s''$$

According to [comp$_{\rm ns}$], there will be transitions

$$\langle \texttt{y := y}\star\texttt{x},\ s\rangle \to s' \text{ and } \langle \texttt{x := x}-\texttt{1},\ s'\rangle \to s''$$

for some s'. From the axiom [ass$_{\rm ns}$], we then get that $s' = s[\texttt{y}\mapsto\mathcal{A}[\![\texttt{y}\star\texttt{x}]\!]s]$ and that $s'' = s'[\texttt{x}\mapsto\mathcal{A}[\![\texttt{x}-\texttt{1}]\!]s']$. Combining these results, we have

$$s'' = s[\mathtt{y}\mapsto(s\ \mathtt{y})\star(s\ \mathtt{x})][\mathtt{x}\mapsto(s\ \mathtt{x})-1]$$

Assuming that $s''\ \mathtt{x} > \mathbf{0}$, we can then calculate

$$(s''\ \mathtt{y}) \star (s''\ \mathtt{x})! = ((s\ \mathtt{y}) \star (s\ \mathtt{x})) \star ((s\ \mathtt{x})-1)! = (s\ \mathtt{y}) \star (s\ \mathtt{x})!$$

and since $s\ \mathtt{x} = (s''\ \mathtt{x}) + \mathbf{1}$ this shows that (*) does indeed hold.

In the *second stage*, we proceed by induction on the shape of the derivation tree for

$$\langle \mathtt{while}\ \neg(\mathtt{x}=1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s\rangle \to s'$$

One of two axioms and rules could have been used to construct this derivation. If [while$_{\rm ns}^{\rm ff}$] has been used, then $s' = s$ and $\mathcal{B}[\![\neg(\mathtt{x}=1)]\!]s = \mathbf{ff}$. This means that $s'\ \mathtt{x} = \mathbf{1}$, and since $\mathbf{1}! = \mathbf{1}$ we get the required $(s\ \mathtt{y}) \star (s\ \mathtt{x})! = s\ \mathtt{y}$ and $s\ \mathtt{x} > \mathbf{0}$. This proves (**).

Next assume that [while$_{\rm ns}^{\rm tt}$] is used to construct the derivation. Then it must be the case that $\mathcal{B}[\![\neg(\mathtt{x}=1)]\!]s = \mathbf{tt}$ and

$$\langle \mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1,\ s\rangle \to s''$$

and

$$\langle \mathtt{while}\ \neg(\mathtt{x}=1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s''\rangle \to s'$$

for some state s''. The induction hypothesis applied to the latter derivation gives that

$$(s''\ \mathtt{y}) \star (s''\ \mathtt{x})! = s'\ \mathtt{y} \text{ and } s'\ \mathtt{x} = \mathbf{1} \text{ and } s''\ \mathtt{x} > \mathbf{0}$$

From (*) we get that

$$(s\ \mathtt{y}) \star (s\ \mathtt{x})! = (s''\ \mathtt{y}) \star (s''\ \mathtt{x})! \text{ and } s\ \mathtt{x} > \mathbf{0}$$

Putting these results together, we get

$$(s\ \mathtt{y}) \star (s\ \mathtt{x})! = s'\ \mathtt{y} \text{ and } s'\ \mathtt{x} = \mathbf{1} \text{ and } s\ \mathtt{x} > \mathbf{0}$$

This proves (**) and thereby the second stage of the proof is completed.

Finally, consider the *third stage* of the proof and the derivation

$$\langle \mathtt{y} := 1;\ \mathtt{while}\ \neg(\mathtt{x}=1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s\rangle \to s'$$

According to [comp$_{\rm ns}$], there will be a state s'' such that

$$\langle \mathtt{y} := \mathbf{1},\ s\rangle \to s''$$

and

$$\langle \mathtt{while}\ \neg(\mathtt{x}=1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}-1),\ s''\rangle \to s'$$

From axiom [ass$_{\rm ns}$], we see that $s'' = s[\mathtt{y}\mapsto\mathbf{1}]$, and from (**) we get that $s''\ \mathtt{x} > \mathbf{0}$ and therefore $s\ \mathtt{x} > \mathbf{0}$. Hence $(s\ \mathtt{x})! = (s''\ \mathtt{y}) \star (s''\ \mathtt{x})!$ holds, and using (**) we get

$$(s\ \texttt{x})! = (s''\ \texttt{y}) \star (s''\ \texttt{x})! = s'\ \texttt{y}$$

as required. This proves the partial correctness of the factorial statement.

Exercise 9.1

Use the natural semantics to prove the partial correctness of the statement

$$\texttt{z := 0; while y}\leq\texttt{x do (z := z+1; x := x}-\texttt{y)}$$

That is, prove that

if the statement terminates in s' when executed from a state s
with $s\ \texttt{x} > \mathbf{0}$ and $s\ \texttt{y} > \mathbf{0}$,
then $s'\ \texttt{z} = (s\ \texttt{x})\ \textbf{div}\ (s\ \texttt{y})$ and $s'\ \texttt{x} = (s\ \texttt{x})\ \textbf{mod}\ (s\ \texttt{y})$

where **div** is integer division and **mod** is the modulo operation. □

Exercise 9.2

Use the natural semantics to prove the following *total correctness* property for the factorial program: for all states s

if $s\ \texttt{x} > \mathbf{0}$ then there exists a state s' such that

$s'\ \texttt{y} = (s\ \texttt{x})!$ and

$\langle\texttt{y := 1; while}\ \neg\texttt{(x=1) do (y := y}\star\texttt{x; x := x}-\texttt{1)},\ s\rangle \to s'$

Structural Operational Semantics

The partial correctness of the factorial statement can also be established using the *structural operational semantics* of Section 2.2. The property is then reformulated as:

For all states s and s', if

$$\langle\texttt{y := 1; while}\ \neg\texttt{(x=1) do (y := y}\star\texttt{x; x := x}-\texttt{1)},\ s\rangle \Rightarrow^* s'$$

then $s'\ \texttt{y} = (s\ \texttt{x})!$ and $s\ \texttt{x} > \mathbf{0}$

Again it is worthwhile to approach the proof in stages:

Stage 1: We prove by induction on the length of derivation sequences that

if $\langle\texttt{while}\ \neg\texttt{(x=1) do (y := y}\star\texttt{x; x := x}-\texttt{1)},\ s\rangle \Rightarrow^{\mathrm{k}} s'$

then $s'\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})!$ and $s'\ \texttt{x} = \mathbf{1}$ and $s\ \texttt{x} > \mathbf{0}$

Stage 2: We prove that

$$\text{if } \langle \texttt{y := 1; while } \neg(\texttt{x=1}) \texttt{ do (y := y}\star\texttt{x; x := x}-\texttt{1)}, s\rangle \Rightarrow^* s'$$

$$\text{then } s' \texttt{ y} = (s \texttt{ x})! \text{ and } s \texttt{ x} > \mathbf{0}$$

Exercise 9.3

Complete the proof of stages 1 and 2. □

Denotational Semantics

We shall now use the *denotational semantics* of Chapter 5 to prove partial correctness properties of statements. The idea is to formulate the property as a *predicate* ψ on the ccpo (**State** $\hookrightarrow$ **State**, $\sqsubseteq$); that is,

$$\psi\text{: } (\textbf{State} \hookrightarrow \textbf{State}) \to \mathbf{T}$$

As an example, the partial correctness of the factorial statement will be written as

$$\psi_{fac}(\mathcal{S}_{\text{ds}}[\![\texttt{y := 1; while } \neg(\texttt{x=1}) \texttt{ do (y := y}\star\texttt{x; x := x}-\texttt{1)}]\!]) = \mathbf{tt}$$

where the predicate ψ_{fac} is defined by

$$\begin{array}{lll} \psi_{fac}(g) = \mathbf{tt} & \text{if and only if} & \text{for all states } s \text{ and } s', \\ & & \text{if } g\ s = s' \\ & & \text{then } s' \texttt{ y} = (s \texttt{ x})! \text{ and } s \texttt{ x} > \mathbf{0} \end{array}$$

A predicate ψ: $D \to \mathbf{T}$ defined on a ccpo $(D,\sqsubseteq)$ is called an *admissible predicate* if and only if we have

$$\text{if } \psi\ d = \mathbf{tt} \text{ for all } d \in Y \text{ then } \psi(\bigsqcup Y) = \mathbf{tt}$$

for every chain Y in D. Thus, if ψ holds on all the elements of the chain, then it also holds on the least upper bound of the chain.

Example 9.4

Consider the predicate ψ'_{fac} defined on **State** $\hookrightarrow$ **State** by

$$\begin{array}{lll} \psi'_{fac}(g) = \mathbf{tt} & \text{if and only if} & \text{for all states } s \text{ and } s', \\ & & \text{if } g\ s = s' \\ & & \text{then } s' \texttt{ y} = (s \texttt{ y}) \star (s \texttt{ x})! \text{ and } s \texttt{ x} > \mathbf{0} \end{array}$$

Then ψ'_{fac} is an admissible predicate. To see this assume that Y is a chain in **State** $\hookrightarrow$ **State** and assume that $\psi'_{fac}\ g = \mathbf{tt}$ for all $g \in Y$. We shall then prove that $\psi'_{fac}(\bigsqcup Y) = \mathbf{tt}$; that is,

$$(\bigsqcup Y)\ s = s' \quad \text{implies} \quad s'\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

From Lemma 5.25, we have $\text{graph}(\bigsqcup Y) = \bigcup\{\,\text{graph}(g) \mid g \in Y\}$. We have assumed that $(\bigsqcup Y)\ s = s'$ so Y cannot be empty and $\langle s, s'\rangle \in \text{graph}(g)$ for some $g \in Y$. But then

$$s'\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

as $\psi'_{fac}\ g = \mathbf{tt}$ for all $g \in Y$. This proves that ψ'_{fac} is indeed an admissible predicate. □

For admissible predicates, we have the following induction principle, called *fixed point induction.*

Theorem 9.5

Let $(D,\sqsubseteq)$ be a ccpo, let $f\colon D \to D$ be a continuous function, and let ψ be an admissible predicate on D. If for all $d \in D$

$$\psi\ d = \mathbf{tt} \quad \text{implies} \quad \psi(f\ d) = \mathbf{tt}$$

then $\psi(\mathsf{FIX}\ f) = \mathbf{tt}$.

Proof: We shall first note that

$$\psi\ \bot = \mathbf{tt}$$

holds by admissibility of ψ (applied to the chain $Y = \emptyset$). By induction on n we can then show that

$$\psi(f^{\mathrm{n}}\ \bot) = \mathbf{tt}$$

using the assumptions of the theorem. By admissibility of ψ (applied to the chain $Y = \{\,f^{\mathrm{n}}\ \bot \mid \mathrm{n} \geq 0\,\}$), we then have

$$\psi(\mathsf{FIX}\ f) = \mathbf{tt}$$

This completes the proof. □

We are now in a position where we can prove the partial correctness of the factorial statement. The first observation is that

$$\mathcal{S}_{\mathrm{ds}}[\![\texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}]\!]s = s'$$

if and only if

$$\mathcal{S}_{\mathrm{ds}}[\![\texttt{while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}]\!](s[\texttt{y}\mapsto\mathbf{1}]) = s'$$

Thus it is sufficient to prove that

$$\psi'_{fac}(\mathcal{S}_{\mathrm{ds}}[\![\texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do}\ (\texttt{y} := \texttt{y}\star\texttt{x};\ \texttt{x} := \texttt{x}-1)]\!]) = \mathbf{tt} \qquad (*)$$

(where ψ'_{fac} is defined in Example 9.4) as this will imply that

$$\psi_{fac}(\mathcal{S}_{\mathrm{ds}}[\![\texttt{y} := 1;\ \texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do}\ (\texttt{y} := \texttt{y}\star\texttt{x};\ \texttt{x} := \texttt{x}-1)]\!]) = \mathbf{tt}$$

We shall now reformulate (*) slightly to bring ourselves to a position where we can use fixed point induction. Using the definition of $\mathcal{S}_{\mathrm{ds}}$ in Table 5.1, we have

$$\mathcal{S}_{\mathrm{ds}}[\![\texttt{while}\ \neg(\texttt{x}=1)\ \texttt{do}\ (\texttt{y} := \texttt{y}\star\texttt{x};\ \texttt{x} := \texttt{x}-1)]\!] = \mathsf{FIX}\ F$$

where the functional F is defined by

$$F\ g = \mathsf{cond}(\mathcal{B}[\![\neg(\texttt{x}=1)]\!],\ g \circ \mathcal{S}_{\mathrm{ds}}[\![\texttt{y} := \texttt{y}\star\texttt{x};\ \texttt{x} := \texttt{x}-1]\!],\ \mathrm{id})$$

Using the semantic equations defining $\mathcal{S}_{\mathrm{ds}}$, we can rewrite this definition as

$$(F\ g)\ s = \begin{cases} s & \text{if } s\ x = \mathbf{1} \\ g(s[\texttt{y}\mapsto(s\ \texttt{y})\star(s\ \texttt{x})][\texttt{x}\mapsto(s\ \texttt{x})-1]) & \text{otherwise} \end{cases}$$

We have already seen that F is a continuous function (for example, in the proof of Proposition 5.47), and from Example 9.4 we have that ψ'_{fac} is an admissible predicate. Thus we see from Theorem 9.5 that (*) follows if we show that

$$\psi'_{fac}\ g = \mathbf{tt} \text{ implies } \psi'_{fac}(F\ g) = \mathbf{tt}$$

To prove this implication, assume that $\psi'_{fac}\ g = \mathbf{tt}$; that is, for all s and s'

$$\text{if } g\ s = s' \text{ then } s'\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

We shall prove that $\psi'_{fac}(F\ g) = \mathbf{tt}$, that is for all states s and s'

$$\text{if } (F\ g)\ s = s' \text{ then } s'\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

Inspecting the definition of F, we see that there are two cases. First assume that $s\ \texttt{x} = \mathbf{1}$. Then $(F\ g)\ s = s$ and clearly $s\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})!$ and $s\ \texttt{x} > \mathbf{0}$. Next assume that $s\ \texttt{x} \neq \mathbf{1}$. Then

$$(F\ g)\ s = g(s[\texttt{y}\mapsto(s\ \texttt{y})\star(s\ \texttt{x})][\texttt{x}\mapsto(s\ \texttt{x})-\mathbf{1}])$$

From the assumptions about g, we then get that

$$s'\ \texttt{y} = ((s\ \texttt{y})\star(s\ \texttt{x})) \star ((s\ \texttt{x})-\mathbf{1})! \text{ and } (s\ \texttt{x})-\mathbf{1} > \mathbf{0}$$

so that the desired result

$$s'\ \texttt{y} = (s\ \texttt{y}) \star (s\ \texttt{x})! \text{ and } s\ \texttt{x} > \mathbf{0}$$

follows.

Exercise 9.6

Repeat Exercise 9.1 using the denotational semantics. □

9.2 Partial Correctness Assertions

One may argue that the proofs above are too detailed to be practically useful; the reason is that they are too closely connected with the semantics of the programming language. One may therefore want to capture the *essential properties* of the various constructs so that it would be less demanding to conduct proofs about given programs. Of course, the choice of "essential properties" will determine the sort of properties that we may accomplish proving. In this section, we shall be interested in partial correctness properties, and therefore the "essential properties" of the various constructs will not include termination.

The idea is to specify properties of programs as *assertions*, or claims, about them. An assertion is a triple of the form

$$\{ P \}\ S\ \{ Q \}$$

where S is a statement and P and Q are predicates. Here P is called the *precondition* and Q is called the *postcondition*. Intuitively, the meaning of $\{ P \}\ S\ \{ Q \}$ is that

if P holds in the initial state, and

if the execution of S terminates when started in that state,

then Q will hold in the state in which S halts

Note that for $\{ P \}\ S\ \{ Q \}$ to hold we do *not* require that S halt when started in states satisfying P — merely that *if* it does halt, *then* Q holds in the final state.

Logical Variables

As an example, we may write

$$\{ \texttt{x=n} \}\ \texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y := x}\star\texttt{y; x := x}-\texttt{1})\ \{ \texttt{y=n!} \land \texttt{n>0} \}$$

to express that if the value of `x` is equal to the value of `n` *before* the factorial program is executed, then the value of `y` will be equal to the factorial of the value of `n` *after* the execution of the program has terminated (if indeed it terminates). Here `n` is a special variable called a *logical* variable, and these logical variables must not appear in any statement considered. The role of these variables is to "remember" the initial values of the program variables. Note that if we replace the postcondition $\texttt{y}=\texttt{n}! \wedge \texttt{n}>0$ by the new postcondition $\texttt{y}=\texttt{x}! \wedge \texttt{x}>0$, then the assertion above will express a relationship between the final value of `y` and the final value of `x`, and this is not what we want. The use of logical variables solves the problem because it allows us to refer to initial values of variables.

We shall thus distinguish between two kinds of variables:

– program variables and

– logical variables.

The states will determine the values of both kinds of variables, and since logical variables do not occur in programs, their values will always be the same. In the case of the factorial program, we know that the value of `n` is the same in the initial state as in the final state. The precondition $\texttt{x} = \texttt{n}$ expresses that `n` has the same value as `x` in the initial state. Since the program will not change the value of `n`, the postcondition $\texttt{y} = \texttt{n}!$ will express that the final value of `y` is equal to the factorial of the initial value of `x`.

The Assertion Language

There are two approaches concerning how to specify the preconditions and postconditions of the assertions:

– the intensional approach versus

– the extensional approach.

In the *intensional approach*, the idea is to introduce an explicit language called an *assertion language* and then the conditions will be formulae of that language. This assertion language is in general much more powerful than the boolean expressions, **Bexp**, introduced in Chapter 1. In fact, the assertion language has to be very powerful indeed in order to be able to express all the preconditions and postconditions that may interest us; we shall return to this in the next section. The approach we shall follow is the *extensional approach*, and it is a kind of shortcut. The idea is that the conditions are predicates; that is, functions in **State** $\rightarrow$ **T**. Thus the meaning of $\{\ P\ \}\ S\ \{\ Q\ \}$ may be reformulated as saying that if P holds on a state s and if S executed from state s results in the state s' then Q holds on s'. We can write any predicates we like and therefore the expressiveness problem mentioned above does not arise.

