HIGHER-LEVEL ALGORITHMIC STRUCTURES IN THE REFINEMENT CALCULUS

by

Steve King

Technical Monograpb PRG-124
ISBN 0-902928-98-8

January 1999

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

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

Higher-level algorithmic structures in the refinement calculus

Stephen King
St Edmund Hall

Trinity Term 1998

Being a Thesis submitted for the degree of D.Phil. in the University of Oxford.

Higher-level algorithmic structures in the refinement calculus

D.Phil. Thesis

Stephen King
St Edmund Hall

Trinity Term 1998

Abstract

This thesis extends the refinement calculus into two new areas: exceptions and iterators. By extending the calculus in this way, it is shown that we can carry out the formal development of programs which exploit the exception-handling and iterator mechanisms of programming languages. For both areas of expansion, the same strategy is used: relatively simple extensions to the language are first proposed, together with the semantic machinery to prove the correctness of new laws. Then these simple extensions are combined into complex mechanisms which mimic more closely the language facilities found in programming languages, which are necessary for programs of realistic size.

For exceptions, the major idea is to distinguish between normal and exceptional termination of program constructs. Dijkstra's weakest precondition semantics are extended to give meaning to this by considering predicate transformers which take as arguments more than one postcondition. The notation is extended to deal with multiple exceptions, and appropriate actions for them, by the use of procedures.

The technical background for the iterator construct proposed comes from the functional programming community: homomorphisms from initial data types. Again, it is shown how this can be related to iterators in programming languages. This involves giving weakest precondition semantics for procedures as parameters.

Both extensions to the refinement calculus are used to give formal developments of programs which use a pre-existing library of abstract data types. A specification is given for a typical library component, and several sample programs are developed.

Contents

Et quand finira mon livre? Problème. Pour qu'il paraisse l'hiver prochaine, je n'ai pas d'ici là une minute à perdre. Mais, par moments, il me semble qne je me liquéfie comme un vieux camembert, tant je me sens fatigué! *Gustave Flaubert*

Acknowledgements

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

In vaguely chronological order, financial support (in terms of a salary!) has come from Oxford University Computing Laboratory, IBM United Kingdom Laboratories Ltd, Oxford University Department for Continuing Education and The University of York, and I am most grateful to these organisations.

Thanks, too, to those who have read and commented-on drafts of parts of the material presented here: Richard Bird, Ana Cavalcanti, Ian Hayes, Tony Hoare, Dave Naumann, Jane Sinclair, Jan van de Snepscheut, Jim Woodcock.

Thanks to Andrew Martin and Mike Spivey for assistance with LaTeX styles.

For the last 3 years, I have been working at the University of York, and I am very grateful to the Computer Science Department, particularly the Head of Department, Dr Keith Mander, and Professors Ian Wand and John McDermid: I do appreciate their support, encouragement and forbearance!

I have benefitted from the love of my family and the friendship of many people at St Edmund Hall, Oxford University Computing Lab and The University of York: they have provided snpport, encouragement and abuse at various appropriate times.

I owe a major debt of gratitude to Carroll Morgan, my supervisor, who has managed to maintain interest (I think) over a considerable period of time. He has consistently provided ideas, suggestions and feedback on drafts, and I am very grateful to him: I'm not sure that this thesis would ever have reached this state without him!

But my major debt is to my wife Clare, who has put up with 'the dreaded T.....' for as long as she has known me, who has seen several deadlines come and go, and who will be more than happy to see the back of it. I am absolutely certain that I would not have reached this state without her, and I am eternally grateful: let real life now commence!

Steve King
September 1998

> God keep me from ever finishing anything. This whole book is but a draft — nay, but a draft of a draft. Oh Time, Strength, Cash, and Patience! *Herman Melville*

Part I

Prologue

Chapter 1

Introduction

Ever since the 1968 NATO conference which introduced the phrase 'the software crisis' to the world, software developers (or at least their managers) have been searching for the Holy Grail of programming — a software development technique which will allow them to develop ever larger and more complex software products, with the certainty that the programs delivered will perform as expected, always giving 'correct results' — whatever that may mean. They also want to be sure that these products will be delivered on time and within budget, and that they will be documented and structured so as to make it simple to carry out any future enhancements — of course, there will be no bugs, so error removal will not form part of maintenance. It is now 30 years since that conference, and, despite the claims of some marketing managers and over-enthusiastic salesmen, the search for the ultimate software development method is still going on. This thesis attempts to fill a small hole by extending an existing development technique and notation to cover two areas that were previously beyond its reach. No claim is made that the extended method is even close to this ultimate development method, but it is certainly a step in the right direction!

1.1 Software development and the refinement calculus

The notation which is to be extended is that of the refinement calculus,[1] which is based firmly on the idea that programming is a mathematical activity. When

[1] In particular, we use Morgan's version of the calculus [44].

Dijkstra introduced his language of guarded commands [17], he included constructs that could be given simple and elegant mathematical meanings. Constructs such as the unrestricted 'goto' of early programming languages were rejected because they did not have a simple mathematical meaning: symptoms of their complex semantics were that there were no simple laws that they obeyed, and that programmers were much more likely to misunderstand their behaviour and make mistakes while using them. The claim of early members of the 'formal methods' school was that programmers who used a restricted language with well-understood semantics were much less error-prone.

One of the areas of research interest which subsequently opened up was the use of mathematical notations for the specification of software: by introducing precision into the software development process at an early stage, to describe what the proposed system was intended to achieve, researchers (and users) hoped to uncover misunderstandings and areas of uncertainty as early as possible, thus enabling corrections to be made before too much further work had been carried out. Because the 'programming language' — Dijkstra's language of guarded commands, or something similar — and the specification language both had a firm mathematical basis, it was possible, at least in theory, to *prove* that a program met its specification. Much research effort was spent, and is still being spent, on ways of proving such developments correct, using techniques such as introducing intermediate design stages to reduce the size of the 'gap' to be proved.

By the early 1980s, it had become clear that one of the problems was the difference in notations used at the specification, design and coding stages: at each stage a favourite notation was used, making it difficult to prove relationships between the stages. This caused a growing interest in 'wide-spectrum languages', which were languages that could be used throughout the development process, with features designed particularly for certain stages, but the whole language described in a coherent mathematical framework. This reduced the difficulty involved in translating from a specification notation described in one sort of mathematics to a design notation described slightly differently, to a programming notation described in yet a third way. The various refinement calculi[2] are such wide-spectrum languages, which developed by extending the programming language — Dijkstra's guarded commands — with non-executable constructs suitable for writing specifications. A development begins with a specification, usually expressed in terms of these non-executable specification constructs. The specification is then 'transformed' until it contains entirely executable constructs. These transformations are justified by the laws of the calculus, which guarantee to 'preserve meaning' in a very precise way. Details of Morgan's version of the refinement calculus notation and its laws are given in Chapter 2.

[2]For historical reasons, we often refer to '*the* refinement calculus', although there are several notations — see Section 2.2 for some details of the history of these calculi.

1.2 Extending the refinement calculus

This then is the context in which we are working: we have chosen to extend the refinement calculus to cover two new features of programming languages: exceptions and iterators. The reason for the choice of these two additional features lies in the author's experience while working on a research project funded by IBM Hursley. By the early 1990s, there had been some success at Hursley in the use of formal methods, at least in the area of recording specifications and designs precisely, if not in the use of proof, and there was some interest in the use of other techniques to improve programmer productivity. One of the techniques investigated involved the use of the Böblingen Building Blocks: these were a collection, produced by IBM's laboratory in Böblingen, Germany, of implementations of commonly-used abstract data types, such as sets, maps trees etc. The Blocks were implemented using the macro-expansion facilities of an IBM-internal high-level systems programming language, PL/AS. The investigation looked at ways in which the use of the Blocks could be incorporated into a development method which was then based on Z. The intention was that specifications would be written in Z, using Z's notions of sets, relations, functions etc, which could then be implemented directly using the relevant Building Blocks. Clearly, the first step was to write formal specifications for the Blocks, but here certain problems arose. Three serious difficulties were discovered while writing the specifications:

- the Blocks included an iterator mechanism, which allowed programmers to apply an operation to each element of a data structure;
- when an operation failed, an exception-handling mechanism came into effect; and
- many of the operations dealt with pointers, either as inpnts or outputs, rather than dealing with the data structures themselves.

The last of these was recognised as just one aspect of a major problem that has been facing the formal methods community for some time — how to deal convincingly with pointers. It was decided not to tackle this, but to work on the first two problems, which provided new and interesting challenges.

At around this time (1993), a new version of the Building Blocks appeared, re-written to use the C++ language, rather than PL/AS, and available as a commercial product, rather than being restricted to the IBM community. It was also no longer dependent on the rather crude macro-expansion facility of PL/AS, and seemed to be a much more stable and robust product. It was therefore decided to base the research around this product, which had now been renamed the IBM Collection Class Library.

1.3 Thesis structure

In order to give meaning to the exception-handling and iterator mechanisms in the Collection Class Library, our strategy is to describe first some simpler constructs which we can add to the refinement calculus and give meaning to, and whose laws we can explore, before showing how these simple constructs can be used for the more complicated mechanisms of the existing library. Prior to that, in Chapter 2, we give a brief summary of the standard refinement calculus notation, and its meaning.