Each boolean expression b defines a predicate $\mathcal{B}[\![b]\!]$. We shall feel free to let b include logical variables as well as program variables, so the precondition

$\mathtt{x} = \mathtt{n}$ used above is an example of a boolean expression. To ease the readability, we introduce the following notation:

$P_1 \wedge P_2$	for	P where $P\ s = (P_1\ s)$ and $(P_2\ s)$
$P_1 \vee P_2$	for	P where $P\ s = (P_1\ s)$ or $(P_2\ s)$
$\neg P$	for	P' where $P'\ s = \neg(P\ s)$
$P[x{\mapsto}\mathcal{A}[\![a]\!]]$	for	P' where $P'\ s = P\ (s[x{\mapsto}\mathcal{A}[\![a]\!]s])$
$P_1 \Rightarrow P_2$	for	$\forall s \in$ **State**: $P_1\ s$ implies $P_2\ s$

When it is convenient, but not when defining formal inference rules, we shall dispense with $\mathcal{B}[\![\cdots]\!]$ and $\mathcal{A}[\![\cdots]\!]$ inside square brackets as well as within preconditions and postconditions.

Exercise 9.7

Show that

– $\mathcal{B}[\![b[x{\mapsto}a]]\!] = \mathcal{B}[\![b]\!][x{\mapsto}\mathcal{A}[\![a]\!]]$ for all b and a,

– $\mathcal{B}[\![b_1 \wedge b_2]\!] = \mathcal{B}[\![b_1]\!] \wedge \mathcal{B}[\![b_2]\!]$ for all b_1 and b_2, and

– $\mathcal{B}[\![\neg b]\!] = \neg\mathcal{B}[\![b]\!]$ for all b. □

The Inference System

The partial correctness assertions will be specified by an inference system consisting of a set of axioms and rules. The formulae of the inference system have the form

$$\{\ P\ \}\ S\ \{\ Q\ \}$$

where S is a statement in the language **While** and P and Q are predicates. The axioms and rules are summarized in Table 9.1 and will be explained below. The inference system specifies an *axiomatic semantics* for **While**.

The axiom for assignment statements is

$$\{\ P[x{\mapsto}\mathcal{A}[\![a]\!]]\ \}\ x := a\ \{\ P\ \}$$

This axiom assumes that the execution of $x := a$ starts in a state s that satisfies $P[x{\mapsto}\mathcal{A}[\![a]\!]]$; that is, in a state s, where $s[x{\mapsto}\mathcal{A}[\![a]\!]s]$ satisfies P. The axiom expresses that if the execution of $x := a$ terminates (which will always be the case), then the final state will satisfy P. From the earlier definitions of the semantics of **While**, we know that the final state will be $s[x{\mapsto}\mathcal{A}[\![a]\!]s]$ so it is easy to see that the axiom is plausible.

[ass$_\text{p}$]	$\{\ P[x\mapsto\mathcal{A}[\![a]\!]]\ \}\ x := a\ \{\ P\ \}$
[skip$_\text{p}$]	$\{\ P\ \}$ `skip` $\{\ P\ \}$
[comp$_\text{p}$]	$\dfrac{\{\ P\ \}\ S_1\ \{\ Q\ \},\quad \{\ Q\ \}\ S_2\ \{\ R\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ R\ \}}$
[if$_\text{p}$]	$\dfrac{\{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S_1\ \{\ Q\ \},\quad \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\ Q\ \}}$
[while$_\text{p}$]	$\dfrac{\{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}}$
[cons$_\text{p}$]	$\dfrac{\{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}}\quad$ if $P \Rightarrow P'$ and $Q' \Rightarrow Q$

Table 9.1 Axiomatic system for partial correctness

For `skip`, the axiom is

$$\{\ P\ \}\ \texttt{skip}\ \{\ P\ \}$$

Thus, if P holds before `skip` is executed, then it also holds afterwards. This is clearly plausible, as `skip` does nothing.

Axioms [ass$_\text{p}$] and [skip$_\text{p}$] are really *axiom schemes* generating separate axioms for each choice of predicate P. The meaning of the remaining constructs is given by rules of inference rather than axiom schemes. Each such rule specifies a way of deducing an assertion about a compound construct from assertions about its constituents. For composition, the rule is

$$\frac{\{\ P\ \}\ S_1\ \{\ Q\ \},\quad \{\ Q\ \}\ S_2\ \{\ R\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ R\ \}}$$

This says that if P holds prior to the execution of $S_1;\ S_2$ and if the execution terminates, then we can conclude that R holds in the final state provided that there is a predicate Q for which we can deduce that

– if S_1 is executed from a state where P holds and if it terminates, then Q will hold for the final state, and that

– if S_2 is executed from a state where Q holds and if it terminates, then R will hold for the final state.

The rule for the conditional is

$$\frac{\{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S_1\ \{\ Q\ \},\quad \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\ Q\ \}}$$

The rule says that if `if` b `then` S_1 `else` S_2 is executed from a state where P holds and if it terminates, then Q will hold for the final state provided that we can deduce that

– if S_1 is executed from a state where P and b hold and if it terminates, then Q holds on the final state, and that

– if S_2 is executed from a state where P and $\neg b$ hold and if it terminates, then Q holds on the final state.

The rule for the iterative statement is

$$\frac{\{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}}$$

The predicate P is called an *invariant* for the `while`-loop, and the idea is that it will hold *before* and *after* each execution of the body S of the loop. The rule says that if additionally b is true before each execution of the body of the loop, then $\neg b$ will be true when the execution of the `while`-loop has terminated.

To complete the inference system, we need one more rule of inference

$$\frac{\{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P\ \}\ S\ \{\ Q\ \}} \quad \text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q$$

This rule says that we can strengthen the precondition P' and weaken the postcondition Q'. This rule is often called the *rule of consequence.*

Note that Table 9.1 specifies a set of axioms and rules just like the tables defining the operational semantics in Chapter 2. The analogue of a derivation tree will now be called an *inference tree* since it shows how to infer that a certain property holds. Thus the leaves of an inference tree will be instances of axioms and the internal nodes will correspond to instances of rules. We shall say that the inference tree gives a *proof* of the property expressed by its root. We shall write

$$\vdash_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \}$$

for the provability of the assertion $\{\ P\ \}\ S\ \{\ Q\ \}$. An inference tree is called *simple* if it is an instance of one of the axioms and otherwise it is called *composite.*

Example 9.8

Consider the statement `while true do skip`. From [skip$_{\mathrm{p}}$] we have (omitting the $\mathcal{B}[\![\cdots]\!]$)

$$\vdash_{\mathrm{p}} \{\ \texttt{true}\ \}\ \texttt{skip}\ \{\ \texttt{true}\ \}$$

Since ($\texttt{true} \wedge \texttt{true}$) $\Rightarrow$ $\texttt{true}$, we can apply the rule of consequence [cons_p] and get

$$\vdash_\text{p} \{\ \texttt{true} \wedge \texttt{true}\ \}\ \texttt{skip}\ \{\ \texttt{true}\ \}$$

Hence, by the rule [while_p], we get

$$\vdash_\text{p} \{\ \texttt{true}\ \}\ \texttt{while true do skip}\ \{\ \neg\texttt{true} \wedge \texttt{true}\ \}$$

We have that $\neg\texttt{true} \wedge \texttt{true} \Rightarrow \texttt{true}$, so by applying [$\text{cons}_\text{p}$] once more we get

$$\vdash_\text{p} \{\ \texttt{true}\ \}\ \texttt{while true do skip}\ \{\ \texttt{true}\ \}$$

The inference above can be summarized by the following inference tree:

$$\cfrac{\cfrac{\cfrac{\{\ \texttt{true}\ \}\ \texttt{skip}\ \{\ \texttt{true}\ \}}{\{\ \texttt{true} \wedge \texttt{true}\ \}\ \texttt{skip}\ \{\ \texttt{true}\ \}}}{\{\ \texttt{true}\ \}\ \texttt{while true do skip}\ \{\ \neg\texttt{true} \wedge \texttt{true}\ \}}}{\{\ \texttt{true}\ \}\ \texttt{while true do skip}\ \{\ \texttt{true}\ \}}$$

It is now easy to see that we cannot claim that $\{\ P\ \}\ S\ \{\ Q\ \}$ means that S will terminate in a state satisfying Q when it is started in a state satisfying P. For the assertion $\{\ \texttt{true}\ \}\ \texttt{while true do skip}\ \{\ \texttt{true}\ \}$ this reading would mean that the program would always terminate, and clearly this is not the case. □

Example 9.9

To illustrate the use of the axiomatic semantics for verification, we shall prove the assertion

$$\{\ \texttt{x} = \texttt{n}\ \}\ \texttt{y := 1; while}\ \neg(\texttt{x}=1)\ \texttt{do}\ (\texttt{y := y}\star\texttt{x; x := x}-1)\ \{\ \texttt{y} = \texttt{n}! \wedge \texttt{n} > 0\ \}$$

where, for the sake of readability, we write $\texttt{y} = \texttt{n}! \wedge \texttt{n} > 0$ for the predicate P given by

$$P\ s = (s\ \texttt{y} = (s\ \texttt{n})! \wedge s\ \texttt{n} > \mathbf{0})$$

The inference of this assertion proceeds in a number of stages. First we define the predicate INV that is going to be the invariant of the $\texttt{while}$-loop:

$$INV\ s = (s\ \texttt{x} > \mathbf{0}\ \text{implies}\ ((s\ \texttt{y}) \star (s\ \texttt{x})! = (s\ \texttt{n})!\ \text{and}\ s\ \texttt{n} \geq s\ \texttt{x}))$$

We shall then consider the body of the loop. Using [ass_p] we get

$$\vdash_\text{p} \{\ INV[\texttt{x}\mapsto\texttt{x}-1]\ \}\ \texttt{x := x}-1\ \{\ INV\ \}$$

Similarly, we get

$$\vdash_p \{\ (INV[\mathtt{x}{\mapsto}\mathtt{x}-1])[\mathtt{y}{\mapsto}\mathtt{y}{\star}\mathtt{x}]\ \}\ \mathtt{y} := \mathtt{y} \star \mathtt{x}\ \{\ INV[\mathtt{x}{\mapsto}\mathtt{x}-1]\ \}$$

We can now apply the rule [comp$_p$] to the two assertions above and get

$$\vdash_p \{\ (INV[\mathtt{x}{\mapsto}\mathtt{x}-1])[\mathtt{y}{\mapsto}\mathtt{y}{\star}\mathtt{x}]\ \}\ \mathtt{y} := \mathtt{y} \star \mathtt{x};\ \mathtt{x} := \mathtt{x}-1\ \{\ INV\ \}$$

It is easy to verify that

$$(\neg(\mathtt{x}{=}1) \wedge INV) \Rightarrow (INV[\mathtt{x}{\mapsto}\mathtt{x}-1])[\mathtt{y}{\mapsto}\mathtt{y}{\star}\mathtt{x}]$$

so using the rule [cons$_p$] we get

$$\vdash_p \{\ \neg(\mathtt{x} = 1) \wedge INV\ \}\ \mathtt{y} := \mathtt{y} \star \mathtt{x};\ \mathtt{x} := \mathtt{x}-1\ \{\ INV\ \}$$

We are now in a position to use the rule [while$_p$] and get

$$\vdash_p \{\ INV\ \}\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}-1)\ \{\neg(\neg(\mathtt{x} = 1)) \wedge INV\ \}$$

Clearly we have

$$\neg(\neg(\mathtt{x} = 1)) \wedge INV \Rightarrow \mathtt{y} = \mathtt{n}! \wedge \mathtt{n} > 0$$

so applying rule [cons$_p$] we get

$$\vdash_p \{\ INV\ \}\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}-1)\ \{\ \mathtt{y} = \mathtt{n}! \wedge \mathtt{n} > 0\ \}$$

We shall now apply the axiom [ass$_p$] to the statement `y := 1` and get

$$\vdash_p \{\ INV[\mathtt{y}{\mapsto}1]\ \}\ \mathtt{y} := 1\ \{\ INV\ \}$$

Using that

$$\mathtt{x} = \mathtt{n} \Rightarrow INV[\mathtt{y}{\mapsto}1]$$

together with [cons$_p$], we get

$$\vdash_p \{\ \mathtt{x} = \mathtt{n}\ \}\ \mathtt{y} := 1\ \{\ INV\ \}$$

Finally, we can use the rule [comp$_p$] and get

$$\vdash_p \{\ \mathtt{x} = \mathtt{n}\ \}\ \mathtt{y} := 1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}-1)\ \{\ \mathtt{y} = \mathtt{n}! \wedge \mathtt{n} > 0\ \}$$

as required. □

Exercise 9.10

Specify a formula expressing the partial correctness property of the program of Exercise 9.1. Construct an inference tree giving a proof of this property using the inference system of Table 9.1. □

Exercise 9.11

Suggest an inference rule for `repeat` S `until` b. You are not allowed to rely on the existence of a `while`-construct in the language. □

Exercise 9.12

Suggest an inference rule for `for` $x := a_1$ `to` a_2 `do` S. You are not allowed to rely on the existence of a `while`-construct in the language. □

Properties of the Semantics

In the operational and denotational semantics, we defined a notion of two programs being semantically equivalent. We can define a similar notion for the axiomatic semantics: two programs S_1 and S_2 are *provably equivalent* according to the axiomatic semantics of Table 9.1 if for all preconditions P and postconditions Q we have

$$\vdash_{\mathrm{p}} \{ P \}\ S_1\ \{ Q \} \quad \text{if and only if} \quad \vdash_{\mathrm{p}} \{ P \}\ S_2\ \{ Q \}$$

Exercise 9.13

Show that the following statements of **While** are provably equivalent in the sense above:

– S; `skip` and S

– S_1; (S_2; S_3) and (S_1; S_2); S_3 □

Proofs of properties of the axiomatic semantics will often proceed by *induction on the shape of the inference tree*:

Induction on the Shape of Inference Trees

1: Prove that the property holds for all the simple inference trees by showing that it holds for the *axioms* of the inference system.

2: Prove that the property holds for all composite inference trees: For each *rule* assume that the property holds for its premises (this is called the *induction hypothesis*) and that the conditions of the rule are satisfied and then prove that it also holds for the conclusion of the rule.

Exercise 9.14 (**)

Using the inference rule for `repeat` S `until` b given in Exercise 9.11, show that `repeat` S `until` b is provably equivalent to S; `while` $\neg b$ `do` S. Hint: It is not too hard to show that what is provable about `repeat` S `until` b is also provable about S; `while` $\neg b$ `do` S. □

Exercise 9.15

Show that $\vdash_{\rm p} \{ P \}\ S\ \{ \texttt{true} \}$ for all statements S and properties P. □

9.3 Soundness and Completeness

We shall now address the relationship between the inference system of Table 9.1 and the operational and denotational semantics of the previous chapters. We shall prove the following.

- The inference system is *sound*: if some partial correctness property can be proved using the inference system, then it does indeed hold according to the semantics.
- The inference system is *complete*: if some partial correctness property does hold according to the semantics, then we can also find a proof for it using the inference system.

The completeness result can only be proved because we use the extensional approach, where preconditions and postconditions are arbitrary predicates. In the intensional approach, we only have a weaker result; we shall return to this later in this section.

As the operational and denotational semantics are equivalent, we only need to consider one of them here and we shall choose the natural semantics. The partial correctness assertion $\{ P \}\ S\ \{ Q \}$ is said to be *valid* if and only if

for all states s, if $P\ s = \mathbf{tt}$ and $\langle S,s\rangle \to s'$ for some s' then $Q\ s' = \mathbf{tt}$

and we shall write this as

$$\models_{\rm p} \{ P \}\ S\ \{ Q \}$$

The soundness property is then expressed by

$$\vdash_{\rm p} \{ P \}\ S\ \{ Q \} \ \text{ implies } \ \models_{\rm p} \{ P \}\ S\ \{ Q \}$$

and the completeness property is expressed by

$$\models_{\rm p} \{ P \}\ S\ \{ Q \} \ \text{ implies } \ \vdash_{\rm p} \{ P \}\ S\ \{ Q \}$$

We have the following theorem.

Theorem 9.16

For all partial correctness assertions $\{\ P\ \}\ S\ \{\ Q\ \}$, we have

$$\models_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \} \ \text{ if and only if } \ \vdash_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \}$$

It is customary to prove the soundness and completeness results separately.

Soundness

Lemma 9.17

The inference system of Table 9.1 is sound; that is, for every partial correctness formula $\{\ P\ \}\ S\ \{\ Q\ \}$, we have

$$\vdash_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \} \text{ implies } \models_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \}$$

Proof: The proof is by induction on the shape of the inference tree used to infer $\vdash_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \}$. This amounts to nothing but a formalization of the intuitions we gave when introducing the axioms and rules.

The case [ass$_{\mathrm{p}}$]: We shall prove that the axiom is valid, so suppose that

$$\langle x := a,\ s\rangle \rightarrow s'$$

and $(P[x{\mapsto}\mathcal{A}[\![a]\!]])\ s = \mathbf{tt}$. We shall then prove that $P\ s' = \mathbf{tt}$. From [ass$_{\mathrm{ns}}$] we get that $s' = s[x{\mapsto}\mathcal{A}[\![a]\!]s]$, and from $(P[x{\mapsto}\mathcal{A}[\![a]\!]])\ s = \mathbf{tt}$ we get that P $(s[x{\mapsto}\mathcal{A}[\![a]\!]s]) = \mathbf{tt}$. Thus $P\ s' = \mathbf{tt}$, as was to be shown.

The case [skip$_{\mathrm{p}}$]: This case is immediate using the clause [skip$_{\mathrm{ns}}$].

The case [comp$_{\mathrm{p}}$]: We assume that

$$\models_{\mathrm{p}} \{\ P\ \}\ S_1\ \{\ Q\ \} \text{ and } \models_{\mathrm{p}} \{\ Q\ \}\ S_2\ \{\ R\ \}$$

and we have to prove that $\models_{\mathrm{p}} \{\ P\ \}\ S_1;\ S_2\ \{\ R\ \}$. So consider arbitrary states s and s'' such that $P\ s = \mathbf{tt}$ and

$$\langle S_1;S_2,\ s\rangle \rightarrow s''$$

From [comp$_{\mathrm{ns}}$], we get that there is a state s' such that

$$\langle S_1,\ s\rangle \rightarrow s' \quad \text{and} \quad \langle S_2,\ s'\rangle \rightarrow s''$$

From $\langle S_1,\ s\rangle \rightarrow s'$, $P\ s = \mathbf{tt}$, and $\models_{\mathrm{p}} \{\ P\ \}\ S_1\ \{\ Q\ \}$, we get $Q\ s' = \mathbf{tt}$. From $\langle S_2,\ s'\rangle \rightarrow s''$, $Q\ s' = \mathbf{tt}$, and $\models_{\mathrm{p}} \{\ Q\ \}\ S_2\ \{\ R\ \}$, it follows that $R\ s'' = \mathbf{tt}$ as was to be shown.

The case [if$_{\mathrm{p}}$]: Assume that

$$\models_{\mathrm{p}} \{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S_1\ \{\ Q\ \} \text{ and } \models_{\mathrm{p}} \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}\ S_2\ \{\ Q\ \}$$

To prove $\models_{\mathrm{p}} \{\ P\ \}$ `if` b `then` S_1 `else` S_2 $\{\ Q\ \}$, consider arbitrary states s and s' such that $P\ s = \mathbf{tt}$ and

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \rightarrow s'$$

There are two cases. If $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, then we get $(\mathcal{B}[\![b]\!] \wedge P)\ s = \mathbf{tt}$, and from $[\text{if}_{\text{ns}}]$ we have

$$\langle S_1,\ s\rangle \rightarrow s'$$

From the first assumption, we therefore get $Q\ s' = \mathbf{tt}$. If $\mathcal{B}[\![b]\!]s = \mathbf{ff}$, the result follows in a similar way from the second assumption.

The case $[\text{while}_{\text{p}}]$: Assume that

$$\models_{\mathrm{p}} \{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S\ \{\ P\ \}$$

To prove $\models_{\mathrm{p}} \{\ P\ \}$ `while` b `do` S $\{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}$, consider arbitrary states s and s'' such that $P\ s = \mathbf{tt}$ and

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \rightarrow s''$$

and we shall show that $(\neg\mathcal{B}[\![b]\!] \wedge P)\ s'' = \mathbf{tt}$. We shall now proceed by induction on the shape of the derivation tree in the natural semantics. One of two cases applies. If $\mathcal{B}[\![b]\!]s = \mathbf{ff}$, then $s'' = s$ according to $[\text{while}^{\text{ff}}_{\text{ns}}]$ and clearly $(\neg\mathcal{B}[\![b]\!] \wedge P)\ s'' = \mathbf{tt}$ as required. Next consider the case where $\mathcal{B}[\![b]\!]s = \mathbf{tt}$ and

$$\langle S,\ s\rangle \rightarrow s' \quad \text{and} \quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \rightarrow s''$$

for some state s'. Thus $(\mathcal{B}[\![b]\!] \wedge P)\ s = \mathbf{tt}$ and we can then apply the assumption $\models_{\mathrm{p}} \{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S\ \{\ P\ \}$ and get that $P\ s' = \mathbf{tt}$. The induction hypothesis can now be applied to the derivation $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \rightarrow s''$ and gives that $(\neg\mathcal{B}[\![b]\!] \wedge P)\ s'' = \mathbf{tt}$. This completes the proof of this case.

The case $[\text{cons}_{\text{p}}]$: Suppose that

$$\models_{\mathrm{p}} \{\ P'\ \}\ S\ \{\ Q'\ \} \text{ and } P \Rightarrow P' \ \text{ and } \ Q' \Rightarrow Q$$

To prove $\models_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \}$, consider states s and s' such that $P\ s = \mathbf{tt}$ and

$$\langle S,\ s\rangle \rightarrow s'$$

Since $P\ s = \mathbf{tt}$ and $P \Rightarrow P'$ we also have $P'\ s = \mathbf{tt}$ and the assumption then gives us that $Q'\ s' = \mathbf{tt}$. From $Q' \Rightarrow Q$, we therefore get $Q\ s' = \mathbf{tt}$, as was required. ☐

Exercise 9.18

Show that the inference rule for `repeat` S `until` b suggested in Exercise 9.11 preserves validity. Argue that this means that the entire proof system consisting of the axioms and rules of Table 9.1 together with the rule of Exercise 9.11 is sound. □

Exercise 9.19

Define $\models' \{ P \}\ S\ \{ Q \}$ to mean that

for all states s such that $P\ s = \mathbf{tt}$ there exists a state s'

such that $Q\ s' = \mathbf{tt}$ and $\langle S,\ s\rangle \to s'$

Show that it is *not* the case that $\vdash_{\mathrm{p}} \{ P \}\ S\ \{ Q \}$ implies $\models' \{ P \}\ S\ \{ Q \}$ and conclude that the proof system of Table 9.1 cannot be sound with respect to this definition of validity. □

Completeness (in the Extensional Approach)

Before turning to the proof of the completeness result, we shall consider a special predicate $\mathrm{wlp}(S,\ Q)$ defined for each statement S and predicate Q:

$$\mathrm{wlp}(S,\ Q)\ s = \mathbf{tt}$$

if and only if

$$\text{for all states } s'\text{: if } \langle S,\ s\rangle \to s' \text{ then } Q\ s' = \mathbf{tt}$$

The predicate is called the *weakest liberal precondition* for Q and it satisfies the following fact.

Fact 9.20

For every statement S and predicate Q, we have

– $\models_{\mathrm{p}} \{ \mathrm{wlp}(S,\ Q) \}\ S\ \{ Q \}$ (*)

– if $\models_{\mathrm{p}} \{ P \}\ S\ \{ Q \}$ then $P \Rightarrow \mathrm{wlp}(S,\ Q)$ (**)

meaning that $\mathrm{wlp}(S,\ Q)$ is the weakest possible precondition for S and Q.

Proof: To verify that (*) holds, let s and s' be states such that $\langle S,\ s\rangle \to s'$ and $\mathrm{wlp}(S,\ Q)\ s = \mathbf{tt}$. From the definition of $\mathrm{wlp}(S,\ Q)$, we get that $Q\ s' = \mathbf{tt}$ as required. To verify that (**) holds, assume that $\models_{\mathrm{p}} \{ P \}\ S\ \{ Q \}$ and let $P\ s = \mathbf{tt}$. If $\langle S,\ s\rangle \to s'$ then $Q\ s' = \mathbf{tt}$ (because $\models_{\rm p} \{\ P\ \}\ S\ \{\ Q\ \}$) so clearly $\mathrm{wlp}(S{,}Q)\ s = \mathbf{tt}$. $\square$

Exercise 9.21

Prove that the predicate *INV* of Example 9.9 satisfies

$$\mathit{INV} = \mathrm{wlp}(\texttt{while}\ \neg(\texttt{x}{=}1)\ \texttt{do}\ (\texttt{y} := \texttt{y}{\star}\texttt{x};\ \texttt{x} := \texttt{x}{-}1),\ \texttt{y} = \texttt{n}! \wedge \texttt{n} > 0)$$

Exercise 9.22

Another interesting predicate called the *strongest postcondition* for S and P can be defined by

$$\mathrm{sp}(P,\ S)\ s' = \mathbf{tt}$$

if and only if

$$\text{there exists } s \text{ such that } \langle S,\ s\rangle \to s' \text{ and } P\ s = \mathbf{tt}$$

Prove that

– $\models_{\rm p} \{\ P\ \}\ S\ \{\ \mathrm{sp}(P,\ S)\ \}$

– if $\models_{\rm p} \{\ P\ \}\ S\ \{\ Q\ \}$ then $\mathrm{sp}(P,\ S) \Rightarrow Q$

Thus $\mathrm{sp}(P,\ S)$ is the strongest possible postcondition for P and S. $\square$

Lemma 9.23

The inference system of Table 9.1 is complete; that is, for every partial correctness formula $\{\ P\ \}\ S\ \{\ Q\ \}$ we have

$$\models_{\rm p} \{\ P\ \}\ S\ \{\ Q\ \} \text{ implies } \vdash_{\rm p} \{\ P\ \}\ S\ \{\ Q\ \}$$

Proof: The completeness result follows if we can infer

$$\vdash_{\rm p} \{\ \mathrm{wlp}(S,\ Q)\ \}\ S\ \{\ Q\ \} \qquad (*)$$

for all statements S and predicates Q. To see this, suppose that

$$\models_{\rm p} \{\ P\ \}\ S\ \{\ Q\ \}$$

Then Fact 9.20 gives that

$$P \Rightarrow \mathrm{wlp}(S{,}Q)$$

so that (*) and [$\mathrm{cons_p}$] give

$$\vdash_{\rm p} \{\ P\ \}\ S\ \{\ Q\ \}$$

as required.

To prove (*), we proceed by structural induction on the statement S.

The case $x := a$: Based on the natural semantics, it is easy to verify that

$$\text{wlp}(x := a,\ Q) = Q[x \mapsto \mathcal{A}[\![a]\!]]$$

so the result follows directly from [ass$_{\rm p}$].

The case `skip`: Since wlp(`skip`, Q) = Q, the result follows from [skip$_{\rm p}$].

The case $S_1;S_2$: The induction hypothesis applied to S_1 and S_2 gives

$$\vdash_{\rm p} \{\ \text{wlp}(S_2,\ Q)\ \}\ S_2\ \{\ Q\ \}$$

and

$$\vdash_{\rm p} \{\ \text{wlp}(S_1,\ \text{wlp}(S_2,\ Q))\ \}\ S_1\ \{\ \text{wlp}(S_2,\ Q)\ \}$$

so that [comp$_{\rm p}$] gives

$$\vdash_{\rm p} \{\ \text{wlp}(S_1,\ \text{wlp}(S_2,\ Q))\ \}\ S_1;S_2\ \{\ Q\ \}$$

We shall now prove that

$$\text{wlp}(S_1;S_2,\ Q) \Rightarrow \text{wlp}(S_1,\ \text{wlp}(S_2,\ Q))$$

as then [cons$_{\rm p}$] will give the required proof in the inference system. So assume that wlp($S_1;S_2$, Q) $s = \mathbf{tt}$ and we shall show that wlp(S_1, wlp(S_2, Q)) $s = \mathbf{tt}$. This is obvious unless there is a state s' such that $\langle S_1,\ s\rangle \to s'$ and then we must prove that wlp(S_2, Q) $s' = \mathbf{tt}$. However, this is obvious, too, unless there is a state s'' such that $\langle S_2,\ s'\rangle \to s''$ and then we must prove that $Q\ s'' = \mathbf{tt}$. But by [comp$_{\rm ns}$] we have $\langle S_1;S_2,\ s\rangle \to s''$ so that $Q\ s'' = \mathbf{tt}$ follows from wlp($S_1;S_2$, Q) $s = \mathbf{tt}$.

The case `if` b `then` S_1 `else` S_2: The induction hypothesis applied to S_1 and S_2 gives

$$\vdash_{\rm p} \{\ \text{wlp}(S_1,\ Q)\ \}\ S_1\ \{\ Q\ \} \text{ and } \vdash_{\rm p} \{\ \text{wlp}(S_2,\ Q)\ \}\ S_2\ \{\ Q\ \}$$

Define the predicate P by

$$P = (\mathcal{B}[\![b]\!] \wedge \text{wlp}(S_1,\ Q)) \vee (\neg\mathcal{B}[\![b]\!] \wedge \text{wlp}(S_2,\ Q))$$

Then we have

$$(\mathcal{B}[\![b]\!] \wedge P) \Rightarrow \text{wlp}(S_1,\ Q) \text{ and } (\neg\mathcal{B}[\![b]\!] \wedge P) \Rightarrow \text{wlp}(S_2,\ Q)$$

so [cons$_{\rm p}$] can be applied twice and gives

$$\vdash_{\rm p} \{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S_1\ \{\ Q\ \} \text{ and } \vdash_{\rm p} \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}\ S_2\ \{\ Q\ \}$$

Using [if$_\text{p}$], we therefore get

$$\vdash_\text{p} \{\ P\ \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{\ Q\ \}$$

To see that this is the desired result, it suffices to show that

$$\text{wlp}(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ Q) \Rightarrow P$$

and this is straightforward by cases on the value of b.

The case `while` b `do` S: Define the predicate P by

$$P = \text{wlp}(\texttt{while}\ b\ \texttt{do}\ S,\ Q)$$

We first show that

$$(\neg\mathcal{B}[\![b]\!] \wedge P) \Rightarrow Q \qquad (^{**})$$

$$(\mathcal{B}[\![b]\!] \wedge P) \Rightarrow \text{wlp}(S,P) \qquad (^{***})$$

To verify (**), let s be such that $(\neg\mathcal{B}[\![b]\!] \wedge P)\ s = \textbf{tt}$. Then it must be the case that $\langle\texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$ so we have $Q\ s = \textbf{tt}$. To verify (***), let s be such that $(\mathcal{B}[\![b]\!] \wedge P)\ s = \textbf{tt}$ and we shall show that $\text{wlp}(S,P)\ s = \textbf{tt}$. This is obvious unless there is a state s' such that $\langle S,\ s\rangle \to s'$ in which case we shall prove that $P\ s' = \textbf{tt}$. We have two cases. First we assume that $\langle\texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$ for some s''. Then [while$^\text{tt}_\text{ns}$] gives us that $\langle\texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''$ and since $P\ s = \textbf{tt}$ we get that $Q\ s'' = \textbf{tt}$ using Fact 9.20. But this means that $P\ s' = \textbf{tt}$ as was required. In the second case, we assume that $\langle\texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$ does *not* hold for any state s''. But this means that $P\ s' = \textbf{tt}$ holds vacuously and we have finished the proof of (***).

The induction hypothesis applied to the body S of the `while`-loop gives

$$\vdash_\text{p} \{\ \text{wlp}(S,P)\ \}\ S\ \{\ P\ \}$$

and using (***) together with [cons$_\text{p}$] we get

$$\vdash_\text{p} \{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S\ \{\ P\ \}$$

We can now apply the rule [while$_\text{p}$] and get

$$\vdash_\text{p} \{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}$$

Finally, we use (**) together with [cons$_\text{p}$] and get

$$\vdash_\text{p} \{\ P\ \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\ Q\ \}$$

as required. □

Exercise 9.24

Prove that the inference system for **While** extended with `repeat` S `until` b as in Exercise 9.11 is complete. □

Exercise 9.25 (*)

Prove the completeness of the inference system of Table 9.1 using the *strongest postconditions* of Exercise 9.22 rather than the weakest liberal preconditions as used in the proof of Lemma 9.23. □

Exercise 9.26

Define a notion of validity based on the denotational semantics of Chapter 5. Then prove the soundness of the inference system of Table 9.1 using this definition; that is, without using the equivalence between the denotational semantics and the operational semantics. □

Exercise 9.27

Use the definition of validity of Exercise 9.26 to prove the completeness of the inference system of Table 9.1. □

Expressiveness Problems (in the Intensional Approach)

So far we have only considered the extensional approach, where the preconditions and postconditions of the formulae are predicates. In the *intensional approach*, they are formulae of some assertion language $\mathcal{L}$. The axioms and rules of the inference system will be as in Table 9.1, the only difference being that the preconditions and postconditions are formulae of $\mathcal{L}$ and that operations such as $P[x\mapsto\mathcal{A}[\![a]\!]]$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$ are operations on formulae of $\mathcal{L}$.

It will be natural to let $\mathcal{L}$ include the boolean expressions of **While**. The soundness proof of Lemma 9.17 then carries directly over to the intensional approach. Unfortunately, this is not the case for the completeness proof of Lemma 9.23. The reason is that the predicates wlp(S, Q) used as preconditions now have to be represented as formulae of $\mathcal{L}$ and this may not be possible.

To illustrate the problems, let S be a statement, for example a universal program in the sense of recursion theory, that has an undecidable Halting problem. Further, suppose that $\mathcal{L}$ only contains the boolean expressions of **While**. Finally, assume that there is a formula b_S of $\mathcal{L}$ such that for all states s we have

$$\mathcal{B}[\![b_S]\!]\ s = \mathbf{tt} \text{ if and only if } \mathrm{wlp}(S,\ \texttt{false})\ s = \mathbf{tt}$$

Then also $\neg b_S$ is a formula of $\mathcal{L}$. We have

$$\mathcal{B}[\![b_S]\!]\ s = \mathbf{tt} \text{ if and only if the computation of } S \text{ on } s \text{ loops}$$

and hence

$$\mathcal{B}[\![\neg b_S]\!]\ s = \mathbf{tt} \text{ if and only if the computation of } S \text{ on } s \text{ terminates}$$

We now have a contradiction: the assumptions about S ensure that $\mathcal{B}[\![\neg b_S]\!]$ must be an undecidable function; on the other hand, Table 1.2 suggests an obvious algorithm for evaluating $\mathcal{B}[\![\neg b_S]\!]$. Hence our assumption about the existence of b_S must be mistaken. Consequently, we cannot mimic the proof of Lemma 9.23.

The obvious remedy is to extend $\mathcal{L}$ to be a much more powerful language that allows quantification as well. A central concept is that $\mathcal{L}$ must be *expressive* with respect to **While** and its semantics, and one then shows that Table 9.1 is *relatively complete* (in the sense of Cook). It is beyond the scope of this book to go deeper into these matters, but we provide references in Chapter 11.

10
More on Axiomatic Program Verification

Having introduced the axiomatic approach to proving *partial* correctness properties in the previous chapter, we now turn to the more demanding situation where termination guarantees need to be established as well. In the first part of the chapter, we present an axiomatic system for proving *total* correctness properties; that is, properties that in addition to partial correctness also guarantee termination of the program of interest.

In the second part of the chapter, we study the more demanding scenario where we want to reason about the *execution time* of programs. To do so, we first extend the natural semantics of Chapter 2 with a notion of time and then we further develop the total correctness axiomatic system to reason about time: the resulting axiomatic system can be used to prove the *order of magnitude* of the execution time of programs.

10.1 Total Correctness Assertions

In this section, we shall modify the partial correctness axiomatic system to prove *total correctness assertions*, thereby allowing us to reason about termination properties. At the end of the section, we shall discuss extensions of the partial correctness and the total correctness axiomatic systems to handle more language constructs, in particular recursive procedures.

For the total correctness properties, we shall now consider formulae of the form

$$\{\ P\ \}\ S\ \{\ \Downarrow\ Q\ \}$$

The idea is that

if the precondition P is fulfilled

then S is guaranteed to terminate (as recorded by the symbol $\Downarrow$)

and the final state will satisfy the postcondition Q.

This is formalized by defining validity of $\{\ P\ \}\ S\ \{\ \Downarrow\ Q\ \}$ by

$$\models_{\mathrm{t}}\ \{\ P\ \}\ S\ \{\ \Downarrow\ Q\ \}$$

if and only if

for all states s, if $P\ s = \mathbf{tt}$ then there exists s' such that

$$Q\ s' = \mathbf{tt} \text{ and } \langle S,\ s\rangle \to s'$$

The inference system for total correctness assertions is very similar to that for partial correctness assertions, the only difference being that the rule for the `while`-construct has changed. The complete set of axioms and rules is given in Table 10.1. We shall write

$$\vdash_{\mathrm{t}}\ \{\ P\ \}\ S\ \{\ \Downarrow\ Q\ \}$$

if there exists an inference tree with the formula $\{\ P\ \}\ S\ \{\ \Downarrow\ Q\ \}$ as root; that is, if the formula is provably in the inference system.