Our work on exception handling starts in Chapter 3, where, after examining the need for exceptions, we make the important distinction between two forms of termination for constructs of the language, normal and exceptional termination. We explore some laws that the extended language constructs obey and show how to develop programs that make use of these constructs. We also extend the usual semantic framework (Dijkstra's weakest precondition) to cover the two forms of termination, and we are therefore able to justify the laws we have proposed.

However, a realistic treatment of exceptions needs to do more than distinguish between normal and exceptional termination: different actions may be appropriate for different errors during a particular invocation of a command, and it may be the case that different actions are appropriate for a single exception during different invocations. In Chapter 4, we deal with this, basing an exception-handling mechanism on the use of procedures.

Part III of the thesis contains our work on iterators. Once again, we start with a fairly simple addition to the refinement calculus notation (in Chapter 5), which is based on work from the functional programming community on catamorphisms — homomorphisms from initial algebras.

In order to encapsulate iterators, as we must if we are to use them as part of a library of abstract data types, we need to pass procedures as parameters. This is not covered in the usual treatments of the refinement calculus, but recent work by Naumann [49] on weakest precondition semantics for procedure variables has laid the foundations on which we base a treatment of procedural parameters in Chapter 6. This allows us to discuss the encapsulation of iterators in the following chapter.

Having set up all this machinery for dealing with exceptions and iterators, we can now return to the Collection Class Library, and work on some applications of our work. The two chapters in Part IV of the thesis deal with applications involving exceptions and iterators, respectively. Having given a specification of one of the Collection Classes, we show how programs are developed which use that specification.

Finally, in Chapter 10, we set our work in the context of other related work, look at possible future work, and draw some conclusions.

Since a large number of laws are introduced at many different points in the thesis, we have collected together all the laws in Appendix B, for ease of reference.

1.4 Contribution

The overall contribution of this thesis is to show how an existing software development notation — the refinement calculus — can be extended in two directions, to cover two additional features commonly found in programming languages: exceptions and iterators. The techniques used the distinction between normal and exceptional termiuation and the use of catamorphisms, together with the semantic extensions used to give meaning to these constructs — are perhaps more important than the application of the extended language to the development of programs using a particular library of abstract data types.

Some of the material in this thesis on exceptions has previously heen published in the Formal Aspects of Computing Journal [25].

1.5 Notation

In Part II of the thesis — on exceptions — and the summary of the refinement calculus, the following notational conventions are used:

- names consisting of a single letter repeated three times, aaa, bbb etc., represent programs; and
- single Greek letters α, β etc., represent predicates, that is, sets of states. (We are not considering the question of expressibility, and thus we sometimes blur the distinction between a Boolean function on a state space and the corresponding set of states.)

Un-numbered laws eg "*following assignment*" refer to laws in the standard text on the refinement calculus [44].

Chapter 2

The refinement calculus

In this chapter, we give a very brief introduction to the refinement calculus. We describe first the basic constructs of the language and their meanings in terms of Dijkstra's weakest precondition. Then we give a brief history of the development of the calculus and look at some more advanced features. Finally, we give a short sample development to show the notation in use.

2.1 Basics

The refinement calculus arose out of a simple extension of Dijkstra's language of guarded commands [17]. A *specification*, here written

$$w : [\alpha, \beta]$$

comprises a frame w, and two predicates: the precondition α and the postcondition β. It is a command in the programming language which, like the others, describes the intended effect of a computation. Unlike conventional programming commands however, it does not necessarily suggest a mechanism for the computation: it gives the *what*, but not the *how*. In the refinement calculus world, we do not distinguish specifications from programs: every specification is also a program (but not *vice versa*).

In the specification $w : [\alpha, \beta]$, the frame w is a (possibly empty) list of variables that the specification (command) may alter. When the precondition α is true initially, the specification is guaranteed to terminate in a state satisfying the postcondition β. On the other hand, when α is not true initially, no guarantees

can be made about the behaviour of the specification: it might terminate in an arbitrary state or it might not terminate at all.

For example:

- $y : [true, y^2 = x]$ is a specification which states that a value should be assigned to y to make the predicate $y^2 = x$ true (thus assigning to y a square root of x);
- $e : [s \neq \{\}, e \in s]$ is a command which chooses an element e from a set s, provided s is non-empty; if s is empty initially, then this command might not terminate, or it might assign an arbitrary value to e; and
- $x : [b^2 \geq 4ac \ , \ ax^2 + bx + c = 0]$ is a command which solves the quadratic equation for x, provided the discriminant is non-negative; if the discriminant is less than 0, then its behaviour is arbitrary.

The meaning of a specification statement can be given in terms of Dijkstra's weakest precondition semantics [18]:

$$wp(w : [\alpha, \beta], \phi) \mathrel{\widehat{=}} \alpha \wedge (\forall w \bullet \beta \Rightarrow \phi) \quad .$$

which means that, for example,

$$wp(e : [s \neq \{\}, e \in s], \phi) \ = \ s \neq \{\} \wedge (\forall r \bullet e \in s \Rightarrow \phi)$$

Apart from the specification statement, the second essential ingredient of the refinement calculus is a relation, called refinement, between programs. We write

$$aaa \ \sqsubseteq \ bbb \quad ,$$

for two programs aaa and bbb, to say that aaa is refined by bbb: and that, in turn, means informally that any client who has asked for the program aaa will be happy if given bbb instead. Formally, the definition of the refinement relation between programs is given by weakest preconditions:

$$aaa \sqsubseteq bbb \mathrel{\widehat{=}} wp(aaa, \phi) \Rightarrow wp(bbb, \phi) \text{ for all postconditions } \phi \quad .$$

For example, the first specification mentioned above, $y : [true, y^2 = x]$, could be refined by the program $y := \sqrt{x}$. On the other hand, it could also be refined by the program $y := -\sqrt{x}$: any client who had agreed that their needs were met by the original specification would have no grounds for complaint, whichever program they were given.

Program development in the refinement calculus is usually carried out via a series of so-called refinement steps, starting from a specification aaa, say, and ending with an executable program zzz. In between might occur a number of 'hybrid' programs, containing both specifications and executable fragments:

$$aaa \sqsubseteq \ \ldots \sqsubseteq lll \sqsubseteq mmm \sqsubseteq nnn \sqsubseteq ooo \sqsubseteq \ldots \sqsubseteq zzz \quad .$$

	Syntax	Semantics
assignment	$x := E$	$wp(x := E, \phi) = \phi[x \backslash E]$
sequential composition	$aaa;\ bbb$	$wp(aaa;\ bbb, \phi) = wp(aaa, wp(bbb, \phi))$
alternation	$\mathbf{if}([]i \bullet \alpha_i \rightarrow aaa_i)\ \mathbf{fi}$[1]	$wp(\mathbf{if}([]i \bullet \alpha_i \rightarrow aaa_i)\mathbf{fi}, \phi) =$ $(\bigvee i \bullet \alpha_i) \wedge$ $(\bigwedge i \bullet \alpha_i \Rightarrow wp(aaa_i, \phi))$[2]
recursion	$\mathbf{mu}\ aaa \bullet \mathcal{P}(aaa)\ \mathbf{um}$	given by least fixed point: see Section 3.4
specification	$w : [\alpha, \beta]$	$wp(w : [\alpha, \beta], \phi) =$ $\alpha \wedge (\forall w \bullet \beta \Rightarrow \phi)$
skip	$\mathbf{skip}$	$wp(\mathbf{skip}, \phi) = \phi$
abort	$\mathbf{abort}$	$wp(\mathbf{abort}, \phi) = false$
nondeterministic choice	$aaa \ [] \ bbb$	$wp(aaa \ [] \ bbb, \phi) =$ $wp(aaa, \phi) \wedge wp(bbb, \phi)$
naked guarded command	$\alpha \rightarrow aaa$	$wp(\alpha \rightarrow aaa, \phi) =$ $\alpha \Rightarrow wp(aaa, \phi)$

Figure 2.1: The major constructs of the refinement calculus

The overall desired result $aaa \sqsubseteq zzz$ follows from the transitivity of $\sqsubseteq$. The justification for each refinement step is given by appealing to one (or more) of a collection of laws, from which the 'calculus' takes its name.

The major constructs of the language are summarised in Figure 2.1, together with their usual meanings in terms of standard weakest preconditions. Later however, we will need to extend the notion of weakest preconditions to cover exceptions.

2.2 History

Historically, Back was the first to embed specifications in programs, using the weakest precondition calculus [3, 4], although his specifications contained only a single predicate. More recently, both Morris [46] and Morgan [42] have extended Back's work by using separate pre- and postconditions. All three authors have the same refinement relation. The refinement calculus continues the tradition of Hoare [21] and Dijkstra [18]; for example, the meanings of the specification statement and the refinement relation were deliberately chosen to make true the following theorem (Theorem 3 of [42]):

[1] This is a shorthand for

$$\mathbf{if}\ \alpha_1 \rightarrow aaa_1\ [] \ldots [] \ \alpha_n \rightarrow aaa_n\ \mathbf{fi}\ .$$

The number of branches must be finite, but may be zero.

[2] $\bigvee$ and $\bigwedge$ denote distributed disjunction and conjunction, respectively.

Taking w to be all program variables, and aaa to be an executable program,

$$w : [pre, post] \sqsubseteq aaa$$

has exactly the same meaning as

$$(\forall w \bullet pre \Rightarrow wp(aaa, post)) \ .$$

This theorem allows us to check the validity of the laws of the refinement calculus, such as this law for decomposing a specification into the sequential composition of two specifications:

$$w : [pre, post] \ \sqsubseteq \ w : [pre, mid] \ ; \ w : [mid, post] \ .$$

There is an extensive collection of laws such as the above, some with side-conditions to be proved, which are used to justify the refinement steps in a program development. A tutorial introduction to these laws may be found in [44], while a collection of more theoretical papers may be found in [45].

2.3 Further features

We look more closely now at some specific features of the refinement calculus. The first is 'naked' guarded commands, which were first described by Nelson in [50]. Morgan and Morris discovered them independently, as a natural consequence of the semantic definitions of the refinement calculus. These are commands of the form '$\alpha \rightarrow aaa$' which do not necessarily occur within **if** ... **fi** or **do** ... **od**. For any predicate α (the guard) and program aaa we define

$$wp(\alpha \rightarrow aaa, \phi) \ \widehat{=} \ \alpha \Rightarrow wp(aaa, \phi) \ .$$

Though such commands are well-behaved, and even to some extent generally accepted, they do not satisfy the 'Law of the Excluded Miracle' [18]: in particular,

$$wp(false \rightarrow \mathbf{skip}, \phi) = true$$

for any postcondition ϕ whatever. For that reason we give the guarded command $false \rightarrow \mathbf{skip}$ the name **magic**, and note that $aaa \sqsubseteq \mathbf{magic}$ for any aaa. It is also easily checked with wp that **magic** is left-absorptive: $\mathbf{magic}; aaa = \mathbf{magic}$ for any program aaa.

The second feature is pure nondeterministic choice. We have

$$wp(aaa \ [] \ bbb, \phi) \ \widehat{=} \ wp(aaa, \phi) \wedge wp(bbb, \phi) \ .$$

The two constructs interact nicely: one example is that $aaa \ [] \ \mathbf{magic} = aaa = \mathbf{magic} \ [] \ aaa$ holds for all programs aaa. Another example is provided by the

first step of the derivation in Section 3.2,[3] which is easily verified using *wp*. In this step, we remove the **if..fi** around an alternation, which is justified so long as the guards are exhaustive, leaving naked guarded commands.

A *logical constant* plays the role of a 'ghost'-variable. It can he used for example to refer in *post*-conditions to values defined *before* a statement. In what follows, we will use the logical constant V to refer to the initial value of the variant expression. Logical constants are introduced by **con**, whose meaning is given as follows:

$$wp(|[\ \mathbf{con}\ x \bullet aaa\ ]|, \phi) \mathrel{\widehat{=}} (\exists x \bullet wp(aaa, \phi))$$
provided that ϕ contains no free x .

An *assumption* is written $\{\alpha\}$, for some predicate α, and in a sense conveys the claim that "α is true here". As a statement it acts as **skip** when α is true, **abort** otherwise. This means that it is different from an 'assert statement', as found in Algol-W for instance, which is guaranteed to terminate[4] when the formula is false. Unlike assert statements, assumptions are therefore useless for program instrumentation, but are intended for use during the development of programs, and are removed before the final code is collected. The meaning of an assumption is given by

$$wp(\{\alpha\}, \phi) \mathrel{\widehat{=}} \alpha \wedge \phi \ .$$

It should be noted that we usually omit the semicolon between an assumption and any following statement:

$$\{\alpha\}; aaa = \{\alpha\}\ aaa \quad .$$

Layout of developments

A particular technique of layout is often found in refinement calculus developments: certain lines of the development are labelled with numbers, and these labels are used to coutinue the derivation at a later stage. We also sometimes label lines with a $\lhd$ symbol: this signifies that this line will be worked on in the very next step of the development. There will therefore be at most one line marked with $\lhd$ at any stage, although it is possible that several lines may be labelled with numbers. The complete program can eventually be found by collecting the code fragments from the branches of the resulting development tree. These notations are used in the example below.

2.4 An example development

We now give a brief example of a program development using the refinement calculus features mentioned above. We will spell out the steps in some detail:

[3] after Law 3.13 on page 24

[4] and to cause immediate program termination!

in a 'real-life' development, such detail would be neither required nor desirable.

The simple program fragment which we will develop is to set the value of a boolean variable depending on whether an array contains some particular value. Our variable declarations are as follows:

$$as : 1..n \rightarrow X$$
$$x : X$$
$$b : Boolean$$

The array is modelled as a total function from the indices $1..n$ to the values which are drawn from the set X. We are required to set the value of b, depending on whether x appears in the array. Our specification is therefore

$$b : [b \Leftrightarrow \exists i : 1..n \bullet as\, i = x] \ .$$

(Notice that the precondition, which is *true*, has been omitted — this is a common abbreviation in specification statements.)

The development starts by introducing a local variable j, which will be used to mark how far through the array we have checked. We then introduce an abbreviation I for the predicate which will be used as the loop invariant, and split the specification into two, for the initialisation and the loop itself:

$$\sqsubseteq \quad \textbf{var}\ j \bullet$$
$$b, j : [b \Leftrightarrow \exists i : 1..n \bullet as\, i = x]$$
$$\sqsubseteq \quad I \mathrel{\widehat{=}} b \Leftrightarrow \exists i : 1..j \bullet as\, i = x \bullet$$
$$b, j : [true, I]; \qquad \triangleleft$$
$$b, j : [I, I \wedge j = n] \qquad (1)$$

The loop initialisation is easily implemented with a multiple assignment to b and j, and we can now introduce the loop, which has invariant I, guard $j \neq n$ and variant $n - j$.

$$\sqsubseteq \quad b, j := false, 0$$
$$(1) \sqsubseteq \quad \text{"}\textbf{variant}\ n - j\text{"}$$
$$\textbf{do}\ j \neq n \rightarrow$$
$$\qquad b, j : [I \wedge j \neq n, I \wedge 0 \leq n - j < n - j_0] \qquad \triangleleft$$
$$\textbf{od}$$

(Hints about the justification for a refinement step are often given as annotations to the refinement symbol, enclosed in "quotation marks". Initial variables are marked with a subscript zero.) The loop body is refined with a following assignment to increment j.

$$\sqsubseteq \quad \text{"following assignment, contract frame"}$$
$$b : [I \wedge j \neq n, I[j \backslash j + 1]]; \qquad \triangleleft$$
$$j := j + 1$$

The remaining part of the loop body is implemented (in a very crude way) using two nested alternations. The first alternation tests the value of b — if it

is true, then x has already heen found, and so we need do nothing.

$$\sqsubseteq \quad \textbf{if } b \longrightarrow \{b\}b : [I \wedge j \neq n, I[j \backslash j+1]] \quad \triangleleft$$
$$[] \neg b \longrightarrow \{\neg\, b\}b : [I \wedge j \neq n, I[j \backslash j+1]] \quad (2)$$
$$\textbf{fi}$$

$$\sqsubseteq \quad \textbf{skip}$$

In the case where b is *false*, we test the value of $as(j+1)$ — if it is equal to x, we set b to *true*, and, if not, we move on to test the next value.

$$(2) = \quad b : [(\neg\ \exists i : 1..j \bullet as\, i = x) \wedge j \neq n, I[j \backslash j+1]]$$

$$\sqsubseteq \quad \textbf{if } as(j+1) = x \longrightarrow b : \left[\begin{array}{c} \neg\ \exists i : 1..j \bullet as\, i = x \\ j \neq n \\ as(j+1) = x \end{array}\right. , I[j \backslash j+1] \left.\right] \quad \triangleleft$$
$$[]\ as(j+1) \neq x \longrightarrow b : \left[\begin{array}{c} \neg\ \exists i : 1..j \bullet as\, i = x \\ j \neq n \\ as(j+1) \neq x \end{array}\right. , I[j \backslash j+1] \left.\right] \quad (3)$$
$$\textbf{fi}$$

$$\sqsubseteq \quad b := \mathit{true}$$

$$(3) \sqsubseteq \quad \textbf{skip}$$

Collecting the code from the development tree gives the following program:

```
var j •
b, j := false, 0;
do j ≠ n →
    if b      → skip
    [] ¬ b    → if as(j+1) = x  → b := true
                [] as(j+1) ≠ x  → skip
                fi
    fi ;
    j := j + 1
od
```

It is certainly not the most efficient code to solve this problem — it would clearly be better to 'drop out' of the loop as soon as an occurrence of x is found — but it is correct! In Section 3.3, we give a much simpler and more efficient solution to a very similar problem, using the **loop/exit/end** construct defined there.

2.5 Conclusion

In this chapter, we have given a brief introduction to the refinement calculus: we have described the basic constructs of the language and given their semantics in terms of Dijkstra's weakest precondition. The chapter concluded with a discussion of some more advanced features of the calculus and the development of a simple program.

Part II

Exception handling

Chapter 3

Adding simple exceptions to the refinement calculus

In this chapter, we show how a form of exception mechanism can be added to the refinement calculus. It is deliberately not a complex mechanism: our concern is to discover, in the simplest possible context, what additional semantic notions are needed for exceptions, and to give a mechanism which can be later be used to model the more sophisticated exception mechanisms that are found in real programming languages.