In the rule [while$_{\mathrm{t}}$], we use a parameterized family $P(\mathbf{z})$ of predicates for the invariant. The idea is that $\mathbf{z}$ is the number of unfoldings of the `while`-loop that will be necessary. So if the `while`-loop does not have to be unfolded at all, then $P(\mathbf{0})$ holds and it must imply that b is false. If the `while`-loop has to be unfolded $\mathbf{z}{+}\mathbf{1}$ times, then $P(\mathbf{z}{+}\mathbf{1})$ holds and b must hold *before* the body of the loop is executed; then $P(\mathbf{z})$ will hold *afterwards* so that we have decreased the total number of times the loop remains to be unfolded. The precondition of the conclusion of the rule expresses that there exists a bound on the number of times the loop has to be unfolded and the postcondition expresses that when the `while`-loop has terminated then no more unfoldings are necessary.

Example 10.1

The total correctness of the factorial statement can be expressed by the assertion

$\{\ \mathtt{x} > \mathtt{0} \land \mathtt{x} = \mathtt{n}\ \}\ \mathtt{y} := \mathtt{1};\ \mathtt{while}\ \neg(\mathtt{x}{=}\mathtt{1})\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1})\ \{\ \Downarrow \mathtt{y} = \mathtt{n}!\ \}$

where $\mathtt{y} = \mathtt{n}!$ is an abbreviation for the predicate

$$P \text{ where } P\ s = (s\ \mathtt{y} = (s\ \mathtt{n})!)$$

[ass$_\text{t}$]	$\{ P[x\mapsto\mathcal{A}[\![a]\!]] \}\ x := a\ \{ \Downarrow P \}$
[skip$_\text{t}$]	$\{ P \}\ \texttt{skip}\ \{ \Downarrow P \}$
[comp$_\text{t}$]	$\dfrac{\{ P \}\ S_1\ \{ \Downarrow Q \},\quad \{ Q \}\ S_2\ \{ \Downarrow R \}}{\{ P \}\ S_1;\ S_2\ \{ \Downarrow R \}}$
[if$_\text{t}$]	$\dfrac{\{ \mathcal{B}[\![b]\!] \wedge P \}\ S_1\ \{ \Downarrow Q \},\quad \{ \neg\mathcal{B}[\![b]\!] \wedge P \}\ S_2\ \{ \Downarrow Q \}}{\{ P \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{ \Downarrow Q \}}$
[while$_\text{t}$]	$\dfrac{\{ P(\mathbf{z}{+}\mathbf{1}) \}\ S\ \{ \Downarrow P(\mathbf{z}) \}}{\{ \exists\mathbf{z}.P(\mathbf{z}) \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{ \Downarrow P(\mathbf{0}) \}}$ where $P(\mathbf{z}{+}\mathbf{1}) \Rightarrow \mathcal{B}[\![b]\!]$, $P(\mathbf{0}) \Rightarrow \neg\mathcal{B}[\![b]\!]$ and $\mathbf{z}$ ranges over natural numbers (that is, $\mathbf{z}{\geq}\mathbf{0}$)
[cons$_\text{t}$]	$\dfrac{\{ P' \}\ S\ \{ \Downarrow Q' \}}{\{ P \}\ S\ \{ \Downarrow Q \}}$ where $P \Rightarrow P'$ and $Q' \Rightarrow Q$

Table 10.1 Axiomatic system for total correctness

In addition to expressing that the final value of y is the factorial of the initial value of x, the assertion also expresses that the program does indeed terminate on all states satisfying the precondition. The inference of this assertion proceeds in a number of stages. First we define the predicate $\mathit{INV}(\mathbf{z})$ that is going to be the invariant of the `while`-loop

$$\mathit{INV}(\mathbf{z})\ s = (s\ \texttt{x} > \mathbf{0} \text{ and } (s\ \texttt{y}) \star (s\ \texttt{x})! = (s\ \texttt{n})! \text{ and } s\ \texttt{x} = \mathbf{z} + \mathbf{1})$$

We shall first consider the body of the loop. Using [ass$_\text{t}$] we get

$$\vdash_\text{t} \{ \mathit{INV}(\mathbf{z})[\texttt{x}\mapsto\texttt{x}-\texttt{1}] \}\ \texttt{x := x}-\texttt{1}\ \{ \Downarrow \mathit{INV}(\mathbf{z}) \}$$

Similarly, we get

$$\vdash_\text{t} \{ (\mathit{INV}(\mathbf{z})[\texttt{x}\mapsto\texttt{x}-\texttt{1}])[\texttt{y}\mapsto\texttt{y}\star\texttt{x}] \}\ \texttt{y := y} \star \texttt{x}\ \{ \Downarrow \mathit{INV}(\mathbf{z})[\texttt{x}\mapsto\texttt{x}-\texttt{1}] \}$$

We can now apply the rule [comp$_\text{t}$] to the two assertions above and get

$$\vdash_\text{t} \{ (\mathit{INV}(\mathbf{z})[\texttt{x}\mapsto\texttt{x}-\texttt{1}])[\texttt{y}\mapsto\texttt{y}\star\texttt{x}] \}\ \texttt{y := y} \star \texttt{x; x := x}-\texttt{1}\ \{ \Downarrow \mathit{INV}(\mathbf{z}) \}$$

It is easy to verify that

$$\mathit{INV}(\mathbf{z}{+}\mathbf{1}) \Rightarrow (\mathit{INV}(\mathbf{z})[\texttt{x}\mapsto\texttt{x}-\texttt{1}])[\texttt{y}\mapsto\texttt{y}\star\texttt{x}]$$

so using the rule [cons$_\text{t}$] we get

$$\vdash_\text{t} \{ \mathit{INV}(\mathbf{z}{+}\mathbf{1}) \}\ \texttt{y := y} \star \texttt{x; x := x}-\texttt{1}\ \{ \Downarrow \mathit{INV}(\mathbf{z}) \}$$

It is straightforward to verify that

$$\mathit{INV}(\mathbf{0}) \Rightarrow \neg(\neg(\mathtt{x}{=}1))$$

and

$$\mathit{INV}(\mathbf{z}{+}\mathbf{1}) \Rightarrow \neg(\mathtt{x}{=}1)$$

Therefore we can use the rule [while$_\mathrm{t}$] and get

$$\vdash_\mathrm{t} \{\ \exists\mathbf{z}.\mathit{INV}(\mathbf{z})\ \}\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}1)\ \{\ \Downarrow \mathit{INV}(\mathbf{0})\ \}$$

We shall now apply the axiom [ass$_\mathrm{t}$] to the statement $\mathtt{y} := 1$ and get

$$\vdash_\mathrm{t} \{\ (\exists\mathbf{z}.\mathit{INV}(\mathbf{z}))[\mathtt{y}{\mapsto}1]\ \}\ \mathtt{y} := 1\ \{\ \Downarrow \exists\mathbf{z}.\mathit{INV}(\mathbf{z})\ \}$$

so using [comp$_\mathrm{t}$] we get

$$\vdash_\mathrm{t}\{\ (\exists\mathbf{z}.\mathit{INV}(\mathbf{z}))[\mathtt{y}{\mapsto}1]\ \}\ \mathtt{y}{:=}1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y}{:=}\mathtt{y}\star\mathtt{x};\ \mathtt{x}{:=}\mathtt{x}{-}1)\ \{\ \Downarrow\mathit{INV}(\mathbf{0})\ \}$$

Clearly we have

$$\mathtt{x} > 0 \land \mathtt{x} = \mathtt{n} \Rightarrow (\exists\mathbf{z}.\mathit{INV}(\mathbf{z}))[\mathtt{y}{\mapsto}\mathbf{1}]$$

and

$$\mathit{INV}(\mathbf{0}) \Rightarrow \mathtt{y} = \mathtt{n}!$$

so applying rule [cons$_\mathrm{t}$] we get

$$\vdash_\mathrm{t}\{\mathtt{x}{>}0 \land \mathtt{x}{=}\mathtt{n}\}\ \mathtt{y} := 1;\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}\star\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}1)\ \{\ \Downarrow \mathtt{y} = \mathtt{n}!\ \}$$

as required. □

Exercise 10.2

Suggest a total correctness inference rule for `repeat` S `until` b. You are not allowed to rely on the existence of a `while`-construct in the programming language. □

Lemma 10.3

The total correctness system of Table 10.1 is sound; that is, for every total correctness formula $\{\ P\ \}\ S\ \{\ \Downarrow Q\ \}$ we have

$$\vdash_\mathrm{t} \{\ P\ \}\ S\ \{\ \Downarrow Q\ \} \text{ implies } \models_\mathrm{t} \{\ P\ \}\ S\ \{\ \Downarrow Q\ \}$$

Proof: The proof proceeds by induction on the shape of the inference tree just as in the proof of Lemma 9.17.

The case [ass$_\mathrm{t}$]: We shall prove that the axiom is valid, so assume that s is such that $(P[x{\mapsto}\mathcal{A}[\![a]\!]])\ s = \mathbf{tt}$ and let $s' = s[x{\mapsto}\mathcal{A}[\![a]\!]s]$. Then [ass$_\mathrm{ns}$] gives

$$\langle x := a,\ s\rangle \to s'$$

and from $(P[x\mapsto\mathcal{A}[\![a]\!]])\ s = \mathbf{tt}$ we get $P\ s' = \mathbf{tt}$ as was to be shown.

The case [skip$_\mathrm{t}$]: This case is immediate.

The case [comp$_\mathrm{t}$]: We assume that

$$\models_\mathrm{t} \{\ P\ \}\ S_1\ \{\ \Downarrow Q\ \} \tag{*}$$

and

$$\models_\mathrm{t} \{\ Q\ \}\ S_2\ \{\ \Downarrow R\ \} \tag{**}$$

and we have to prove that $\models_\mathrm{t} \{\ P\ \}\ S_1;\ S_2\ \{\ \Downarrow R\ \}$. So let s be such that $P\ s = \mathbf{tt}$. From (*) we get that there exists a state s' such that $Q\ s' = \mathbf{tt}$ and

$$\langle S_1,\ s\rangle \to s'$$

Since $Q\ s' = \mathbf{tt}$, we get from (**) that there exists a state s'' such that $R\ s''$ $= \mathbf{tt}$ and

$$\langle S_2,\ s'\rangle \to s''$$

Using [comp$_\mathrm{ns}$, we therefore get

$$\langle S_1;\ S_2,\ s\rangle \to s''$$

and since $R\ s'' = \mathbf{tt}$ we have finished this case.

The case [if$_\mathrm{t}$]: Assume that

$$\models_\mathrm{t} \{\ \mathcal{B}[\![b]\!] \wedge P\ \}\ S_1\ \{\ \Downarrow Q\ \} \tag{*}$$

and

$$\models_\mathrm{t} \{\ \neg\mathcal{B}[\![b]\!] \wedge P\ \}\ S_2\ \{\ \Downarrow Q\ \}$$

To prove $\models_\mathrm{t} \{\ P\ \}$ $\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ $\{\ \Downarrow Q\ \}$, consider a state s such that $P\ s = \mathbf{tt}$. We have two cases. If $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, then $(\mathcal{B}[\![b]\!] \wedge P)\ s = \mathbf{tt}$ and from (*) we get that there is a state s' such that $Q\ s' = \mathbf{tt}$ and

$$\langle S_1,\ s\rangle \to s'$$

From [if$_\mathrm{ns}$] we then get

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'$$

as was to be proved. If $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ the result follows in a similar way from the second assumption.

The case [while$_\mathrm{t}$]: Assume that

$$\models_\mathrm{t} \{\ P(\mathbf{z}{+}\mathbf{1})\ \}\ S\ \{\ \Downarrow P(\mathbf{z})\ \} \tag{*}$$

and

$$P(\mathbf{z}{+}\mathbf{1}) \Rightarrow \mathcal{B}[\![b]\!]$$

and

$$P(\mathbf{0}) \Rightarrow \neg\mathcal{B}[\![b]\!]$$

To prove $\models_{\mathrm{t}} \{\ \exists \mathbf{z}.P(\mathbf{z})\ \}$ `while` b `do` S $\{\ \Downarrow P(\mathbf{0})\ \}$, it is sufficient to prove that for all natural numbers $\mathbf{z}$

$$\begin{array}{l}\text{if } P(\mathbf{z})\ s = \mathbf{tt} \text{ then there exists a state } s' \text{ such that}\\ P(\mathbf{0})\ s' = \mathbf{tt} \text{ and } \langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s'\end{array} \qquad (**)$$

So consider a state s such that $P(\mathbf{z})\ s = \mathbf{tt}$. The proof is now by numerical induction on $\mathbf{z}$.

First assume that $\mathbf{z} = \mathbf{0}$. The assumption $P(\mathbf{0}) \Rightarrow \neg\mathcal{B}[\![b]\!]$ gives that $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ and from [while$^{\mathrm{ff}}_{\mathrm{ns}}$] we get

$$\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s$$

Since $P(\mathbf{0})\ s = \mathbf{tt}$, this proves the base case.

For the induction step assume that (**) holds for all states satisfying $P(\mathbf{z})$ and that $P(\mathbf{z}{+}\mathbf{1})\ s = \mathbf{tt}$. From (*), we get that there is a state s' such that $P(\mathbf{z})\ s' = \mathbf{tt}$ and

$$\langle S,\ s\rangle \to s'$$

The numerical induction hypothesis applied to s' gives that there is some state s'' such that $P(\mathbf{0})\ s'' = \mathbf{tt}$ and

$$\langle \texttt{while } b \texttt{ do } S,\ s'\rangle \to s''$$

Furthermore, the assumption $P(\mathbf{z}{+}\mathbf{1}) \Rightarrow \mathcal{B}[\![b]\!]$ gives $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. We can therefore apply [while$^{\mathrm{tt}}_{\mathrm{ns}}$] and get that

$$\langle \texttt{while } b \texttt{ do } S,\ s\rangle \to s''$$

Since $P(\mathbf{0})\ s'' = \mathbf{tt}$, this completes the proof of (**).

The case [cons$_{\mathrm{t}}$]: Suppose that

$$\models_{\mathrm{t}} \{\ P'\ \}\ S\ \{\ \Downarrow Q'\ \}$$

and

$$P \Rightarrow P' \qquad \text{and} \qquad Q' \Rightarrow Q$$

To prove $\models_{\mathrm{t}} \{\ P\ \}\ S\ \{\ \Downarrow Q\ \}$, consider a state s such that $P\ s = \mathbf{tt}$. Then $P'\ s = \mathbf{tt}$ and there is a state s' such that $Q'\ s' = \mathbf{tt}$ and

$$\langle S,\ s\rangle \to s'$$

However, we also have that $Q\ s' = \mathbf{tt}$ and this proves the result. □

Exercise 10.4

Show that the inference rule for `repeat` S `until` b suggested in Exercise 10.2 preserves validity. Argue that this means that the entire proof system consisting of the axioms and rules of Table 10.1 together with the rule of Exercise 10.2 is sound. □

Exercise 10.5 (*)

Prove that the inference system of Table 10.1 is complete, that is

$$\models_{\mathrm{t}} \{\ P\ \}\ S\ \{\ \Downarrow Q\ \} \text{ implies } \vdash_{\mathrm{t}} \{\ P\ \}\ S\ \{\ \Downarrow Q\ \}$$

Exercise 10.6 (*)

Prove that

$$\text{if } \vdash_{\mathrm{t}} \{\ P\ \}\ S\ \{\ \Downarrow Q\ \} \text{ then } \vdash_{\mathrm{p}} \{\ P\ \}\ S\ \{\ Q\ \}$$

Does the converse result hold? □

Extensions of While

We conclude this section by considering an extension of **While** with non-determinism and (parameterless) procedures. The syntax of the extended language is given by

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1\ ;\ S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid S_1\ \texttt{or}\ S_2 \\ & \mid & \texttt{begin proc}\ p\ \texttt{is}\ S_1;\ S_2\ \texttt{end} \mid \texttt{call}\ p \end{array}$$

Note that in `begin proc` p `is` S_1; S_2 `end` the body of p is S_1 and the body of the construct itself is S_2.

Non-determinism. It is straightforward to handle non-determinism (in the sense of Section 3.1) in the axiomatic approach. The idea is that an assertion holds for S_1 `or` S_2 if the similar assertion holds for S_1 as well as for S_2. The motivation for this is that when reasoning about the statement we have no way of influencing whether S_1 or S_2 is chosen. For partial correctness, we thus extend Table 9.1 with the rule

$$[\mathrm{or_p}] \qquad \frac{\{\ P\ \}\ S_1\ \{\ Q\ \},\ \{\ P\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ S_1\ \texttt{or}\ S_2\ \{\ Q\ \}}$$

For total correctness, we extend Table 10.1 with the rule

$$[\mathrm{or_t}] \qquad \frac{\{\ P\ \}\ S_1\ \{\ \Downarrow Q\ \},\ \{\ P\ \}\ S_2\ \{\ \Downarrow Q\ \}}{\{\ P\ \}\ S_1\ \texttt{or}\ S_2\ \{\ \Downarrow Q\ \}}$$

When dealing with soundness and completeness of these rules, one must be careful to use a semantics that models "non-deterministic choice" in the proper manner. We saw in Section 3.1 that this is the case for structural operational semantics but not for natural semantics. With respect to the structural operational semantics, one can show that the rules above are sound and that the resulting inference systems are complete. If one insists on using the natural semantics, the or-construct would model a kind of "angelic choice" and both rules would be sound. However, only the partial correctness inference system will be complete.

Non-recursive procedures. For the sake of simplicity, we shall restrict our attention to statements with at most one procedure declaration. For non-recursive procedures the idea is that an assertion that holds for the body of the procedure also holds for the calls of the procedure. This motivates extending the partial correctness inference system of Table 9.1 with the rule

$$[\mathrm{call_p}] \qquad \frac{\{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{call}\ p\ \{\ Q\ \}} \quad \text{where } p \text{ is defined by } \texttt{proc}\ p\ \texttt{is}\ S$$

Similarly, the inference system for total correctness in Table 10.1 can be extended with the rule

$$[\mathrm{call_t}] \qquad \frac{\{\ P\ \}\ S\ \{\ \Downarrow Q\ \}}{\{\ P\ \}\ \texttt{call}\ p\ \{\ \Downarrow Q\ \}} \quad \text{where } p \text{ is defined by } \texttt{proc}\ p\ \texttt{is}\ S$$

In both cases, the resulting inference system can be proved sound and complete.