We start with an examination of the need for exceptions in programming languages. Having convinced ourselves that we are not chasing a complete red herring, we show how to add a very simple exit mechanism to the refinement calculus. This mechanism is based on the distinction between normal and exceptional termination: in addition to terminating normally, certain program constructs are now also permitted to terminate exceptionally. We propose some algebraic laws for the new constructs, using our intuition about their behaviour to guide us. Section 3.3 shows how to deal with recursion in our extended language, giving a law for iteration and proposing a loop/exit/end construction. The last section of this chapter gives a formal basis to the previous work. In it, we extend Dijkstra's weakest precondition semantics to cover predicate transformers which take more than one postcondition as arguments — we need two postconditions, to describe conditions for normal and exceptional termination. Given this semantic framework, we are then able to justify the laws proposed earlier, both the simple ones and those involving recursion.

In the next chapter, we will extend this scheme to deal with named exceptions, and the association of program fragments with those exceptions.

3.1 The need for exceptions

Interest in exception handling as a concept in programming language design arose in the early 1970s out of the increasing realisation of the importance of abstraction and modularity. Language designers wanted to allow programmers to write 'robust software' which would continue to function (or at least to behave predictably) under whatever circumstances it might be used. As larger programs were written, often consisting of many levels of procedure call, it became important to specify the precise effect of a procedure. But when things 'go wrong' — there is an arithmetic overflow or a subscript out of range, for instance — there is the question of who is in the best position to decide on the appropriate recovery action: is it the writer of the procedure which has run into problems, or the caller of that procedure? Parnas noted [51] that it has to be the caller, and that the possibility (however remote) of such errors was an important part of the procedure's interface. It is not difficult to appreciate that a condition such as not finding a particular value while searching an array might be an error in some circumstances, and expected in others: the appropriate action in the two cases could be very different. If exceptions are not used, then either the invoking procedure has to pass more information to the invoked procedure to enable it to interpret the 'error' correctly, or the invoking procedure needs to include code around each invocation to ensure that inputs are in the required range and so on. But these checks may also be carried out in the invoked procedure, or they may more easily be performed there, and in any case, certain exceptional states (for instance, lack of resources) may be impossible to detect prior to the invocation. Thus exceptions can be seen as conditions detected while a procedure is being performed that need to be brought to the attention of the invoker of the procedure, so that appropriate action can be taken. The use of exceptions is one way to generalise an operation: by specifying that under certain circumstances an exception will be raised, rather than leaving the operation undefined, the programmer can make the procedure more generally useful.

Goodenough [20] has identified three potential uses for exceptions:

- to deal with impending failure;
- to give additional information about a valid result; and
- to monitor the progress of an operation.

An invoked procedure may 'fail' in one of two ways: on its domain or on its range. Domain failures occur when the inputs are outside the precondition of the procedure, while range failures are caused by the procedure's inability to achieve its postcondition. This may be because of the failure of some lower-level component, or it may be a problem with resource depletion, for instance. The exception mechanism that we introduce below is chiefly concerned with the use of exceptions for notification of failure.

Goodenough's other potential uses for exceptions are perhaps not so common. One could imagine a browsing operation on a sequential data structnre, which. at the same time as returning the final element, gave the user a warning that there were no more elements to be retrieved, thus saving a subsequent browse which was doomed to failure. Exceptions could be used to give this sort of additional information abont valid results of an operation. It would also be possible to use exceptions to monitor an operation, perhaps by writing a message to the console explaining how the operation is progressing — for instance. what percentage of a search has been completed -- and enquiring whether the user wants to continue.

One of the reasons that we concentrate on the first of Goodenough's uses for exceptions is that it fits most easily with our model of exceptiou handling: in designing an exception mechanism, there is a fundamental choice between the resumption model and the termination model. The question that distinguishes between these two models is 'Does the raiser of an exception continue to exist after the exception has been raised?' In the resumption model, the answer is yes, and it is possible that whatever code handles the exception might solve the problem and return coutrol to the point where the exception was raised. In the termination model, it is assumed that it will not he worth returning to the point of raising the exception, and so the whole procedure is terminated. The choice between the two models is a balance between expressive power and simplicity in the semantics. Although the resumption model is more complex, leading to complications in the relationships between procedures[1] and the specification of procedures, it seems to offer a more general approach. However, Liskov and Snyder [31, Section V] claim tbat the termination model, because of its simplicity, is preferable to the resumption model, provided -- and this is an important proviso -- that it supplies 'adeqnate expressive power'. They go on to discuss situations that are 'handled awkwardly' by the termination model and 'simply' by tbe resumption model, and claim that such situations do not arise frequently in practice. Following their lead from the design of the exception handling mechanism in CLU, we use a mechanism based on the termination model.

3.2 Syntax for exceptions

In order to be able to develop programs with exceptions, we need to extend the language of the refinement calculus. The main change is to alter the specification statement, but we will later define some other useful additional notation.

[1] Normally, ignoring recursion, a calling procedure is dependent on a procedure which it calls, relying on it to perform some computation. However, in the resumptiou model, the calling procedure and the raiser of the exception are mutually dependent: the caller depends on the exception raiser in the normal way, but the exception raiser also depends on the handler, which is part of the calling procedure, to perform some action when an exception is raised

The exceptional specification statement

Our first addition to traditional refinement calculus notation involves a generalisation of the notion of the postcondition of an operation. Since we are considering exceptional behaviour, it is no longer enough to consider a single postcondition for an operation to represent the condition which must hold when the operation terminates. Instead, we consider two postconditions — one for normal termination and one for exceptional termination. We therefore need to extend the specification statement. In the simple form of the refinement calculus, without exceptions, the specification statement contains a precondition and a single postcondition. We now consider a specification statement with a precondition and a pair of postcouditions — one for normal, and one for exceptional behaviour. Thus we write

$$w : [\alpha, \beta > \gamma]$$

for a specification which, when α is initially true, is guaranteed either

- to terminate normally, satisfying β; or
- to terminate exceptionally, satisfying γ.

As before, only variables in the frame w may be changed. All logical connectives are assumed to bind more tightly than $>$.

The formal semantics of this exceptional specification statement will be given in Section 3.4, when we have introduced the extended version of *wp* which is necessary to deal with exceptional termination. In fact, our extension of the specification statement arises naturally from the extension of the *wp* predicate transformer.

The connection with the original specification statement, which has only a single postcondition, is given by taking *false* as the exceptional postcondition:

$$w : [\alpha, \beta] = w : [\alpha, \beta > \mathit{false}] \ .$$

In the same way that **skip**, **abort** and **magic** are special cases of the traditional specification statement, there are two special cases of the exceptional specification statement. The more important is obtained by taking an empty frame, with *true* as the precondition and exceptional postcondition, and *false* as the normal postcondition:

$$\mathbf{exit} \mathrel{\widehat{=}} : [\mathit{false} > \mathit{true}] \ .$$

(As is usual in the refinement calculus, the *true* precondition has been omitted.) Execution of **exit** always causes exceptional termination, with no change to any variable.

The second special case of the exceptional specification is obtained by taking *true* as the precondition and both of the postconditions. This statement does

not seem sufficiently useful to require a name, but it can easily be expressed as a nondeterministic choice between **skip** and **exit**:

$$w : [true, true \succ true] = \mathbf{skip} \sqcap \mathbf{exit} \ .$$

This statement is always guaranteed to terminate, but that termination may be normal or exceptional.

The final piece of notation that we need at present is the exception block, which shows the extent of the scope of an exceptional termination. For instance, an **exit** occurring inside a pair of block brackets $[\![\ ]\!]$ causes control to be passed to the statement following the closing bracket. The laws for introducing blocks will show that they can be nested but not otherwise overlapping.

Simple laws

We can immediately propose some simple algebraic laws which show how these new language constructs should interact. These laws will be proved sound later, when we have set up the semantics for exceptions.

Law 3.1 *program after exit*

A program following **exit** has no effect.

$$\mathbf{exit}; aaa = \mathbf{exit}$$

(Equality of programs means semantic equivalence, that is, mutual refinement.)

Law 3.2 *exit ending block*

An **exit** at the end of an exception block has no effect.

$$[\![aaa; \mathbf{exit}]\!] = [\![aaa]\!]$$

Law 3.3 *exception-free block*

Exception blocks have no effect on exception-free programs.

$$[\![aaa]\!] = aaa \qquad \textit{provided } aaa \textit{ is exception-free}$$

Section 3.4 contains a weakest precondition characterisation of the idea of a program being exception-free, but for now we can think of it as "syntactically without occurrences of **exit**s or specification statements with exceptional post-conditions".

With these three laws alone, and the usual laws of the refinement calculus, we can show the equivalence of some simple code fragments. For instance,

$$\mathbf{if}\ \alpha\ \mathbf{then}\ aaa\ \mathbf{fi}^2 \ ,$$

can be shown to be equivalent to the following, which is often used when *aaa* is long, and α tests for some error condition:

⟦ **if** $\neg\alpha$ **then exit fi**; *aaa* ⟧ .

(The equivalence only holds when *aaa* is exception-free.) Although the following algebraic derivation may seem daunting in length for an essentially simple result, we set it out in full simply to illustrate clearly the nature of such reasoning.

if α **then** *aaa* **fi**

= "*exception-free block* 3.3, provided *aaa* is exception-free"

⟦**if** α **then** *aaa* **fi**⟧

= "*exit ending block* 3.2"

⟦**if** α **then** *aaa* **fi**; **exit**⟧

= "definition of **if**..**then**..**fi**"

⟦ **if** α → *aaa*
[] $\neg\alpha$ → **skip**
fi ;
exit
⟧

= "distribution of ; **exit** into **if** "

⟦ **if** α → *aaa*; **exit**
[] $\neg\alpha$ → **skip**; **exit**
fi
⟧

= "**skip** left identity of ; and *program after exit* 3.1"

⟦ **if** α → **skip**; *aaa*; **exit**
[] $\neg\alpha$ → **exit**; *aaa*; **exit**
fi
⟧

= "distribution out of **if** "

⟦ **if** α → **skip**
[] $\neg\alpha$ → **exit**
fi ;
aaa; **exit**
⟧

= "definition of **if**..**then**..**fi** and *exit ending block* 3.2"

⟦ **if** $\neg\alpha$ **then exit fi**;
aaa
⟧ .

We have thus shown the eqnivalence as programs of

if α **then** *aaa* **fi**

[2] An alternative notation for **if** α → *aaa* [] $\neg\alpha$ → **skip fi**

and

$$[\![\textbf{if } \neg\alpha \textbf{ then exit fi};\ aaa]\!]$$

nsing only the simple equivalences of Laws 3.1–3.3.

In Section 3.2 we will be able to show a much shorter derivation of the same result.

Further development laws

The laws given in the previous section are clearly not powerful enough to allow us to make all the development steps we would like. In particular, they are all equations, whereas we would expect some laws which actually involve refinements; and there is no law for introducing exceptions into a program, other than with an **exit** right at the end of an exception block.

The first additional law that we now give allows us to convert a traditional specification statement into an exceptional one:

Law 3.4 *exceptional specification*

An exceptional specification can be formed by duplicating the postcondition of a non-exceptional specification statement, and surrounding it with an exception block.

$w : [\alpha, \beta] = [\![w : [\alpha, \beta > \beta]]\!]$

We can refine a specification statement by discarding either the exceptional or the normal branch.

Law 3.5 *take normal branch*

A specification statement can be implemented by taking the normal branch unconditionally.

$w : [\alpha, \beta > \gamma] \sqsubseteq w : [\alpha, \beta]$

Law 3.6 *take exceptional branch*

A specification statement can be implemented by achieving the exceptional postcondition, and then performing an **exit**.

$w : [\alpha, \beta > \gamma] \sqsubseteq w : [\alpha, \gamma]; \textbf{exit}$

It is convenient to introdnce at this stage a further abbreviation which will make the layout of developments slightly easier: for programs *aaa* and *bbb*, we write

$$aaa > bbb$$

(pronounced "*aaa* else *bbb*") as a shorthand for

$$aaa \;[]\; (bbb; \mathbf{exit}) \ .$$

$aaa > bbb$ is a specification of a computation which, if it terminates normally, will have executed aaa; if it terminates exceptionally, it will have executed bbb. The choice between the two is arbitrary and unpredictable. This notation allows us to separate the normal and exceptional behaviours in a development, and therefore to continue their development separately. The relationship between this new construct and the specification statement is very simple:

Law 3.7 *else notation*

Specifications and the 'else' notation

$$w : [\alpha, \beta > \gamma] = w : [\alpha, \beta] > w : [\alpha, \gamma]$$

We can immediately give a few simple laws for the new 'else' construct; although we do not prove their soundness here, such proofs are easy exercises given the semantics of Section 3.4.

Law 3.8 *take normal branch*

An 'else' construct can be implemented by taking the first branch unconditionally.

$$aaa > bbb \ \sqsubseteq \ aaa$$

Law 3.9 *take exceptional branch*

An 'else' construct can be implemented by taking the second branch unconditionally, followed by an **exit**.

$$aaa > bbb \ \sqsubseteq \ bbb; \mathbf{exit}$$

Law 3.10 *choice-else*

A nondeterministic choice between two programs which do not contain exceptions is equivalent to an exception block containing the programs as branches of an 'else' construct.

$$aaa \;[]\; bbb = [\![aaa > bbb]\!] \qquad \textit{provided } aaa \textit{ and } bbb \textit{ are exception-free}$$

Law 3.11 *introduce trivial else*

An **exit**-exception pair can be introduced by offering the trivial choice between equal alternatives (corollary to *choice-else* 3.10).

$$aaa = [\![aaa > aaa]\!] \qquad \textit{provided } aaa \textit{ is exception-free}$$

We see that *choice-else* 3.10 and *introduce trivial else* 3.11 are the crucial laws which allow us to introduce the else construct (and thus the possibility of an exception) into a program which previously did uot contain one.

The laws above are reasonably straightforward. Since 'taking an **exit**' alters control flow in a program, however, we might expect the interaction of **exit** and sequential composition to be less obvious:

Law 3.12 *sequential composition*

Distribute sequential composition through 'else'.

$$aaa > bbb \sqsubseteq (ccc > bbb)\ ;\ (ddd > eee) \qquad \text{provided } aaa \sqsubseteq ccc;\ ddd \text{ and } bbb \sqsubseteq ccc;\ eee$$

The law is informally justified by examining the three possible behaviours of the right-hand side: the first is that *ccc*; *ddd* terminates normally, and refines the normal termination path *aaa* of the left-hand side; in the second we find that *bbb* terminates exceptionally, and equals the exceptional termination path of the left-hand side; finally *ccc*; *eee* terminates exceptionally, and refines the exceptional termination path of the left-hand side.

We can also give a somewhat simpler law for introducing a sequential composition into a specification statement:

Law 3.13 *sequential composition*

Splitting a specification with sequential composition.

$$w : [\alpha, \beta > \gamma] \sqsubseteq w : [\alpha, \delta > \gamma]\ ;\ w : [\delta, \beta > \gamma]$$

We can now return to the example of the previous section. Use of the else construct allows a much more concise development.

if α **then** *aaa* **fi**

= "definition of **if..then..fi**, and removing **if..fi**"

$\alpha \rightarrow aaa$

$[\!]\ \neg\alpha \rightarrow \mathbf{skip}$

= "*choicc-else* 3.10, assuming *aaa* is exception-free"

$[\![$

$\alpha \rightarrow aaa > \neg\alpha \rightarrow \mathbf{skip}$ (1)

$]\!]$

(1) $\sqsubseteq$ "*sequential composition* 3.12, justification below"

$(\alpha \rightarrow \mathbf{skip} \mathbin{>} \neg\alpha \rightarrow \mathbf{skip})$; (2)

$aaa \mathbin{>} \mathbf{magic}$ (3)

(2) $\sqsubseteq$ "expand >, definition of **if..then..fi**"

if $\neg\alpha$ **then exit fi**

(3) $\sqsubseteq$ "*take normal branch* 3.8"

aaa .

That concludes the development; collecting the code gives

$[\![$ **if** $\neg\alpha$ **then exit fi**; aaa $]\!]$

as before.

The two side-conditions for the application of *sequential composition* 3.12 are satisfied as follows: we require first that $\alpha \rightarrow aaa \sqsubseteq (\alpha \rightarrow \mathbf{skip})$. aaa (easily checked with *wp*; alternatively viewed as a sort of associativity of $\rightarrow$ and ;). We also require that $\neg\alpha \rightarrow \mathbf{skip} \sqsubseteq \alpha \rightarrow \mathbf{skip}$; **magic** (not so obvious, but in fact the right-hand side simplifies to **magic** on its own).

This development is much shorter than the previous version, but it should be noted that we have only proved refinement, not equality as before. Notice also that **magic** has appeared in our development, though we have not needed to implement it (fortunately); we have used *take normal branch* 3.8, finally, to discard it by choosing the left-hand side of the 'else' construct.

3.3 Recursion

The laws presented above can — in principle — convert any 'finitary' program that contains exceptions into an equivalent program that is exception-free. The same is not true for ('infinitary') programs which contain recursion, either explicitly or implicitly.

Explicit recursion is usually found in the form of recursive procedure calls, in which a given procedure A, say, contains a call to the same A within it. (For simplicity, we consider only the case where there are no parameters.) These two notions — recursion and procedure call — are not inextricably linked however; we separate them by using the recursion block

mu $X \bullet \mathcal{P}(X)$ **um** ,

thus freeing us to deal with recursion on its own. Further details can be found in [44].

The meaning of the above is the least fixed point of the program-to-program function $\mathcal{P}$. (For an example, see the treatment of loops below.) More precisely, we consider $\mathcal{P}$ to have type $PT \rightarrow PT$, and to be monotonic. The type PT,

in turn, is $\mathbf{P}\,S \to \mathbf{P}\,S$: predicate transformers taking sets of (final) states to sets of (initial) states. The set S of states is fixed throughout the discussion, but in practice would be large enough to contain the standard and constructed types. Recursive procedures are no longer entities to be distinguished in their own right; but one might say that a procedure was recursive if its body were a recursion block.

Implicit recursion is that introduced by iteration. The **do...od** coustruction in the guarded command language is — for us — defined as follows:

do $G \to$ *body* **od**

is equivalent to the following recursion block (in which D is a fresh identifier):

mu D •
  if G **then** *body*; D **fi**
um .

The *body* of an iteration can be any program at all. For instance, takiug a rather extreme case. it might be **magic**, so we could have an iteration

do *true* $\to$ **magic od** .

This program can be simplified as follows:

do *true* $\to$ **magic od**
  = "unwinding the recursive definition once"
    if *true* **then (magic**; **do** *true* $\to$ **magic od) fi**
  = "removing **if** *true*"
    magic; **do** *true* $\to$ **magic od**
  = "**magic** absorptive"
    magic .