Recursive procedures. The rules above turn out to be insufficient when procedures are allowed to be recursive: in order to prove an assertion for `call` p, one has to prove the assertion for the body of the procedure, and this implies that one has to prove an assertion about each occurrence of `call` p inside the body and so on.

Consider first the case of *partial correctness* assertions. In order to prove some property $\{\ P\ \}$ `call` p $\{\ Q\ \}$, we shall prove the similar property for the body of the procedure but *under the assumption that* $\{\ P\ \}$ `call` p $\{\ Q\ \}$ holds for the recursive calls of p. Often this is expressed by a rule of the form

$$[\text{call}_{\text{p}}^{\text{rec}}]\qquad \frac{\{\ P\ \}\ \texttt{call}\ p\ \{\ Q\ \} \vdash_{\text{p}} \{\ P\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{call}\ p\ \{\ Q\ \}}$$

where p is defined by `proc` p `is` S

The premise of the rule expresses that $\{\ P\ \}\ S\ \{\ Q\ \}$ is provable under the assumption that $\{\ P\ \}$ `call` p $\{\ Q\ \}$ can be proved for the recursive calls present in S. The conclusion expresses that $\{\ P\ \}$ `call` p $\{\ Q\ \}$ holds for all calls of p.

Example 10.7

Consider the statement

```
begin proc fac is ( if x = 1 then skip
                    else (y := x⋆y; x := x−1; call fac) );
      y := 1; call fac
end
```

We want to prove that the final value of `y` is the factorial of the initial value of `x`. We shall prove that

$$\{\ \texttt{x} > \texttt{0} \land \texttt{n} = \texttt{y} \star \texttt{x}!\ \}\ \texttt{call fac}\ \{\ \texttt{y} = \texttt{n}\ \}$$

where $\texttt{x} > \texttt{0} \land \texttt{n} = \texttt{y} \star \texttt{x}!$ is an abbreviation for the predicate P defined by

$$P\ s = (s\ \texttt{x} > \mathbf{0}\ \text{and}\ s\ \texttt{n} = s\ \texttt{y} \star (s\ \texttt{x})!)$$

We assume that

$$\vdash_{\text{p}} \{\ \texttt{x} > \texttt{0} \land \texttt{n} = \texttt{y} \star \texttt{x}!\ \}\ \texttt{call fac}\ \{\ \texttt{y} = \texttt{n}\ \} \qquad (*)$$

holds for the recursive calls of `fac`. We shall then construct a proof of

$$\begin{array}{l}\{\ \texttt{x} > \texttt{0} \land \texttt{n} = \texttt{y} \star \texttt{x}!\ \}\\ \texttt{if x = 1 then skip else (y := x}\star\texttt{y; x := x}-\texttt{1; call fac)} \qquad (**)\\ \{\ \texttt{y} = \texttt{n}\ \}\end{array}$$

and, using $[\text{call}_{\text{p}}^{\text{rec}}]$, we obtain a proof of (*) for all occurrences of `call fac`. To prove (**), we first use the assumption (*) to get

$$\vdash_{\text{p}} \{\ \texttt{x} > \texttt{0} \land \texttt{n} = \texttt{y} \star \texttt{x}!\ \}\ \texttt{call fac}\ \{\ \texttt{y} = \texttt{n}\ \}$$

Then we apply $[\text{ass}_{\text{p}}]$ and $[\text{comp}_{\text{p}}]$ twice and get

$$\vdash_{\mathrm{p}} \quad \{ \; ((\mathtt{x} > 0 \land \mathtt{n} = \mathtt{y} \star \mathtt{x}!)[\mathtt{x}{\mapsto}\mathtt{x}{-}1])[\mathtt{y}{\mapsto}\mathtt{x}{\star}\mathtt{y}] \; \}$$
$$\mathtt{y} := \mathtt{x}{\star}\mathtt{y};\ \mathtt{x} := \mathtt{x}{-}1;\ \mathtt{call\ fac}$$
$$\{ \; \mathtt{y} = \mathtt{n} \; \}$$

We have

$$\neg(\mathtt{x}{=}1) \land (\mathtt{x} > 0 \land \mathtt{n} = \mathtt{y} \star \mathtt{x}!) \Rightarrow ((\mathtt{x} > 0 \land \mathtt{n} = \mathtt{y} \star \mathtt{x}!)[\mathtt{x}{\mapsto}\mathtt{x}{-}1])[\mathtt{y}{\mapsto}\mathtt{x}{\star}\mathtt{y}]$$

so using [$\mathrm{cons_p}$] we get

$$\vdash_{\mathrm{p}} \quad \{ \; \neg(\mathtt{x}{=}1) \land (\mathtt{x} > 0 \land \mathtt{n} = \mathtt{y} \star \mathtt{x}!) \; \}$$
$$\mathtt{y} := \mathtt{x}{\star}\mathtt{y};\ \mathtt{x} := \mathtt{x}{-}1;\ \mathtt{call\ fac}$$
$$\{ \; \mathtt{y} = \mathtt{n} \; \}$$

Using that

$$\mathtt{x}{=}1 \land \mathtt{x} > 0 \land \mathtt{n} = \mathtt{y} \star \mathtt{x}! \Rightarrow \mathtt{y} = \mathtt{n}$$

it is easy to prove

$$\vdash_{\mathrm{p}} \{ \; \mathtt{x}{=}1 \land \mathtt{x} > 0 \land \mathtt{n} = \mathtt{y} \star \mathtt{x}! \; \} \ \mathtt{skip} \ \{ \; \mathtt{y} = \mathtt{n} \; \}$$

so [$\mathrm{if_p}$] can be applied and gives a proof of (**). □

Table 9.1 extended with the rule [$\mathrm{call_p^{rec}}$] can be proved to be sound. However, in order to get a completeness result, the inference system has to be extended with additional rules. To illustrate why this is necessary, consider the following version of the factorial program:

```
begin proc fac is  if x=1 then y := 1
                   else (x := x−1; call fac; x := x+1; y := x⋆y);
      call fac
end
```

Assume that we want to prove that this program does not change the value of `x`; that is,

$$\{ \; \mathtt{x} = \mathtt{n} \; \} \ \mathtt{call\ fac} \ \{ \; \mathtt{x} = \mathtt{n} \; \} \qquad (*)$$

In order to do that, we assume that we have a proof of (*) for the recursive call of `fac` and we have to construct a proof of the property for the body of the procedure. It seems that in order to do so, we must construct a proof of

$$\{ \; \mathtt{x} = \mathtt{n}{-}1 \; \} \ \mathtt{call\ fac} \ \{ \; \mathtt{x} = \mathtt{n}{-}1 \; \}$$

and there are no axioms and rules that allow us to obtain such a proof from (*). We shall not go further into this, but Chapter 11 will provide appropriate references.

The case of *total correctness* is slightly more complicated because we have to bound the number of recursive calls. The rule adopted is

$$[\text{call}_{\text{t}}^{\text{rec}}] \quad \frac{\{\ P(\mathbf{z})\ \}\ \texttt{call}\ p\ \{\ \Downarrow\ Q\ \} \vdash_{\text{t}} \{\ P(\mathbf{z}{+}\mathbf{1})\ \}\ S\ \{\ \Downarrow\ Q\ \}}{\{\ \exists\mathbf{z}.P(\mathbf{z})\ \}\ \texttt{call}\ p\ \{\ \Downarrow\ Q\ \}}$$

where $\neg P(\mathbf{0})$ holds
and $\mathbf{z}$ ranges over the natural numbers (that is, $\mathbf{z}{\geq}\mathbf{0}$)
and where p is defined by $\texttt{proc}\ p\ \texttt{is}\ S$

The premise of this rule expresses that if we assume that we have a proof of $\{\ P(\mathbf{z})\ \}\ \texttt{call}\ p\ \{\ \Downarrow\ Q\ \}$ for all recursive calls of p of depth at most $\mathbf{z}$, then we can prove $\{\ P(\mathbf{z}{+}\mathbf{1})\ \}\ S\ \{\ \Downarrow\ Q\ \}$. The conclusion expresses that for any depth of recursive calls we have a proof of $\{\ \exists\mathbf{z}.P(\mathbf{z})\ \}\ \texttt{call}\ p\ \{\ \Downarrow\ Q\ \}$.

The inference system of Table 10.1 extended with the rule $[\text{call}_{\text{t}}^{\text{rec}}]$ can be proved to be sound. If it is extended with additional rules (as discussed above), it can also be proved to be complete.

10.2 Assertions for Execution Time

A proof system for total correctness can be used to prove that a program does indeed terminate, but it does not say how many resources it needs in order to terminate. We shall now show how to extend the total correctness proof system of Table 10.1 to prove *the order of magnitude of the execution time* of a statement.

It is easy to give some informal guidelines for how to determine the order of magnitude of the execution time:

- assignment: the execution time is $\mathcal{O}(\mathbf{1})$ (that is, it is bounded by a constant),
- skip: the execution time is $\mathcal{O}(\mathbf{1})$,
- composition: the execution time is, to within a constant factor, the sum of the execution times of each of the statements,
- conditional: the execution time is, to within a constant factor, the largest of the execution times of the two branches, and
- iteration: the execution time of the loop is, to within a constant factor, the sum, over all iterations around the loop, of the time to execute the body.

The idea now is to formalize these rules by giving an inference system for reasoning about execution times. To do so, we shall proceed in three stages:

- first we specify the exact time needed to evaluate arithmetic and boolean expressions,
- next we extend the natural semantics of Chapter 2 to count the exact execution time, and
- finally we extend the total correctness proof system to prove the order of magnitude of the execution time of statements.

However, before addressing these issues, we have to fix a *cost model*; that is, we have to determine how to count the cost of the various operations. The actual choice is not so important, but for the sake of simplicity we have based it upon the abstract machine of Chapter 4. The idea is that each instruction of the machine takes one time unit and the time required to execute an arithmetic expression, a boolean expression, or a statement will be the time required to execute the generated code. However, no knowledge of Chapter 4 is required in the sequel.

Exact Execution Times for Expressions

The time needed to evaluate an arithmetic expression is given by a function

$$\mathcal{TA}\colon \mathbf{Aexp} \to \mathbf{Z}$$

so $\mathcal{TA}[\![a]\!]$ is the number of time units required to evaluate a in any state. Similarly, the function

$$\mathcal{TB}\colon \mathbf{Bexp} \to \mathbf{Z}$$

determines the number of time units required to evaluate a boolean expression. These functions are defined in Table 10.2.

Exact Execution Times for Statements

Turning to the execution time for statements, we shall extend the natural semantics of Table 2.1 to specify the time requirements. This is done by extending the transitions to have the form

$$\langle S,\ s\rangle \to^{t} s'$$

meaning that if S is executed from state s, then it will terminate in state s' and exactly t time units will be required for this. The extension of Table 2.1 is fairly straightforward and is given in Table 10.3.

$$\begin{aligned}
\mathcal{TA}[\![n]\!] &= \mathbf{1}\\
\mathcal{TA}[\![x]\!] &= \mathbf{1}\\
\mathcal{TA}[\![a_1 + a_2]\!] &= \mathcal{TA}[\![a_1]\!] + \mathcal{TA}[\![a_2]\!] + \mathbf{1}\\
\mathcal{TA}[\![a_1 \star a_2]\!] &= \mathcal{TA}[\![a_1]\!] + \mathcal{TA}[\![a_2]\!] + \mathbf{1}\\
\mathcal{TA}[\![a_1 - a_2]\!] &= \mathcal{TA}[\![a_1]\!] + \mathcal{TA}[\![a_2]\!] + \mathbf{1}\\[6pt]
\mathcal{TB}[\![\texttt{true}]\!] &= \mathbf{1}\\
\mathcal{TB}[\![\texttt{false}]\!] &= \mathbf{1}\\
\mathcal{TB}[\![a_1 = a_2]\!] &= \mathcal{TA}[\![a_1]\!] + \mathcal{TA}[\![a_2]\!] + \mathbf{1}\\
\mathcal{TB}[\![a_1 \leq a_2]\!] &= \mathcal{TA}[\![a_1]\!] + \mathcal{TA}[\![a_2]\!] + \mathbf{1}\\
\mathcal{TB}[\![\neg b]\!] &= \mathcal{TB}[\![b]\!] + \mathbf{1}\\
\mathcal{TB}[\![b_1 \wedge b_2]\!] &= \mathcal{TB}[\![b_1]\!] + \mathcal{TB}[\![b_2]\!] + \mathbf{1}
\end{aligned}$$

Table 10.2 Exact execution times for expressions

The Inference System

The inference system for proving the order of magnitude of the execution time of statements will have assertions of the form

$$\{\ P\ \}\ S\ \{\ e \Downarrow Q\ \}$$

where P and Q are predicates as in the previous inference systems and e is an arithmetic expression (that is, $e \in \mathbf{Aexp}$). The idea is that

if the execution of S is started in a state satisfying P

then it terminates in a state satisfying Q

and the required execution time is $\mathcal{O}(e)$ (i.e., has order of magnitude e).

So, for example,

$\{\ \texttt{x} = 3\ \}\ \texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}\ \{\ \texttt{1} \Downarrow \texttt{true}\ \}$

expresses that the execution of the factorial statement from a state where `x` has the value **3** has order of magnitude `1`; that is, it is bounded by a constant. Similarly,

$\{\ \texttt{x} > 0\ \}\ \texttt{y := 1; while}\ \neg(\texttt{x=1})\ \texttt{do (y := y}\star\texttt{x; x := x}-\texttt{1)}\ \{\ \texttt{x} \Downarrow \texttt{true}\ \}$

expresses that the execution of the factorial statement on a state where `x` is positive has order of magnitude `x`.

$[\mathrm{ass}_{\mathrm{tns}}]$	$\langle x := a,\ s\rangle \to^{\mathcal{TA}[\![a]\!]+1} s[x \mapsto \mathcal{A}[\![a]\!]s]$	
$[\mathrm{skip}_{\mathrm{tns}}]$	$\langle \texttt{skip},\ s\rangle \to^{1} s$	
$[\mathrm{comp}_{\mathrm{tns}}]$	$\dfrac{\langle S_1,s\rangle \to^{t_1} s',\ \langle S_2,s'\rangle \to^{t_2} s''}{\langle S_1;S_2,\ s\rangle \to^{t_1+t_2} s''}$	
$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{tns}}]$	$\dfrac{\langle S_1,s\rangle \to^{t} s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to^{\mathcal{TB}[\![b]\!]+t+1} s'}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{tns}}]$	$\dfrac{\langle S_2,s\rangle \to^{t} s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to^{\mathcal{TB}[\![b]\!]+t+1} s'}$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\mathrm{while}^{\mathrm{tt}}_{\mathrm{tns}}]$	$\dfrac{\langle S,s\rangle \to^{t} s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to^{t'} s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to^{\mathcal{TB}[\![b]\!]+t+t'+2} s''}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{while}^{\mathrm{ff}}_{\mathrm{tns}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to^{\mathcal{TB}[\![b]\!]+3} s$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Table 10.3 Natural semantics for **While** with exact execution times

Formally, *validity* of the formula $\{\ P\ \}\ S\ \{\ e \Downarrow Q\ \}$ is defined by

$$\models_{\mathrm{e}} \{\ P\ \}\ S\ \{\ e \Downarrow Q\ \}$$

if and only if

> there exists a natural number $\mathbf{k}$ such that for all states s,
>
> if $P\ s = \mathbf{tt}$ then there exists a state s' and a number t such that
>
> $Q\ s' = \mathbf{tt}$, $\langle S,\ s\rangle \to^{t} s'$, and $t \le \mathbf{k} \star (\mathcal{A}[\![e]\!]s)$

Note that the expression e is evaluated in the initial state rather than the final state.

The axioms and rules of the inference system are given in Table 10.4. Provability of the assertion $\{\ P\ \}\ S\ \{\ e \Downarrow Q\ \}$ in the inference system is written

$$\vdash_{\mathrm{e}} \{\ P\ \}\ S\ \{\ e \Downarrow Q\ \}$$

The assignment statement and the `skip` statement can be executed in constant time and therefore we use the arithmetic expression `1`.

The rule $[\mathrm{comp}_{\mathrm{e}}]$ assumes that we have proofs showing that e_1 and e_2 are the orders of magnitude of the execution times for the two statements. However, e_1 expresses the time requirements of S_1 relative to the initial state of S_1 and e_2 expresses the time requirements relative to the initial state of S_2. This means that we cannot simply use $e_1 + e_2$ as the time requirement for S_1; S_2. We have

$[\text{ass}_\text{e}]$	$\{\ P[x\mapsto\mathcal{A}\llbracket a\rrbracket]\ \}\ x := a\ \{\ \mathtt{1} \Downarrow P\ \}$
$[\text{skip}_\text{e}]$	$\{\ P\ \}\ \mathtt{skip}\ \{\ \mathtt{1} \Downarrow P\ \}$
$[\text{comp}_\text{e}]$	$\dfrac{\{\ P \wedge \mathcal{B}\llbracket e_2'\mathtt{=}\mathbf{u}\rrbracket\ \}\ S_1\ \{\ e_1 \Downarrow Q \wedge \mathcal{B}\llbracket e_2\mathtt{\leq}\mathbf{u}\rrbracket\ \},\ \ \{\ Q\ \}\ S_2\ \{\ e_2 \Downarrow R\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ e_1\mathtt{+}e_2' \Downarrow R\ \}}$ where $\mathbf{u}$ is an unused logical variable
$[\text{if}_\text{e}]$	$\dfrac{\{\ \mathcal{B}\llbracket b\rrbracket \wedge P\ \}\ S_1\ \{\ e \Downarrow Q\ \},\ \ \{\ \neg\mathcal{B}\llbracket b\rrbracket \wedge P\ \}\ S_2\ \{\ e \Downarrow Q\ \}}{\{\ P\ \}\ \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \{\ e \Downarrow Q\ \}}$
$[\text{while}_\text{e}]$	$\dfrac{\{\ P(\mathbf{z}\mathbf{+}\mathbf{1}) \wedge \mathcal{B}\llbracket e'\ \mathtt{=}\mathbf{u}\rrbracket\ \}\ S\ \{\ e_1 \Downarrow P(\mathbf{z}) \wedge \mathcal{B}\llbracket e\mathtt{\leq}\mathbf{u}\rrbracket\ \}}{\{\ \exists\mathbf{z}.P(\mathbf{z})\ \}\ \mathtt{while}\ b\ \mathtt{do}\ S\ \{\ e \Downarrow P(\mathbf{0})\ \}}$ where $P(\mathbf{z}\mathbf{+}\mathbf{1}) \Rightarrow \mathcal{B}\llbracket b\rrbracket \wedge \mathcal{B}\llbracket e\mathtt{\geq}e_1\mathtt{+}e'\rrbracket$, $P(\mathbf{0}) \Rightarrow \neg\mathcal{B}\llbracket b\rrbracket \wedge \mathcal{B}\llbracket \mathtt{1}\mathtt{\leq}e\rrbracket$ and $\mathbf{u}$ is an unused logical variable and $\mathbf{z}$ ranges over natural numbers (that is, $\mathbf{z}\mathbf{\geq}\mathbf{0}$)
$[\text{cons}_\text{e}]$	$\dfrac{\{\ P'\ \}\ S\ \{\ e' \Downarrow Q'\ \}}{\{\ P\ \}\ S\ \{\ e \Downarrow Q\ \}}$ where (for some natural number $\mathtt{k}$) $P \Rightarrow P' \wedge \mathcal{B}\llbracket e'\mathtt{\leq}\mathtt{k}\star e\rrbracket$ and $Q' \Rightarrow Q$

Table 10.4 Axiomatic system for order of magnitude of execution time

to replace e_2 with an expression e_2' such that e_2' evaluated in the initial state of S_1 will bound the value of e_2 in the initial state of S_2 (which is the final state of S_1). This is expressed by the extended precondition and postcondition of S_1 using the logical variable $\mathbf{u}$.