Thus **magic** even 'jumps out of infinite loops'.

Refining to recursion

In order for recursion to appear in a program whose specification did not contain it, there must be a refiuement step whose right-hand side introduces a recnrsion block. Temporarily ignoring the matter of termination, a law to justify such a step might be

If $aaa \sqsubseteq \mathcal{P}(aaa)$, then $aaa \sqsubseteq$ **mu** $D \bullet \mathcal{P}(D)$ **um** ,

given some monotonic program-to-program function $\mathcal{P}$.

We can take terminatiou into account — as we **must**, to avoid the absurd 'everything is refined by **mu** D • D **um**' — by a **small** amount of trickery

involving logical constants, assnmptions, and a variant function. (These issues are explained in more detail in Section 2.3 and [44].)

The (non-exceptional) law for recursion introduction [44] is as follows:

Law 3.14 *recursion*

Let e be an integer-valued expression, V a logical constant, aaa a program, and $\mathcal{P}$ a monotonic program-to-program function, and assume that neither aaa nor $\mathcal{P}$ contains V. Then if

$$\{e = V\}\, aaa \;\sqsubseteq\; \mathcal{P}(\{0 \le e < V\}\, aaa)$$

we can conclude

$$aaa \sqsubseteq \textbf{mu}\ D \bullet \mathcal{P}(D)\ \textbf{um}\ .$$

In practice, $\mathcal{P}$ will always he huilt from the constructs of the language, and so it is guaranteed to he a monotonic function since the constructs themselves are monotonic.

The *variant function* for a recursion is an integer expression that is bounded below, yet is strictly decreased on each recnrsive call. Although the requirement for the variant to be integer-valued is sufficient for our needs, it is stronger than necessary: there exist programs which need ordinal-valued variants — see, for example, [12]. We show in in Section 3.4 below that the above law remains valid in the presence of exceptions — and the proof uses transfinite induction. For now we proceed, on that assumption, with the presentation of recursion and iteration.

Iteration

We recall [44, Law 5.5] that the law used in the refinement calculus for introducing iteration (without exceptions) is

$$\begin{array}{l} w : [\alpha, \alpha \wedge \neg G] \\ \sqsubseteq \\ \textbf{do}\ G \rightarrow \\ \quad w : [G \wedge \alpha\ , \alpha \wedge (0 \le e < e_0)] \\ \textbf{od}. \end{array}$$

The conventional conditions for loop correctness appear in the above as follows, given that the invariant is α and the variant[3] is e: the invariant is true

[3]Zero-subscripted variables in a postcondition are used to refer to the values of those variables in the initial state. They are defined in terms of logical constants, and are a very convenient abbreviation for the sort of specifications we wish to write. A zero-subscripted expression, like e_0 here, is an abbreviation for the expression with all variable occurrences zero-subscripted.

initially (indicated by the precondition of the left-hand side); its truth finally, and the negated guard, are sufficient to establish the desired result (shown by the postcondition of the left-hand side); the invariant is maintained by the loop body (it appears both in the pre- and postconditious of the loop body, on the right-hand side); the variant is strictly decreased (postcondition of the body); the variant is bounded below (postcondition of the body).

To incorporate exceptions into the above we can show first, using the techniques of earlier sections[4], that

$$
\begin{array}{l}
\{e = V\}\ w : [\alpha, \alpha \wedge \neg G > \beta] \\
\sqsubseteq \\
\textbf{if } G \textbf{ then} \\
\quad w : [\alpha \wedge G\ , \alpha \wedge (0 \leq e < e_0) > \beta]; \\
\quad \{0 \leq e < V\}\ w : [\alpha, \alpha \wedge \neg G > \beta] \\
\textbf{fi}\ ,
\end{array}
$$

which matches the condition for recursion introduction in *recursion* 3.14. The proof of this is not complicated; rather than give it formally, however, we sketch an argument as follows. We consider separately the two cases distinguished by whether $\neg G$ is true initially. If $\neg G$ holds initially, the left-hand side is refined by **skip** because the required postcondition of the normally-terminating branch holds already (α in the precondition, $\neg G$ assumed). Given $\neg G$ initially, the right-hand side equals **skip**.

If G holds initially, then the right-hand side either

1. terminates normally having executed
   $w : [\alpha \wedge G\ , \alpha \wedge (0 \leq e < e_0)]$; $\{0 \leq e < V\}\ w : [\alpha, \alpha \wedge \neg G]$; or
2. terminates exceptionally having executed
   $w : [\alpha \wedge G\ , \alpha \wedge (0 \leq e < e_0)]$; $w : [\alpha, \beta]$; or
3. terminates exceptionally having executed $w : [\alpha \wedge G, \beta]$.

In all three cases, the postcondition established by the right-band side is appropriate for the mode of termination (as given on the left-hand side): $\alpha \wedge \neg G$ normally and β exceptionally.

Thus we can conclude from *recursion* 3.14 that

$$
\begin{array}{l}
w : [\alpha, \alpha \wedge \neg G > \beta] \\
\sqsubseteq \\
\textbf{mu } D \bullet \\
\quad \textbf{if } G \textbf{ then} \\
\quad\quad w : [\alpha \wedge G\ , \alpha \wedge (0 \leq e < e_0) > \beta]; D \\
\quad \textbf{fi} \\
\textbf{um}\ ,
\end{array}
$$

[4] We recall that the logical connectives bind more tightly than $>$, so, for example, the specification statement on the LHS of this refinement is parsed as $w : [\alpha, (\alpha \wedge \neg G) > \beta]$.

which gives us the iteration law required:

Law 3.15 *iteration*

$$
\begin{array}{l}
w : [\alpha, \alpha \wedge \neg G > \beta] \\
\sqsubseteq \\
\textbf{do } G \rightarrow \\
\quad w : [\alpha \wedge G \ , \alpha \wedge (0 \leq e < e_0) > \beta] \\
\textbf{od}
\end{array}
$$

Compared to the iteration law for the refinemeut calculus without exceptions, the extended law simply contains "$> \beta$" in both postconditions. It operates analogously to the non-exceptional version if it terminates normally; however, the loop body is provided with the possibility of exceptional termination, when β must be established. as demanded of the overall exceptional postcondition. The exceptional branch may assume (additionally) G, since the loop body would not be executed if G were not true.

Loop/exit/end

As an application of the above, we consider the **loop/exit/end** construction found in Modula-2 (or equivalently the while/break construction of C). This is defined to be equivalent to a **do..od** loop with a *true* guard, enclosed in an exception block:

$$
\begin{array}{l}
\textbf{loop } aaa \textbf{ end} \\
= \\
[\![\textbf{ do } true \rightarrow aaa \textbf{ od}]\!] \ ,
\end{array}
$$

where the *aaa* will usually include an **exit** command, to ensure loop termination.

We proceed as follows to construct a rule for introducing **loop** into a program; note that the (extreme) strengthening of the postcondition for the non-exceptional case to *false* effectively forces exceptional termination, which is the way **loop/exit/end** behaves.

$$
\begin{array}{l}
w : [\alpha, \beta] \\
= \text{ "application of } \textit{introduce trivial else } 3.11 \text{ above"} \\
\quad [\![w : [\alpha, \beta > \beta]]\!] \\
\sqsubseteq \text{ "strengthen normal postcondition"} \\
\quad [\![w : [\alpha, false > \beta]]\!] \\
\sqsubseteq \text{ "}\textit{iteration } 3.15\text{"} \\
\quad [\![\textbf{ do } true \rightarrow \\
\qquad w : [\alpha \wedge true \ , \alpha \wedge (0 \leq e < e_0) > \beta]
\end{array}
$$

od

$[\!]$

Removing the superfluous "$\wedge$ *true*" gives the following law:

Law 3.16 *loop introduction*

$$
\begin{array}{l}
w : [\alpha, \beta] \\
\sqsubseteq \\
\textbf{loop} \\
\quad w : [\alpha\ , \alpha \wedge (0 \leq e < e_0) > \beta] \\
\textbf{end}
\end{array}
$$

Thus the task of the non-exceptional part of the loop body is to maintain α (the invariant) while strictly decreasing the variant e on each iteration (but not below 0). Since that cannot continue indefinitely, eventually the exceptional route must be taken, after establishing β as the left-hand side requires.

Note that, like *introduce trivial else* 3.11, this is a law which introduces an exception and a corresponding block together -- the extent of the exception block is taken to be the **loop**...**end** block, so an **exit** in the body of the loop will cause a jump to the bottom of the loop, just after the **end** statement.

Example

To show this law in action, we develop a program which is intended to find the index of a particular value guaranteed to occur in an array. We make the following variable declarations:

$$
\begin{array}{l}
as : \textbf{array}\ [0..N-1]\ \textbf{of}\ A \\
a : A \\
i : 0..N
\end{array}
$$

The development is as follows:

$$
\begin{array}{ll}
 & i : [a \in as[0..N-1], a = as[i]] \\
\sqsubseteq & i := 0; \\
 & i : [a \in as[i..N-1], a = as[i]] \qquad (1) \\
(1) \sqsubseteq & \text{"}\textit{loop introduction}\ 3.16\text{, invariant } a \in as[i..N-1]\text{; variant } N - i\text{"} \\
 & \textbf{loop} \\
 & \quad i : \left[a \in as[i..N-1],\ \begin{array}{c} a \in as[i..N-1] \\ i_0 < i \leq N \end{array} \ > \ a = as[i] \right] \qquad (2) \\
 & \textbf{end} \\
(2) \sqsubseteq & \text{"}\textit{else notation}\ 3.7\text{"} \\
 & i : [a \in as[i..N-1], a \in as[i..N-1] \wedge i_0 < i \leq N] \qquad (3) \\
 & [\!]\ i : [a \in as[i..N-1], a = as[i]]; \textbf{exit} \qquad (4) \\
(3) \sqsubseteq & a \neq as[i] \rightarrow i := i + 1 \\
(4) \sqsubseteq & a = as[i] \rightarrow \textbf{exit}
\end{array}
$$

Notice that the exceptional and the non-exceptional branches are refined to naked guarded commands which, combined by [], lead to the expected alternation in the loop body. The justifications for these final refinements to naked guarded commands are omitted, but they can easily be checked with *wp*.

We can collect the code to give:

```
i := 0;
loop
    a ≠ as[i] → i := i + 1
    [] a = as[i] → exit
end ,
```

which we may rewrite, using the definition of **if..then..fi** and rules from Section 3.2, as

```
i := 0;
loop
    if a = as[i] then exit fi ;
    i := i + 1
end .
```

3.4 Semantics

Weakest preconditions for languages with exceptions

The traditional weakest precondition technique for giving semantics to a language involves defining a function *wp*, which, for any statement *aaa* in the language, returns a predicate-to-predicate function (a predicate transformer). The function $wp(aaa)$ maps a postcondition α to the weakest precondition β from which *aaa* is guaranteed to terminate satisfying α. For example, the weakest precondition of the simple assignment statement $x := E$ is given by

$$wp(x := E, \alpha) = \alpha[x \backslash E] \ .$$

This method of giving semantics for languages without exceptions is not sufficiently powerful for our needs, because we have to distinguish between normal and exceptional termination. Following Back [6] and Manasse and Nelson [37], we therefore introduce a predicate transformer which is a function of two arguments rather than the usual one. We use the notation

$$wp(aaa, \nu, \epsilon)$$

where *aaa* is a program, and ν and ϵ are predicates, to denote

the weakest precondition from which *aaa* is guaranteed either:

- to terminate normally satisfying ν (for *normal*); or

Syntax	Semantics
$x := E$	$wp(x := E, \nu, \epsilon) = \nu[x \backslash E]$
$aaa;\ bbb$	$wp(aaa;\ bbb, \nu, \epsilon) = wp(aaa, wp(bbb.\ \nu, \epsilon), \epsilon)$
$\mathbf{if}([]i \bullet \alpha_i \rightarrow aaa_i)\ \mathbf{fi}$	$wp(\mathbf{if}([]i \bullet \alpha_i \rightarrow aaa_i)\mathbf{fi}, \nu, \epsilon) =$ $(\bigvee i \bullet \alpha_i) \wedge (\bigwedge i \bullet \alpha_i \Rightarrow wp(aaa_i, \nu, \epsilon))$
$\mathbf{mu}\ aaa \bullet \mathcal{P}(aaa)\ \mathbf{um}$	given by least fixed point: see below
$w : [\alpha, \beta > \gamma]$	$wp(w : [\alpha, \beta > \gamma], \nu, \epsilon) =$ $\alpha \wedge (\forall w \bullet \beta \Rightarrow \nu) \wedge (\forall w \bullet \gamma \Rightarrow \epsilon)$
$aaa\ []\ bbb$	$wp(aaa[]bbb, \nu, \epsilon) = wp(aaa, \nu, \epsilon) \wedge wp(bbb, \nu, \epsilon)$
$\alpha \rightarrow aaa$	$wp(\alpha \rightarrow aaa, \nu, \epsilon) = \alpha \Rightarrow wp(aaa, \nu, \epsilon)$

Figure 3.1: Weakest precoudition semantics

- to terminate exceptionally satisfying ϵ (for *exceptional*).

Now we can give a compositional semantics to our language, using this notation. For any construct which was in the language before we added **exits**, say *ppp*, the correspondiug new weakest precondition definition is given by

$$wp(ppp, \nu, \epsilon) \mathrel{\widehat{=}} wp(ppp, \nu)$$

(where the *wp* on the left is our uew version, and the *wp* on the right is the standard Dijkstra *wp*). That is because the 'original' constructs terminate normally by definition — they contain no **exits**. Since they cannot terminate exceptionally, the right-hand side is independent of ϵ. For instance, the commands **skip** and **abort** are given meaning thus:

$$wp(\mathbf{skip}, \nu, \epsilon) = \nu$$
$$wp(\mathbf{abort}, \nu, \epsilon) = \mathit{false}$$

The other constructs of the language (apart from recursion) have defining equations very similar to the usual (Dijkstra) *wp* equations: these are given in Figure 3.1.

Notice that, in Figure 3.1, we have given a *wp* definition to the exceptional specification statement $w : [\alpha, \beta > \gamma]$. As we remarked earlier, the simple specification statement is a special case:

$$w : [\alpha, \beta] = w : [\alpha, \beta > \mathit{false}]\ .$$

We can therefore calculate its weakest precondition:

$$\begin{aligned} & wp(w : [\alpha, \beta], \nu, \epsilon) \\ & \quad = wp(w : [\alpha, \beta > \mathit{false}], \nu, \epsilon) \\ & \quad = \alpha \wedge (\forall w \bullet \beta \Rightarrow \nu) \wedge (\forall w \bullet \mathit{false} \Rightarrow \epsilon) \\ & \quad = \alpha \wedge (\forall w \bullet \beta \Rightarrow \nu)\ , \end{aligned}$$

which agrees with the definition in [42].

More interesting are the equations for the **exit** command and the exception block:

$$wp(\mathbf{exit}, \nu, \epsilon) = \epsilon$$
$$wp([\![aaa]\!], \nu, \epsilon) = wp(aaa, \nu, \nu) \ .$$

The equation for the exception block reflects the fact that any **exit** inside *aaa* will be caught by the exception block.

Using these weakest precondition definitions, we can verify the laws which were given earlier, and justified at that stage only in terms of the informal operational semantics. For instance, *exit ending block* 3.2 states

$$[\![\ aaa; \mathbf{exit} \ ]\!] = [\![\ aaa \ ]\!] \ .$$

Taking weakest preconditions, we find

$$\begin{aligned} & wp([\![\ aaa; \mathbf{exit} \ ]\!], \nu, \epsilon) \\ & \quad = wp(aaa; \mathbf{exit}, \nu, \nu) \\ & \quad = wp(aaa, wp(\mathbf{exit}, \nu, \nu), \nu) \\ & \quad = wp(aaa, \nu, \nu) \\ & \quad = wp([\![\ aaa \ ]\!], \nu, \epsilon) \ . \end{aligned}$$

Many of the laws given earlier have side-conditions stating that certain components must be **exit**-free, where the obvious test for **exit**-freeness is syntactic. But we can be more precise if we use a weakest precondition characterisation of the concept:

aaa is **exit**-free iff $wp(aaa, \nu, \epsilon) = wp(aaa, \nu, \epsilon')$ for any ϵ, ϵ' .

Now we can verify *choice-else* 3.10, for instance. We need to show that

$$aaa \ [\!] \ bbb = [\![\ aaa > bbb \ ]\!]$$

given that *aaa* and *bbb* are both **exit**-free. Taking the weakest precondition on the right, we obtain

$$\begin{aligned} & wp([\![\ aaa > bbb \ ]\!], \nu, \epsilon) \\ & \quad = wp(aaa > bbb, \nu, \nu) \\ & \quad = wp(aaa \ [\!] \ (bbb; \mathbf{exit}), \nu, \nu) \\ & \quad = wp(aaa, \nu, \nu) \wedge wp(bbb; \mathbf{exit}, \nu, \nu) \\ & \quad = wp(aaa, \nu, \nu) \wedge wp(bbb, wp(\mathbf{exit}, \nu, \nu), \nu) \\ & \quad = wp(aaa, \nu, \nu) \wedge wp(bbb, \nu, \nu) \\ & \quad = \text{“since } aaa \text{ and } bbb \text{ are both } \mathbf{exit}\text{-free”} \\ & \qquad wp(aaa, \nu, \epsilon) \wedge wp(bbb, \nu, \epsilon) \\ & \quad = wp(aaa \ [\!] \ bbb, \nu, \epsilon) \ . \end{aligned}$$

Many of the other laws which were given earlier involve refinements, rather than just equalities. In order to verify these laws, we need a weakest precondition definition of refinement. Following [44] and other writers on the refinement calculus, our definition is as follows:

For any programs aaa and bbb, we say that aaa is refined by bbb, written $aaa \sqsubseteq bbb$, exactly when for all postconditions ν and ϵ,

$$wp(aaa, \nu, \epsilon) \Rightarrow wp(bbb, \nu, \epsilon) \ .$$

Now we can prove some of the laws that were given earlier. For instance, to prove Law 3.6, we must show that

$$w : [\alpha, \beta > \gamma] \sqsubseteq w : [\alpha, \gamma]; \ \mathbf{exit} \ .$$

Taking the weakest precondition of the left-hand side, we get

$$\alpha \wedge (\forall w \bullet \beta \Rightarrow \nu) \wedge (\forall w \bullet \gamma \Rightarrow \epsilon) \ ,$$

while the right-hand side gives

$$\begin{aligned} &wp(w : [\alpha, \gamma], wp(\mathbf{exit}, \nu, \epsilon), \epsilon) \\ &\quad = \alpha \wedge (\forall w \bullet \gamma \Rightarrow wp(\mathbf{exit}, \nu, \epsilon)) \\ &\quad = \alpha \wedge (\forall w \bullet \gamma \Rightarrow \epsilon) \ . \end{aligned}$$

So the weakest precondition of the left-hand side implies the weakest precondition of the right-hand side, as required.