The rule [if$_\text{e}$] is fairly straightforward since the time required for the test is constant.

In the rule for the `while`-construct we assume that the execution time is e_1 for the body and is e for the loop itself. As in the rule [comp$_\text{e}$], we cannot just use $e_1 + e$ as the total time required because e_1 refers to the state before the body of the loop is executed and e to the state after the body is executed once. We shall therefore require that there be an expression e' such that e' evaluated before the body will bound e evaluated after the body. Then it must be the case that e satisfies $e \geq e_1 + e'$ because e has to bound the time for executing the `while`-loop independently of the number of times it is unfolded. As we shall see in Example 10.9, this corresponds to the *recurrence equations* that often have to be solved when analysing the execution time of a program. Finally, the rule [cons$_\text{e}$] should be straightforward.

Example 10.8

We shall now prove that the execution time of the factorial statement has order of magnitude `x`. This can be expressed by the following assertion:

$$\{\ \mathtt{x} > \mathtt{0}\ \}\ \mathtt{y} := \mathtt{1};\ \mathtt{while}\ \neg(\mathtt{x}{=}\mathtt{1})\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1})\ \{\ \mathtt{x} \Downarrow \mathtt{true}\ \}$$

The inference of this assertion proceeds in a number of stages. First we define the predicate $\mathit{INV}(\mathbf{z})$; that is, to be the invariant of the `while`-loop

$$\mathit{INV}(\mathbf{z})\ s = (s\ \mathtt{x} > \mathbf{0} \text{ and } s\ \mathtt{x} = \mathbf{z} + \mathbf{1})$$

The logical variables $\mathtt{u}_1$ and $\mathtt{u}_2$ are used for the `while`-loop and the body of the `while`-loop, respectively. We shall first consider the body of the loop. Using [ass_e], we get

$$\vdash_\text{e} \{\ (\mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1)[\mathtt{x}{\mapsto}\mathtt{x}{-}\mathtt{1}]\ \}\ \mathtt{x} := \mathtt{x} - \mathtt{1}\ \{\ \mathtt{1} \Downarrow \mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1\ \}$$

Similarly, we get

$$\begin{aligned}\vdash_\text{e}\quad & \{\ ((\mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1)[\mathtt{x}{\mapsto}\mathtt{x}{-}\mathtt{1}] \wedge \mathtt{1}{\le}\mathtt{u}_2)[\mathtt{y}{\mapsto}\mathtt{y}{\star}\mathtt{x}]\ \}\\ & \mathtt{y} := \mathtt{y} \star \mathtt{x}\\ & \{\ \mathtt{1} \Downarrow (\mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1)[\mathtt{x}{\mapsto}\mathtt{x}{-}\mathtt{1}] \wedge \mathtt{1}{\le}\mathtt{u}_2\ \}\end{aligned}$$

Before applying the rule [comp_e], we have to modify the precondition of the assertion above. We have

$$\begin{aligned}& \mathit{INV}(\mathbf{z}{+}\mathbf{1}) \wedge \mathtt{x}{-}\mathtt{1}{=}\mathtt{u}_1 \wedge \mathtt{1}{=}\mathtt{u}_2\\ & \Rightarrow ((\mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1)[\mathtt{x}{\mapsto}\mathtt{x}{-}\mathtt{1}] \wedge \mathtt{1}{\le}\mathtt{u}_2)[\mathtt{y}{\mapsto}\mathtt{y}{\star}\mathtt{x}]\end{aligned}$$

so using [cons_e] we get

$$\begin{aligned}\vdash_\text{e}\quad & \{\ \mathit{INV}(\mathbf{z}{+}\mathbf{1}) \wedge \mathtt{x}{-}\mathtt{1}{=}\mathtt{u}_1 \wedge \mathtt{1}{=}\mathtt{u}_2\ \}\\ & \mathtt{y} := \mathtt{y} \star \mathtt{x}\\ & \{\ \mathtt{1} \Downarrow (\mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1)[\mathtt{x}{\mapsto}\mathtt{x}{-}\mathtt{1}] \wedge \mathtt{1}{\le}\mathtt{u}_2\ \}\end{aligned}$$

We can now apply [comp_e] and get

$$\begin{aligned}\vdash_\text{e}\quad & \{\ \mathit{INV}(\mathbf{z}{+}\mathbf{1}) \wedge \mathtt{x}{-}\mathtt{1}{=}\mathtt{u}_1\ \}\\ & \mathtt{y} := \mathtt{y} \star \mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1}\\ & \{\ \mathtt{1}{+}\mathtt{1} \Downarrow \mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1\ \}\end{aligned}$$

and using [cons_e] we get

$$\begin{aligned}\vdash_\text{e}\quad & \{\ \mathit{INV}(\mathbf{z}{+}\mathbf{1}) \wedge \mathtt{x}{-}\mathtt{1}{=}\mathtt{u}_1\ \}\\ & \mathtt{y} := \mathtt{y} \star \mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1}\\ & \{\ \mathtt{1} \Downarrow \mathit{INV}(\mathbf{z}) \wedge \mathtt{x}{\le}\mathtt{u}_1\ \}\end{aligned}$$

It is easy to verify that

$$INV(\mathbf{z}{+}\mathbf{1}) \Rightarrow \neg(\mathtt{x} = \mathtt{1}) \wedge \mathtt{x}{\geq}\mathtt{1}{+}(\mathtt{x}{-}\mathtt{1})$$

and

$$INV(\mathbf{0}) \Rightarrow \neg(\neg(\mathtt{x} = \mathtt{1})) \wedge \mathtt{1}{\leq}\mathtt{x}$$

Therefore we can use the rule [while$_\mathrm{e}$] and get

$$\vdash_\mathrm{e} \{\exists\mathbf{z}.INV(\mathbf{z})\}\ \mathtt{while}\ \neg(\mathtt{x}{=}\mathtt{1})\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1})\ \{\ \mathtt{x} \Downarrow INV(\mathbf{0})\}$$

We shall now apply the axiom [ass$_\mathrm{e}$] to the statement `y := 1` and get

$$\vdash_\mathrm{e} \{\ (\exists\mathbf{z}.INV(\mathbf{z}) \wedge \mathtt{1}{\leq}\mathtt{u}_3)[\mathtt{y}{\mapsto}\mathtt{1}]\ \}\ \mathtt{y} := \mathtt{1}\ \{\ \mathtt{1} \Downarrow \exists\mathbf{z}.INV(\mathbf{z}) \wedge \mathtt{1}{\leq}\mathtt{u}_3\ \}$$

We have

$$\mathtt{x}{>}\mathtt{0} \wedge \mathtt{1}{=}\mathtt{u}_3 \Rightarrow (\exists\mathbf{z}.INV(\mathbf{z}) \wedge \mathtt{1}{\leq}\mathtt{u}_3)[\mathtt{y}{\mapsto}\mathtt{1}]$$

so using [cons$_\mathrm{e}$] we get

$$\vdash_\mathrm{e} \{\ \mathtt{x}{>}\mathtt{0} \wedge \mathtt{1}{=}\mathtt{u}_3\ \}\ \mathtt{y} := \mathtt{1}\ \{\ \mathtt{1} \Downarrow \exists\mathbf{z}.INV(\mathbf{z}) \wedge \mathtt{1}{\leq}\mathtt{u}_3\ \}$$

The rule [comp$_\mathrm{e}$] now gives

$$\begin{array}{ll} \vdash_\mathrm{e} & \{\ \mathtt{x}{>}\mathtt{0}\ \} \\ & \mathtt{y} := \mathtt{1};\ \mathtt{while}\ \neg(\mathtt{x}{=}\mathtt{1})\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1}) \\ & \{\ \mathtt{1}{+}\mathtt{x} \Downarrow INV(\mathbf{0})\ \} \end{array}$$

Clearly we have $\mathtt{x}{>}0 \Rightarrow \mathtt{1}{+}\mathtt{x} \leq \mathbf{2}{\star}\mathtt{x}$, and $INV(\mathbf{0}) \Rightarrow \mathtt{true}$, so applying rule [cons$_\mathrm{e}$] we get

$$\begin{array}{ll} \vdash_\mathrm{e} & \{\ \mathtt{x} > 0\ \} \\ & \mathtt{y} := \mathtt{1};\ \mathtt{while}\ \neg(\mathtt{x}{=}\mathtt{1})\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1}) \\ & \{\ \mathtt{x} \Downarrow \mathtt{true}\ \} \end{array}$$

as required. □

Example 10.9

Assume now that we want to determine an arithmetic expression e_fac such that

$$\begin{array}{ll} \vdash_\mathrm{e} & \{\ \mathtt{x} > 0\ \} \\ & \mathtt{y} := \mathtt{1};\ \mathtt{while}\ \neg(\mathtt{x}{=}\mathtt{1})\ \mathtt{do}\ (\mathtt{y} := \mathtt{y}{\star}\mathtt{x};\ \mathtt{x} := \mathtt{x}{-}\mathtt{1}) \\ & \{\ e_\mathrm{fac} \Downarrow \mathtt{true}\ \} \end{array}$$

In other words, we want to determine the order of magnitude of the time required to execute the factorial statement. We can then attempt to construct a proof of the assertion above using the inference system of Table 10.4 with e_{fac} being an unspecified arithmetic expression. The various side conditions of the rules will then specify a set of (in)equations that have to be fulfilled by e_{fac} in order for the proof to exist.

We shall first consider the body of the loop. Very much as in the previous example, we get

$$\vdash_{\text{e}} \{\ \mathit{INV}(\mathbf{z}{+}\mathbf{1}) \wedge e[\mathtt{x}{\mapsto}\mathtt{x}{-}1]{=}\mathtt{u}_1\ \}\ \mathtt{y{:=}y \star x;\ x{:=}x{-}1}\ \{\ 1 \Downarrow \mathit{INV}(\mathbf{z}) \wedge e{\le}\mathtt{u}_1\}$$

where e is the execution time of the `while`-construct. We can now apply the rule [while$_{\text{e}}$] if e fulfils the conditions

$$\mathit{INV}(\mathbf{z}{+}\mathbf{1}) \Rightarrow e{\ge}1{+}e[\mathtt{x}{\mapsto}\mathtt{x}{-}1] \qquad \text{and} \qquad \mathit{INV}(\mathbf{0}) \Rightarrow 1{\le}e \qquad (*)$$

and we will get

$$\vdash_{\text{e}} \{\exists\mathbf{z}.\mathit{INV}(\mathbf{z})\}\ \mathtt{while}\ \neg(\mathtt{x}{=}1)\ \mathtt{do}\ (\mathtt{y := y{\star}x;\ x := x{-}1})\ \{e \Downarrow \mathit{INV}(\mathbf{0})\}$$

The requirement (*) corresponds to the recurrence equation

$$T(\mathtt{x}) = 1 + T(\mathtt{x}{-}1) \qquad \text{and} \qquad T(1) = 1$$

obtained by the standard techniques from execution time analysis. If we take e to be `x`, then (*) is fulfilled. The remainder of the proof is very much as in Exercise 10.8, and we get that e_{fac} must satisfy

$$\mathtt{x} > 0 \Rightarrow \mathtt{x}{+}1 \le \mathbf{k}{\star}e_{\text{fac}} \text{ for some constant } \mathbf{k}$$

so e_{fac} may be taken to be `x`. □

Exercise 10.10

Modify the proof of Lemma 10.3 to show that the inference system of Table 10.4 is sound. □

Exercise 10.11 (**)

Suggest a rule for `while` b `do` S that expresses that its execution time, neglecting constant factors, is the product of the number of times the loop is executed and the maximal execution time for the body of the loop. □

Exercise 10.12

Suggest an inference rule for `repeat` S `until` b. You are not allowed to rely on the existence of a `while`-construct in the language. □

11 Further Reading

In this book, we have covered the basic ingredients in three approaches to semantics:

– operational semantics,

– denotational semantics, and

– axiomatic semantics.

We have concentrated on the rather simple language **While** of `while`-programs in order to more clearly present the *fundamental ideas* behind the approaches, to stress their *relationship* by formulating and proving the relevant theorems, and to illustrate the *applications* of semantics in computer science. The power of the three approaches has been briefly illustrated by various extensions of **While**: non-determinism, parallelism, recursive procedures, and exceptions.

In our choice of applications, we have selected some of the historically important application areas, as well as some of the more promising candidates for future applications:

– the use of semantics for validating prototype implementations of programming languages;

– the use of semantics for verifying program analyses that are part of more advanced implementations of programming languages;

– the use of semantics for verifying security analyses; and

– the use of semantics for verifying useful program properties, including information about execution time.

Hopefully they have convinced the reader that formal semantics is an important tool for reasoning about many aspects of the behaviour of programs and programming languages.

In conclusion, we shall provide a few pointers to the literature where the ideas emerged or where a more comprehensive treatment of language features or theoretical aspects may be found.

Operational Semantics (Chapters 2 and 3)

Structural operational semantics was introduced by Gordon Plotkin [21, 23]. These are standard references and cover a number of features from imperative and functional languages, whereas features from parallel languages are covered in [22]. A more introductory treatment of structural operational semantics is given in [8]. *Natural semantics* is derived from structural operational semantics, and the basic ideas are presented in [3] for a functional language.

Although we have covered many of the essential ideas behind operational semantics, we should like to mention three techniques that had to be omitted.

A technique that is often used when specifying a structural operational semantics is to extend the syntactic component of the configurations with special notation for recording *partially processed constructs*. The inference system will then contain axioms and rules that handle these "extended" configurations. This technique may be used to specify a structural operational semantics of the languages **Block** and **Proc** in Chapter 3 and to specify a structural operational semantics of expressions.

Another technique often used to cut down on the number of inference rules is to use *evaluation contexts*. They are often introduced using a simple grammar for program constructs with "holes" in them. This usually allows the inference rules to be written in a much more succinct manner: each fundamental computational step is taken care of by one rule that can be instantiated to the evaluation context at hand.

Both kinds of operational semantics can easily be extended to cope explicitly with *dynamic errors* (e.g., division by zero). The idea is to extend the set of configurations with special error configurations and then augment the inference system with extra axioms and rules for how to handle these configurations.

Often programs have to fulfil certain conditions in order to be *statically well-formed* and hence preclude certain dynamic errors. These conditions can be formulated using inductively defined predicates and may be integrated with the operational semantics.

Provably Correct Implementation (Chapter 4)

The *correctness of the implementation* of Chapter 4 was a relatively simple proof because it was based on an abstract machine designed for the purpose. In general, when more realistic machines or larger languages are considered, proofs easily become unwieldy, and perhaps for this reason there is no ideal textbook in this area. We therefore only reference [5] for an approach based on natural semantics and [16, 19] for an approach based on denotational semantics.

Denotational Semantics (Chapters 5 and 6)

A general introduction to *denotational semantics* (as developed by C. Strachey and D. Scott) may be found in [24]. It covers denotational semantics for mainly imperative languages and covers the fundamentals of domain theory (including reflexive domains). We should also mention a classic in the field [25] even though the domain theory is based on the (by now obsolete) approach of complete lattices. General background on partially ordered sets may be found in [4].

More recent books include [7], which has wide coverage of denotational and operational semantics, including domain theory, full abstractness, parametric polymorphism, and many other concepts.

We have restricted the treatment of domain theory to what is needed for specifying the denotational semantics of **While**. The benefit of this is that we can restrict ourselves to partial functions between states and thereby obtain a relatively simple theoretical development. The drawback is that it becomes rather cumbersome to verify the existence of semantic specifications for other languages (as evidenced in Chapter 5).

The traditional solution is to develop a *meta-language* for expressing denotational definitions. The theoretical foundation of this language will then ensure that the semantic functions do exist as long as one only uses domains and operations from the meta-language. The benefit of this is obvious; the drawback is that one has to prove a fair amount of results but the efforts are greatly rewarded in the long run. Both [24] and [25] contain such a development.

The denotational approach can handle *abortion* and *non-determinism* using a kind of powerset called powerdomain. Certain kinds of *parallelism* can be handled as well, but for many purposes it is simpler to use a structural operational semantics instead.

The algebraic approach to semantics shares many of the benefits of denotational semantics; we refer to [6] for a recent *executable* approach using the OBJ language.

Program Analysis (Chapters 7 and 8)

A comprehensive treatment of the four main approaches to static program analysis may be found in [20]. It covers data flow analysis, constraint-based analysis, abstract interpretation, and type and effect systems. Each approach is accompanied by proofs of correctness (with respect to an operational semantics) and the necessary algorithms for implementing the analyses.

The formulation of the detection of signs analysis in Chapter 7 is typical of the denotational approach to *abstract interpretation* [10, 19]. More general studies of program transformation are often carried out in the context of partial evaluation [11].

The distinction between forward and backward analyses in Chapter 7 is standard, whereas the distinction between first-order and second-order is due to the authors [15]. The formulation of a general framework for backward analyses is inspired by [17]. The security analysis is patterned after the dependency analysis already present in [18], which again builds on previous work on second-order analyses [15].

Axiomatic Program Verification (Chapters 9 and 10)

A general introduction to *program verification*, and in particular *axiomatic semantics* as introduced by C. A. R. Hoare, may be found in [12]. The presentation covers a flowchart language, a language of `while`-programs and a (first-order) functional language, and also includes a study of expressiveness (as needed for the intensional approach to axiomatic semantics). Many books, including [9], develop axiomatic program verification together with practically motivated examples. Rules for procedures may be found in [2].

A good introduction to the analysis of *resource requirements* of programs is [1]. The formulation of an analysis for verifying execution times in the form of a formal inference system may be found in [14] and the extension to recursive procedures in [13].

We should point out that we have used the extensional approach to specifying the assertions of the inference systems. This allows us to concentrate on the *formulation* of the inference systems without having to worry about the *existence* of the assertions in an explicit assertion language. However, it is more common to use the intensional approach as is done in [12].

A
Review of Notation

We use the following notation:

$\exists$	there exists (e.g., $\exists x : (x > 1)$)
$\forall$	for all (e.g., $\forall x : (x + 1 > x)$)
$\{ x \mid \cdots x \cdots \}$	the set of those x such that $\cdots x \cdots$ holds
$x \in X$	x is a member of the set X
$X \subseteq Y$	set X is contained in set Y (i.e., $\forall x \in X : x \in Y$)
$X \cup Y$	$\{ z \mid z{\in}X$ or $z{\in}Y \}$ (union)
$X \cap Y$	$\{ z \mid z{\in}X$ and $z{\in}Y \}$ (intersection)
$X \setminus Y$	$\{ z \mid z{\in}X$ and $z{\notin}Y \}$ (set difference)
$X \times Y$	$\{ (x, y) \mid x{\in}X$ and $y{\in}Y \}$ (Cartesian product)
$\mathcal{P}(X)$	$\{ Z \mid Z \subseteq X \}$ (powerset)
$\bigcup\mathcal{Y}$	$\{ y \mid \exists Y{\in}\mathcal{Y}\colon y{\in}Y \}$ (so that $\bigcup\{Y_1, Y2\} = Y_1 \cup Y_2$)
$\emptyset$	the empty set
$\mathbf{T}$	$\{$ **tt**, **ff** $\}$ (truth values **tt** (true) and **ff** (false))
$\mathbf{N}$	$\{ \mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots \}$ (natural numbers)
$\mathbf{Z}$	$\{ \cdots, -\mathbf{2}, -\mathbf{1}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \cdots \}$ (integers)

$f{:}X \to Y$	f is a total function from X to Y
$X \to Y$	$\{ f \mid f{:}X \to Y \}$
$f{:}X \hookrightarrow Y$	f is a partial function from X to Y
$X \hookrightarrow Y$	$\{ f \mid f{:}X \hookrightarrow Y \}$

In addition to this, we have special notations for functions, relations, predicates, and transition systems.

Functions

For the application of a function f to an argument x, we generally write $f\ x$ and only occasionally $f(x)$. The effect of a function $f{:}X \to Y$ is expressed by its *graph*

$$\text{graph}(f) = \{\ (x,\ y) \in X \times Y \mid f\ x = y\ \}$$

which is merely an element of $\mathcal{P}(X \times Y)$. The graph of f has the following properties

– $(x,\ y) \in \text{graph}(f)$ and $(x,\ y') \in \text{graph}(f)$ imply $y = y'$, and

– $\forall x \in X{:}\ \exists y \in Y{:}\ (x,\ y) \in \text{graph}(f)$

This expresses the single-valuedness of f and the totality of f. We say that f is *injective* if $f\ x = f\ x'$ implies that $x = x'$.

A *partial* function $g{:}X \hookrightarrow Y$ is a function from a subset X_g of X to Y; that is, $g{:}X_g \to Y$. Again one may define

$$\text{graph}(g) = \{\ (x,\ y) \in X \times Y \mid g\ x = y \text{ and } x \in \mathrm{X}_g\ \}$$

but now only the single-valuedness property is satisfied. We shall write $g\ x = y$ whenever $(x,\ y) \in \text{graph}(g)$ and $g\ x = \underline{\text{undef}}$ whenever $x \notin X_g$; that is, whenever $\neg\exists y \in Y{:}\ (x,\ y) \in \text{graph}(g)$. To distinguish between a function f and a partial function g one often calls f a *total* function. We shall view the partial functions as encompassing the total functions.

For total functions f_1 and f_2, we define their composition $f_2 \circ f_1$ by

$$(f_2 \circ f_1)\ x = f_2(f_1\ x)$$

(Note that the opposite order is sometimes used in the literature.) For partial functions g_1 and g_2, we define $g_2 \circ g_1$ similarly:

$(g_2 \circ g_1)\ x = z$	if there exists y such that $g_1\ x = y$ and $g_2\ y = z$
$(g_2 \circ g_1)\ x = \underline{\text{undef}}$	if $g_1\ x = \underline{\text{undef}}$ or if there exists y such that $g_1\ x = y$ but $g_2\ y = \underline{\text{undef}}$

The identity function $\mathrm{id}: X \to X$ is defined by

$$\mathrm{id}\ x = x$$

Finally, if $f{:}X \to Y$, $x{\in}X$, and $y{\in}Y$, then the function $f[x{\mapsto}y]{:}X \to Y$ is defined by

$$(f[x{\mapsto}y])\ x' = \begin{cases} y & \text{if } x = x' \\ f\ x' & \text{otherwise} \end{cases}$$

A similar notation may be used when f is a partial function.

The function f from numbers to numbers is of *order of magnitude* g, written $\mathcal{O}(g)$, if there exist natural numbers $\mathbf{n}$ and $\mathbf{k}$ such that $\forall x{\geq}\mathbf{n}$: $f\ x \leq \mathbf{k} \cdot (g\ x)$.

Relations

A *relation from X to Y* is a subset of $X \times Y$ (that is, an element of $\mathcal{P}(X \times Y)$). A relation *on* X is a subset of $X \times X$. If $f{:}X \to Y$ or $f{:}X \hookrightarrow Y$, then the graph of f is a relation. (Sometimes a function is identified with its graph, but we shall keep the distinction.) The *identity relation* on X is the relation

$$\mathrm{I}_X = \{\ (x,\ x) \mid x{\in}X\ \}$$

from X to X. When X is clear from the context, we shall omit the subscript X and simply write I.

If $R_1 \subseteq X \times Y$ and $R_2 \subseteq Y \times Z$, the *composition* of R_1 followed by R_2, which we denote by $R_1 \diamond R_2$, is defined by

$$R_1 \diamond R_2 = \{\ (x,\ z) \mid \exists y{\in}Y{:}\ (x,\ y){\in}R_1 \text{ and } (y,\ z){\in}R_2\ \}$$

Note that the order of composition differs from that used for functions,

$$\mathrm{graph}(f_2{\circ}f_1) = \mathrm{graph}(f_1) \diamond \mathrm{graph}(f_2)$$

and that we have the equation

$$\mathrm{I} \diamond R = R \diamond \mathrm{I} = R$$

If R is a relation on X, then the *reflexive transitive closure* is the relation R^* on X defined by

$$R^* = \{\ (x,\ x') \mid \exists \mathrm{n}{\geq}1{:}\ \exists x_1,\ \ldots,\ x_{\mathrm{n}}{:}\ x = x_1 \text{ and } x' = x_{\mathrm{n}}$$
$$\text{and } \forall \mathrm{i}{<}\mathrm{n}{:}\ (x_{\mathrm{i}},\ x_{\mathrm{i}+1}){\in}R\ \}$$

Note that by taking n=1 and $x{=}x'{=}x_1$, it follows that $\mathrm{I}{\subseteq}R^*$. In a similar way, it follows that $R{\subseteq}R^*$. Finally, we define

$$R^+ = R \diamond R^*$$

and observe that $R \subseteq R^+ \subseteq R^*$.

Usually we write $x\ R\ y$ as a more readable notation for $(x, y) \in R$.

Predicates

A *predicate* on X is a function from X to $\mathbf{T}$. If $p{:}X{\to}\mathbf{T}$ is a predicate on X, the relation I_p on X is defined by

$$\mathrm{I}_p = \{\ (x,\ x) \mid x{\in}X \text{ and } p\ x = \mathbf{tt}\ \}$$

Note that $\mathrm{I}_p \subseteq \mathrm{I}$ and that

$$\mathrm{I}_p \diamond R = \{\ (x,\ y) \mid p\ x = \mathbf{tt} \text{ and } (x,\ y){\in}R\ \}$$
$$R \diamond \mathrm{I}_q = \{\ (x,\ y) \mid (x,\ y){\in}R \text{ and } q\ y = \mathbf{tt}\ \}$$

Transition Systems

A *transition system* is a triple of the form

$$(\mathit{\Gamma},\mathrm{T},\ \rhd)$$

where $\mathit{\Gamma}$ is a set of *configurations*, T is a subset of $\mathit{\Gamma}$ called the *terminal* (or *final*) configurations, and $\rhd$ is a relation on $\mathit{\Gamma}$ called a *transition relation.* The relation $\rhd$ must satisfy

$$\forall\gamma{\in}\mathrm{T}{:}\ \forall\gamma'{\in}\mathit{\Gamma}{:}\ \neg(\gamma{\rhd}\gamma')$$

Any configuration γ in $\mathit{\Gamma}\backslash\mathrm{T}$ such that the transition $\gamma{\rhd}\gamma'$ holds for no γ' is called *stuck*.

B
Implementation of Program Analysis

In this appendix, we consider the problem of computing fixed points in program analysis. The whole purpose of program analysis is to get information about programs without actually running, them and it is important that the analyses always terminate. In general, the analysis of a recursive (or iterative) program will itself be recursively defined, and it is therefore important to "solve" this recursion such that termination is ensured.

In general, the setting is as follows. The analysis associates to each program an element h of a complete lattice $(A \to B, \sqsubseteq)$ of abstract values. In the case of an iterative construct such as the `while`-loop, the value h is determined as the least fixed point, $\mathsf{FIX}\ H$, of a continuous functional $H : (A \to B) \to (A \to B)$. Formally, the fixed point of H is given by

$$\mathsf{FIX}\ H = \bigsqcup\{H^{\mathrm{n}}\bot \mid \mathrm{n} \geq 0\}$$

where $\bot$ is the least element of $A \to B$ and $\bigsqcup$ is the least upper bound operation on $A \to B$. The iterands $\{H^{\mathrm{n}}\bot \mid \mathrm{n} \geq 0\}$ form a chain in $A \to B$, and in Exercise 7.8 we have shown that

$$\text{if } H^{k}\bot = H^{k+1}\bot \text{ for some } k \geq 0 \text{ then } \mathsf{FIX}\ H = H^{k}\bot$$

So the obvious algorithm for computing $\mathsf{FIX}\ H$ will be to determine the iterands $H^0\bot,\ H^1\bot, \cdots$ one after the other while testing for stabilization (i.e., equality with the predecessor). When $A \to B$ is finite, this procedure is guaranteed to terminate. The cost of this algorithm depends on

– the number k of iterations needed before stabilization,

– the cost of comparing two iterands, and

– the cost of computing a new iterand.

We shall now study how to minimize the cost of the algorithm above. Most of our efforts are spent on minimizing the number k.

We shall assume that A and B are finite complete lattices, and we shall consider three versions of the framework:

– In the *general framework*, functions of $A \to B$ only are required to be total; this is written $A \to_t B$.

– In the *monotone framework*, functions of $A \to B$ must be monotone; this is written $A \to_m B$.

– In the *completely additive framework*, functions of $A \to B$ must be strict and additive; this is written $A \to_{sa} B$. (We shall explain strict and additive shortly.)

We give precise bounds on the number k of iterations needed to compute the fixed point of an arbitrary continuous functional H.

For many program analyses, including the detection of signs analysis and the security analysis, A and B will be the same (e.g., **PState**). Since a given program always mentions a finite number of variables, we can assume that **Var** is finite so **PState** = **Var** $\to$ **P** is a function space with a finite domain. In the more general development, we shall therefore pay special attention to the case where $A = B = S \to L$ for some non-empty set S with p elements and some finite complete lattice $(L, \sqsubseteq)$. We shall show that the number k of iterations needed to compute the fixed point is at most

– *exponential* in p for the general and the monotone frameworks and

– *quadratic* in p for the completely additive framework.

The results above hold for arbitrary continuous functionals H. A special case is where H is in iterative form:

H is in *iterative form* if it is of the form $H\ h = f \sqcup (h \circ g)$

For strict and additive functions f and g, we then show that k is at most

– *linear* in p, and furthermore

– the fixed point can be computed *pointwise*.

This result is of particular interest for the analysis of **While** since the functional obtained for the `while`-loop often can be written in this form.

B.1 The General and Monotone Frameworks

We shall first introduce some notation. Let $(D, \sqsubseteq)$ be a *finite complete lattice*; that is,

– $\sqsubseteq$ is a partial order on D, and

– each subset Y of D has a least upper bound in D denoted $\bigsqcup Y$.

When $d \sqsubseteq d' \land d \neq d'$, we shall write $d \sqsubset d'$. Next we write

$\mathbf{C}\ D$	:	for the cardinality of D
$\mathbf{H}\ D$	:	for the height (maximal length of chains) of D

where a chain $\{d_0, d_1, \cdots, d_k\}$ has length k if it contains $k+1$ distinct elements.

For the detection of signs analysis considered in Chapter 7, we have that **Sign** has cardinality 8 and height 3, whereas **TT** has cardinality 4 and height 2. For the security analysis considered in Chapter 8, we have that **P** has cardinality 2 and height 1.

For a complete lattice of the form $S \to L$, we have the following fact.

Fact B.1

$\mathbf{C}(S \to L) = (\mathbf{C}\ L)^p$ and $\mathbf{H}(S \to L) = p \cdot (\mathbf{H}\ L)$ for $p \geq 1$ being the cardinality of S.

Thus, for the detection of signs analysis of the factorial program, we have $\mathbf{C}$ **PState** $= 64$ and $\mathbf{H}$ **PState** $= 6$ because the program contains only two variables. Similarly, for the security analysis of the factorial program, we have $\mathbf{C}$ **PState** $= 8$ and $\mathbf{H}$ **PState** $= 3$ because the program contains only two variables and there is the additional token called history.

In the general framework, we have the following proposition.

Proposition B.2

$\mathbf{H}(A \to_t B) \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$.

Proof: Let $h_i : A \to_t B$ and assume that

$$h_0 \sqsubset h_1 \sqsubset \cdots \sqsubset h_k$$

From $h_i \sqsubset h_{i+1}$, we get that there exists $w \in A$ such that $h_i\ w \sqsubset h_{i+1}\ w$ because the ordering on $A \to_t B$ is defined componentwise. There are at most $(\mathbf{C}\ A)$ choices of w, and each w can occur at most $(\mathbf{H}\ B)$ times. Thus

$$k \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$$

as was to be shown. □

Any monotone function is a total function, so Proposition B.2 yields the following corollary.

Corollary B.3

$\mathbf{H}(A \to_m B) \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$.

We shall now apply Proposition B.2 to the special chains obtained when computing fixed points.

Theorem B.4

In the general framewor, any continuous functional

$$H : (A \to_t B) \to (A \to_t B)$$

satisfies $\mathsf{FIX}\ H = H^k \bot$ for

$$k = (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$$

This result carries over to the monotone framework as well. When $A = B = S \to L$, we have $k = (\mathbf{C}\ L)^p \cdot p \cdot (\mathbf{H}\ L)$, where $p \geq 1$ is the cardinality of S.

Proof: Consider the chain

$$H^0 \bot \sqsubseteq H^1 \bot \sqsubseteq \cdots$$

Since $A \to_t B$ is a finite complete lattice, it cannot be the case that all $H^i \bot$ are distinct. Let k' be the minimal index for which $H^{k'} \bot = H^{k'+1} \bot$. Then

$$H^0 \bot \sqsubset H^1 \bot \sqsubset \cdots \sqsubset H^{k'} \bot$$

Using Proposition B.2, we then get that $k' \leq (\mathbf{C}\ A) \cdot (\mathbf{H}\ B)$ (i.e., $k' \leq k$). Since $H^{k'} \bot = \mathsf{FIX}\ H$ and $H^{k'} \bot \sqsubseteq H^k \bot \sqsubseteq \mathsf{FIX}\ H$, we get $\mathsf{FIX}\ H = H^k \bot$. This completes the proof. □

Note that, by finiteness of A and B, the continuity of H simply amounts to monotonicity of H.

Example B.5

The detection of signs analysis of the factorial program gives rise to a continuous functional

$$H : (\mathbf{PState} \to_t \mathbf{PState}) \to (\mathbf{PState} \to_t \mathbf{PState})$$

Now $\mathbf{C}\ \mathbf{PState} = 64$ and $\mathbf{H}\ \mathbf{PState} = 6$ because the factorial program only contains two variables. So, according to the theorem, at most $64 \cdot 6 = 384$ iterations are needed to determine the fixed point. However, the simple calculation of Example 7.7 shows that the fixed point is obtained already after the *second* iteration! Thus our bound is very pessimistic. ☐

Example B.6

The security analysis of the factorial program also gives rise to a continuous functional

$$H : (\mathbf{PState} \to_t \mathbf{PState}) \to (\mathbf{PState} \to_t \mathbf{PState})$$

Since $\mathbf{C}\ \mathbf{PState} = 8$ and $\mathbf{H}\ \mathbf{PState} = 3$, it follows from the theorem that at most $8 \cdot 3 = 24$ iterations are needed to determine the fixed point. However, the simple calculation of Example 8.14 shows that the fixed point is obtained already after the *first* iteration! Thus our bound is once more pessimistic. ☐

B.2 The Completely Additive Framework

The reason why the upper bound determined in Theorem B.4 is so imprecise is that we consider *all* functions in $\mathbf{PState} \to \mathbf{PState}$ and do not exploit any special properties of the functions $H^n \bot$, such as continuity. To obtain a better bound, we shall exploit properties of the analysis functions.

We shall assume that the functions of interest are strict and additive; by *strictness* of a function h we mean that

$$h\ \bot = \bot$$

and by *additivity* that

$$h(d_1 \sqcup d_2) = (h\ d_1) \sqcup (h\ d_2)$$

Since the complete lattices considered are all finite, it follows that a strict and additive function h is also *completely additive*; that is,

$$h(\bigsqcup Y) = \bigsqcup\{\ h\ d \mid d \in Y\}$$

for all subsets Y.

Exercise B.7

Consider the security analysis of Chapter 8 and the following claims: (1) each $\mathcal{SA}[\![a]\!]$ is strict; (2) each $\mathcal{SB}[\![b]\!]$ is strict; (3) each $\mathcal{SS}[\![S]\!]$ is strict; (4) each $\mathcal{SA}[\![a]\!]$ is additive; (5) each $\mathcal{SB}[\![b]\!]$ is additive; (6) each $\mathcal{SS}[\![S]\!]$ is additive. Determine the truth or falsity of each of these claims. □

An element d of a complete lattice $(D, \sqsubseteq)$ is called *join-irreducible* if for all $d_1, d_2 \in D$:

$$d = d_1 \sqcup d_2 \text{ implies } d = d_1 \text{ or } d = d_2$$

As an example **Sign**, has four join-irreducible elements: NONE, NEG, ZERO, and POS. The element NON-POS is not join-irreducible since it is the least upper bound of ZERO and NEG but it is equal to neither of them.

From the definition, it follows that the least element $\perp$ of D is always join-irreducible, but we shall be more interested in the non-trivial join-irreducible elements (i.e., those that are not $\perp$). To this end, we shall write

RJC L : for the number of non-bottom join-irreducible elements of L.

We thus have **RJC Sign** $= 3$, **RJC TT** $= 2$, and **RJC P** $= 1$.

Fact B.8

$\mathbf{RJC}(S \to L) = p \cdot (\mathbf{RJC}\ L)$ where $p \geq 1$ is the cardinality of S.

Proof: The non-trivial join-irreducible elements of $S \to L$ are those functions h that map all but one element of S to $\perp$ and one element to a non-trivial join-irreducible element of L. □

Lemma B.9

If $(L, \sqsubseteq)$ is a finite complete lattice, we have

$$w = \bigsqcup\{\, x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \perp\}$$

for all $w \in L$.

Proof: Assume by way of contradiction that the claim of the lemma is false. Let $W \subseteq L$ be the set of $w \in L$ for which the condition fails. Since W is finite and non-empty, it has a minimal element w. From $w \in W$ it follows that w is not join-irreducible. Hence there exist w_1 and w_2 such that

$$w = w_1 \sqcup w_2,\ w \neq w_1,\ w \neq w_2$$

It follows that $w_1 \sqsubset w$, $w_2 \sqsubset w$, and by choice of w that $w_1 \notin W$ and $w_2 \notin W$. We may then calculate

$$\begin{array}{rcl} w & = & w_1 \sqcup w_2 \\ & = & \bigsqcup\{\, x \mid x \sqsubseteq w_1,\ x \text{ is join-irreducible and } x \neq \bot\} \\ & & \sqcup \bigsqcup\{\, x \mid x \sqsubseteq w_2,\ x \text{ is join-irreducible and } x \neq \bot\} \\ & = & \bigsqcup\{\, x \mid (x \sqsubseteq w_1 \text{ or } x \sqsubseteq w_2),\ x \text{ is join-irreducible and } x \neq \bot\} \\ & \sqsubseteq & \bigsqcup\{\, x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \bot\} \\ & \sqsubseteq & w \end{array}$$

This shows that

$$w = \bigsqcup\{\, x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \bot\}$$

and contradicts $w \in W$. Hence $W = \emptyset$ and the claim of the lemma holds. □

In the completely additive framework, we have the following proposition.

Proposition B.10

$\mathbf{H}(A \rightarrow_{sa} B) \leq (\mathbf{RJC}\ A) \cdot (\mathbf{H}\ B)$.

Proof: The proof is a refinement of that of Proposition B.2. So we begin by assuming that $h_i \in A \rightarrow_{sa} B$ and that

$$h_0 \sqsubset h_1 \sqsubset \cdots \sqsubset h_k$$

As in the proof of Proposition B.2, we get an element w such that $h_i\ w \sqsubset h_{i+1}\ w$ for each $h_i \sqsubset h_{i+1}$. The element w is an arbitrary element of A, so in the proof of Proposition B.2 there were $(\mathbf{C}\ A)$ choices for w. We shall now show that w can be chosen as a non-trivial join-irreducible element of A, thereby reducing the number of choices to $(\mathbf{RJC}\ A)$. Calculations similar to those in the proof of Proposition B.2 will then give the required upper bound on k; i.e., $k \leq (\mathbf{RJC}\ A) \cdot (\mathbf{H}\ B)$.

The element w satisfies $h_i\ w \sqsubset h_{i+1}\ w$. By Lemma B.9, we have

$$w = \bigsqcup\{\ x \mid x \sqsubseteq w,\ x \text{ is a join-irreducible and } x \neq \bot\}$$

From the strictness and additivity of h_i and h_{i+1}, we get

$$\begin{array}{rcl} h_i\ w & = & \bigsqcup\{\, h_i\ x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \bot\} \\ h_{i+1}\ w & = & \bigsqcup\{\, h_{i+1}\ x \mid x \sqsubseteq w,\ x \text{ is join-irreducible and } x \neq \bot\} \end{array}$$

It cannot be the case that $h_i\ x = h_{i+1}\ x$ for all non-bottom join-irreducible elements x of A since then $h_i\ w = h_{i+1}\ w$. So let x be a non-bottom join-irreducible element where $h_i\ x \sqsubset h_{i+1}\ x$. Then there will only be $(\mathbf{RJC}\ A)$ choices for x and this completes the proof. □

We can now apply Proposition B.10 to the special chains obtained when computing fixed points.

Theorem B.11

In the completely additive framework, any continuous functional

$$H : (A \to_{sa} B) \to (A \to_{sa} B)$$

satisfies $\mathsf{FIX}\ H = H^k \bot$ for

$$k = (\mathbf{RJC}\ A) \cdot (\mathbf{H}\ B)$$

When $A = B = S \to L$ we have $k = p \cdot (\mathbf{RJC}\ L) \cdot p \cdot (\mathbf{H}\ L)$, where $p \geq 1$ is the cardinality of S.

Proof: Analogous to the proof of Theorem B.4. □

The equality test between the iterands $H^0\bot, H^1\bot, \cdots$ can be simplified in this framework. To see this, consider two functions $h_1, h_2 \in A \to_{sa} B$. Then

$$h_1 = h_2$$

if and only if

$$h_1\ x = h_2\ x \text{ for all non-trivial join-irreducible elements } x \text{ of } A.$$

Example B.12

Continuing Example B.6 (and using Exercise B.7), we have $\mathbf{RJC\ PState} = 3$ and $\mathbf{H\ PState} = 3$. So according to the theorem, at most $3 \cdot 3 = 9$ iterations are needed. □

B.3 Iterative Program Schemes

The upper bounds expressed by Theorems B.4 and B.11 are obtained without any assumptions about the functional H except that it is a continuous function over the relevant lattices. In this section, we shall restrict the form of H.

For iterative programs such as a `while`-loop, the functional H will typically have the form

$$H\ h = f \sqcup (h \circ g)$$

Since our aim is to further improve the bound of the previous section, we shall assume that f and g are strict and additive functions. Then also the iterands $H^i\bot$ will be strict and additive.

Exercise B.13

The functionals obtained by analysing the `while`-loop in the security analysis of Chapter 8 can be written in this form provided that a minor modification is made in the definition of cond_S. To see this, define the auxiliary functions f-func and g-func by

$$\mathsf{f\text{-}func}(f)ps \ = \begin{cases} \text{LOST} & \text{if } f\ ps = \text{HIGH} \\ ps & \text{if } f\ ps = \text{LOW} \end{cases}$$

and

$$\mathsf{g\text{-}func}(f,g)ps \ = \begin{cases} \text{LOST} & \text{if } f\ ps = \text{HIGH} \\ g\ ps & \text{if } f\ ps = \text{LOW} \end{cases}$$

Show that the functional H obtained from `while` b `do` S can be written as

$$H\ h = \ \mathsf{f\text{-}func}(\mathcal{SB}[\![b]\!]) \ \sqcup\ \mathsf{g\text{-}func}(\mathcal{SB}[\![b]\!], h \circ \mathcal{SS}[\![S]\!])$$

and that a minor modification in the definition of cond_S suffices for obtaining the desired

$$H\ h = \ \mathsf{f\text{-}func}(\mathcal{SB}[\![b]\!]) \ \sqcup\ h\ \circ\ \mathsf{g\text{-}func}(\mathcal{SB}[\![b]\!], \mathcal{SS}[\![S]\!])$$

Discuss the implications of making these variations. □

The first part of the next theorem is a refinement of Theorem B.11.

Theorem B.14

Let $f \in A \to_{sa} B$ and $g \in A \to_{sa} A$ be given and define

$$H\ h = f \sqcup (h \circ g)$$

Then $H : (A \to_{sa} B) \to (A \to_{sa} B)$ is a continuous functional and taking

$$k = (\mathbf{H}\ A)$$

will ensure

$$\mathsf{FIX}\ H = H^k \bot$$

When $A = B = S \to L$, we have $k = p \cdot (\mathbf{H}\ L)$, where $p \geq 1$ is the cardinality of S.

Writing $H_0\ h = \mathtt{id} \sqcup (h \circ g)$ and taking k_0 to satisfy

$$H_0^{k_0} \bot\ w = H_0^{k_0+1} \bot\ w$$

we have

$$\mathsf{FIX}\ H\ w = H^{k_0} \bot\ w = f(H_0^{k_0} \bot\ w)$$

In other words, fixed points of H_0 may be computed pointwise.

Basically this result says that in order to compute $\mathsf{FIX}\ H$ on a particular value w, it is sufficient to determine the values of the iterands $H_0^i \bot$ at w and then compare these values. So rather than having to test the extensional equality of two functions on a set of arguments, we only need to test the equality of two function values. Furthermore, the theorem states that this test has to be performed at most a linear number of times.

To prove the theorem, we need the following two lemmas.

Lemma B.15

Let $H\ h = f \sqcup (h \circ g)$ for $f \in A \to_{sa} B$ and $g \in A \to_{sa} A$. Then for $i \geq 0$ we have

$$H^{i+1} \bot = \bigsqcup \{\, f \circ g^j \mid 0 \leq j \leq i \}.$$

Proof: We proceed by induction on i. If $i = 0$ then the result is immediate, as $H^1 \bot = f \sqcup (\bot \circ g) = f = \bigsqcup \{\, f \circ g^j \mid 0 \leq j \leq 0 \}$. For the induction step, we calculate

$$\begin{array}{rcl} H^{i+2} \bot & = & f \sqcup (H^{i+1} \bot) \circ g \\ & = & f \sqcup (\bigsqcup \{\, f \circ g^j \mid 0 \leq j \leq i \}) \circ g \\ & = & \bigsqcup \{\, f \circ g^j \mid 0 \leq j \leq i+1 \} \end{array}$$

where the last equality uses the pointwise definition of $\bigsqcup$ on $A \to_{sa} B$. $\square$

Lemma B.16

Let $H\ h = f \sqcup (h \circ g)$ for $f \in A \to_{sa} B$ and $g \in A \to_{sa} A$. Then

$$\mathsf{FIX}\ H = f \circ (\mathsf{FIX}\ H_0) \text{ and } H^k \bot = f \circ H_0^k \bot$$

where $H_0\ h = \mathtt{id} \sqcup (h \circ g)$.

Proof: We shall first prove that

$$H^i \bot = f \circ H_0^i \bot \text{ for } i \geq 0.$$

The case $i = 0$ is immediate because f is strict. So assume that $i > 0$. We shall then apply Lemma B.15 to H and H_0 and get

$$\begin{aligned} H^i \bot &= \bigsqcup\{\, f \circ g^j \mid 0 \leq j \leq i-1\} \\ H_0^i \bot &= \bigsqcup\{\, g^j \mid 0 \leq j \leq i-1\} \end{aligned}$$

Since f is additive, we get

$$H^i \bot = f \circ \bigsqcup\{\, g^j \mid 0 \leq j \leq i-1\} = f \circ H_0^i \bot$$

as required.

We now have

$$\begin{aligned} \mathsf{FIX}\ H &= \bigsqcup\{\, H^i \bot \mid i \geq 0\} \\ &= \bigsqcup\{\, f \circ H_0^i \bot \mid i \geq 0\} \\ &= f \circ \bigsqcup\{\, H_0^i \bot \mid i \geq 0\} \\ &= f \circ (\mathsf{FIX}\ H_0) \end{aligned}$$

where the third equality uses the complete additivity of f. □

We now turn to the proof of Theorem B.14.

Proof: We shall first prove that if $H_0^{k_0} \bot\ w = H_0^{k_0+1} \bot\ w$, then $\mathsf{FIX}\ H\ w = f(H_0^{k_0} \bot\ w)$. This is done in two stages.

First assume that $k_0 = 0$. Then $H_0^0 \bot\ w = H_0^1 \bot\ w$ amounts to $\bot = w$. Using Lemma B.15 and the strictness of g, we get

$$H_0^{i+1} \bot\ \bot = \bigsqcup\{\, g^j \bot \mid 0 \leq j \leq i\} = \bot$$

for $i \geq 0$ and thereby $\mathsf{FIX}\ H_0\ \bot = \bot$. But then Lemma B.16 gives

$$\mathsf{FIX}\ H\ \bot = f(\mathsf{FIX}\ H_0\ \bot) = f \bot = f(H_0^0 \bot\ \bot)$$

as required.

Second, assume that $k_0 > 0$. From $H_0^{k_0} \bot\ w = H_0^{k_0+1} \bot\ w$, we get, using Lemma B.15, that

$$\bigsqcup\{\, g^j\ w \mid 0 \leq j < k_0\} = \bigsqcup\{\, g^j\ w \mid 0 \leq j \leq k_0\}$$

This means that

$$g^{k_0}\ w \sqsubseteq \bigsqcup\{\, g^j\ w \mid 0 \leq j < k_0\}$$

We shall now prove that for all $l \geq 0$

$$g^{k_0+l}\ w \sqsubseteq \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\} \qquad (*)$$

We have already established the basis $l = 0$. For the induction step, we get

$$\begin{array}{rcl} g^{k_0+l+1}\ w & = & g(g^{k_0+l}\ w) \\ & \sqsubseteq & g(\bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\}) \\ & = & \bigsqcup\{\ g^j\ w \mid 1 \leq j \leq k_0\} \\ & \sqsubseteq & \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\} \end{array}$$

where we have used the additivity of g. This proves $(*)$. Using Lemma B.15 and $(*)$, we get

$$\begin{array}{rcl} H_0^{k_0+l} \bot\ w & = & \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0 + l\} \\ & = & \bigsqcup\{\ g^j\ w \mid 0 \leq j < k_0\} \\ & = & H_0^{k_0} \bot\ w \end{array}$$

for all $l \geq 0$. This means that $\mathsf{FIX}\ H_0\ w = H_0^{k_0} \bot\ w$, and using Lemma B.16 we get

$$\mathsf{FIX}\ H\ w = f(H_0^{k_0} \bot\ w)$$

as required.

To complete the proof of the theorem, we have to show that one may take $k = (\mathbf{H}\ A)$. For this it suffices to show that one cannot have a chain

$$H_0^0 \bot\ w \sqsubset H_0^1 \bot\ w \sqsubset \cdots \sqsubset H_0^k \bot\ w \sqsubset H_0^{k+1} \bot\ w$$

in A. But this is immediate since $k + 1 > \mathbf{H}(A)$. □

Example B.17

Continuing Examples B.6, B.12 (and using Exercise B.13), we have $\mathbf{H}\ \mathbf{PState} = 3$. So according to the theorem, at most 3 iterations are needed. In general, for a statement S with p variables, we have $\mathbf{H}\ \mathbf{PState} = p + 1$ and hence in the security analysis at most $p + 1$ iterations are needed. □

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison–Wesley, Reading MA (1982).

[2] K. R. Apt, Ten Years of Hoare's Logic: A Survey — Part 1, *ACM Toplas*, **3** 431–483 (1981).

[3] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn, A Simple Applicative Language: Mini-ML, in *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* 13–27, ACM Press, New York (1986).

[4] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge (1990).

[5] J. Despeyroux, Proof of Translation in Natural Semantics, in *Proceedings of Symposium on Logic in Computer Science*, 193–205, IEEE Press, Cambridge, MA (1986).

[6] J. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs*, MIT Press, Cambridge, MA (1996).

[7] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, MIT Press, Cambridge, MA (1992).

[8] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*, Wiley, New York (1991).

[9] C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, NJ (1980).

[10] N. D. Jones and F. Nielson, Abstract Interpretation: a Semantics Based Tool for Program Analysis, in *Handbook of Logic in Computer Science* **4**,

S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (editors), 527–636, Oxford University Press, Oxford (1995).

[11] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, Englewood Cliffs, NJ (1993).

[12] J. Loeckx and K. Sieber, *The Foundations of Program Verification*, Wiley–Teubner Series in Computer Science, Wiley, New York (1984).

[13] H. Riis Nielson, *Hoare Logic's for Run-time Analysis of Programs*, PhD-thesis, Edinburgh University (1984).

[14] H. Riis Nielson, A Hoare-like Proof System for Run-time Analysis of Programs, *Science of Computer Programming*, **9** 107–136 (1987).

[15] F. Nielson, Program Transformations in a Denotational Setting, *ACM TOPLAS*, **7** 359–379 (1985).

[16] F. Nielson and H. Riis Nielson, Two-level Semantics and Code Generation, *Theoretical Computer Science*, **56** 59–133 (1988).

[17] F. Nielson, Two-Level Semantics and Abstract Interpretation, *Theoretical Computer Science – Fundamental Studies*, **69** 117–242 (1989).

[18] H. Riis Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*, Wiley, New York (1992).

[19] F. Nielson and H. Riis Nielson, *Two-Level Functional Languages*, Cambridge University Press, Cambridge (1992).

[20] F. Nielson, H. Riis Nielson, and C. Hankin, *Principles of Program Analysis*, Springer, New York (2005).

[21] G. D. Plotkin, *A Structural Approach to Operational Semantics*, lecture notes, DAIMI FN-19, Aarhus University, Denmark (1981, reprinted 1991).

[22] G. D. Plotkin, An Operational Semantics for CSP, in *Formal Description of Programming Concepts II*, Proceedings of TC-2 Working Conference, D. Bjørner (editor), North–Holland, Amsterdam (1982).

[23] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Journal of Logic and Algebraic Programming, **60–61** 17–139 (2004).

[24] D. A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn & Bacon, Boston (1986).

[25] J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge MA (1977).

Index