A more complicated example is given by Law 3.12, for which we must show that

$$\begin{aligned} &aaa > bbb \\ &\sqsubseteq \\ &(ccc > bbb) \ ; \ (ddd > eee) \end{aligned}$$

when we know that

$$\begin{aligned} &aaa \sqsubseteq ccc; \ ddd \\ &\text{and } bbb \sqsubseteq ccc; \ eee \ . \end{aligned}$$

In terms of weakest preconditions, these provisos say

$$\begin{aligned} &wp(aaa, \nu, \epsilon) \Rightarrow wp(ccc; \ ddd, \nu, \epsilon) \\ &\text{and } wp(bbb, \nu, \epsilon) \Rightarrow wp(ccc; \ eee, \nu, \epsilon) \ . \end{aligned}$$

for all postconditions ν and ϵ.

Following a similar argument to the proof of *choice-else* 3.10 above, we can take the weakest precondition of the left-hand side of this law to get

$$\begin{aligned} &wp(aaa > bbb, \nu, \epsilon) \\ &\quad = wp(aaa \ [\!] \ (bbb; \ \mathbf{exit}), \nu, \epsilon) \\ &\quad = wp(aaa, \nu, \epsilon) \wedge wp(bbb; \ \mathbf{exit}, \nu, \epsilon) \\ &\quad = wp(aaa, \nu, \epsilon) \wedge wp(bbb, wp(\mathbf{exit}, \nu, \epsilon), \epsilon) \\ &\quad = wp(aaa, \nu, \epsilon) \wedge wp(bbb, \epsilon, \epsilon) \ . \end{aligned}$$

The right-hand side is slightly more complicated:

$$
\begin{aligned}
&wp((ccc > bbb) : (ddd > eee), \nu, \epsilon) \\
&\quad = wp(ccc > bbb.\, wp(ddd > eee, \nu, \epsilon), \epsilon) \\
&\quad = wp(ccc > bbb, wp(ddd.\, \nu, \epsilon) \wedge wp(eee.\, \epsilon, \epsilon).\, \epsilon) \\
&\quad = wp(ccc, wp(ddd, \nu, \epsilon) \wedge wp(eee, \epsilon, \epsilon), \epsilon) \\
&\qquad \wedge wp(bbb, \epsilon, \epsilon) \\
&\quad = \text{“by conjunctivity”} \\
&\qquad wp(ccc, wp(ddd, \nu, \epsilon), \epsilon) \\
&\qquad \wedge wp(ccc, wp(eee, \epsilon, \epsilon), \epsilon) \\
&\qquad \wedge wp(bbb, \epsilon, \epsilon)
\end{aligned}
$$

Now the first conjunct here is just $wp(ccc;\ ddd, \nu, \epsilon)$, and so we know that the first conjunct of the left-hand side implies this, by the side-condition. The second conjunct is $wp(ccc;\ eee, \epsilon, \epsilon)$, and we know that this is implied by the second conjunct of the left-hand side, since the definition of the refinement relation says that the implication holds *for all postconditions.* and so it must hold when we take ϵ for both postconditions. The third conjunct appears on the left-hand side in exactly the same form.

Recursion

As usual, the semantics of recursion is given by a least fixed point construction In general, given a program-to-program function $\mathcal{P}$ we write $\mu\mathcal{P}$ for its least fixed point, and take that to be the meaning of the syntax

mu $X \bullet \mathcal{P}(X)$ **um**

given in Section 3.3.

Rather than proving tbe recursion law directly (Law *recursion* 3.14 in Section 3.3 above), we will instead give a lemma from which it is easy to derive the law. We will give an outline proof of this lemma, noting that Greek letters denote ordinals, not predicates, for the duration of this lemma!

Recursion lemma Let an ordinal-indexed family of programs aaa_α be such that for any ordinal α

$$aaa_\alpha \sqsubseteq \mathcal{P}(\bigsqcup \beta \mid \beta < \alpha \bullet aaa_\beta) \quad ,$$

for some monotonic program-to-program function $\mathcal{P}$, where $\bigsqcup$ denotes least upper bound in the refinement ordering given above. Then

$$aaa_\alpha \sqsubseteq \mu\mathcal{P}$$

for all α.

Proof outline By transfinite induction on α, with all three cases together:

$$
\begin{array}{ll}
& aaa_\alpha \\
\sqsubseteq & \text{"assumption"} \\
& \quad \mathcal{P}(\bigsqcup \beta \mid \beta < \alpha \bullet aaa_\beta) \\
\sqsubseteq & \text{"}\bigsqcup,\mathcal{P} \text{ monotonic; inductive hypothesis"} \\
& \quad \mathcal{P}(\bigsqcup \beta \mid \beta < \alpha \bullet \mu\,\mathcal{P}) \\
\sqsubseteq & \text{"even when } \alpha = 0\text{"} \\
& \quad \mathcal{P}(\mu\,\mathcal{P}) \\
= & \text{"}\mu\,\mathcal{P} \text{ fixed point"} \\
& \quad \mu\,\mathcal{P} \ .
\end{array}
$$

□

Now we must show how to obtain the recursion law from this lemma.

Recursion Law (Law 3.14)
Let e be an integer-valued expression, V a logical constant, aaa a program, and $\mathcal{P}$ a monotonic program-to-program function, and assume that neither aaa nor $\mathcal{P}$ contains V. Then if

$$\{e = V\}aaa \ \sqsubseteq \ \mathcal{P}(\{0 \le e < V\}aaa)$$

we can conclude

$$aaa \sqsubseteq \ \mathbf{mu}\ D \bullet \mathcal{P}(D)\ \mathbf{um} \ .$$

Proof Let us define a family of programs aaa_α by

$$aaa_\alpha = \{e = \alpha\}aaa \ .$$

We may assume from the statement of the law that, for any α,

$$\{e = \alpha\}aaa \ \sqsubseteq \ \mathcal{P}(\{0 \le e < \alpha\}aaa).$$

But

$$
\begin{array}{ll}
& \{0 \le e < \alpha\}aaa \\
= & \quad \{(\bigvee \beta \mid \beta < \alpha \bullet e = \beta)\}aaa \\
= & \quad \text{"by wp"} \\
& \quad (\bigsqcup \beta \mid \beta < \alpha \bullet \{e = \beta\});\ aaa \\
= & \quad \text{"left-distribution of ; into } \bigsqcup\text{"} \\
& \quad (\bigsqcup \beta \mid \beta < \alpha \bullet \{e = \beta\}aaa) \\
= & \quad (\bigsqcup \beta \mid \beta < \alpha \bullet aaa_\beta) \ .
\end{array}
$$

So we have

$$aaa_\alpha \sqsubseteq \ \mathcal{P}(\bigsqcup \beta \mid \beta < \alpha \bullet aaa_\beta)$$

as required for the recursion lemma above, and we may conclude that, for any α,

$$aaa_\alpha \sqsubseteq \mu\,\mathcal{P} \ .$$

In other words,

$$\{e = \alpha\}aaa \sqsubseteq \mu\,\mathcal{P} \quad \text{for all } \alpha \ .$$

Since this holds for any α, we conclude that $aaa \sqsubseteq \mu\,\mathcal{P}$ simply by letting α range over all possible values of e permitted by our original choice of state space S.[5]

□

3.5 Conclusion

In this chapter, we have introduced the basic form of an exception mechanism for the refinement calculus. The **exit** construct was used to distinguish between normal and exceptional termination of a program. Some laws about the interaction of **exit**s with other constructs were given, guided initially by intuition. Finally, a semantic framework was introduced, involving an extension of the standard weakest precondition to take separate postconditions for normal and exceptional termination. Given this framework, the laws proposed earlier were proved correct, including the law for recursion.

[5] In fact, we also need the equivalence

$$(\forall\, i \bullet x_i \sqsubseteq y) \ \equiv \ (\bigsqcup i \bullet x_i) \sqsubseteq y.$$

Chapter 4

Exceptions on a larger scale

The simple exception mechanism introduced in the last chapter is clearly not powerful enough to be used in any serious programming endeavour. However, it is not too difficult to combine the idea of **exits** with the procedure mechanism already in the language, to provide a flexible and powerful exception-handling system. This is the concern of this chapter. After describing how to deal with named **exits** and exception-handling routines, we propose and prove some laws for the new constructs, and show how to use them in a sample development.

The exception-handling system introduced here will be used in Chapter 8 for the development of some more significant programs, which use a library of abstract data types.

4.1 Named exits and handling routines

There are two reasons why we decided to start our investigation of exceptions in the previous chapter with a very simple scheme of exits and exception blocks: the first is that it is obviously better to understand a simple scheme before moving on to consider anything more complex. The second reason concerns the sheer variety of mechanisms for generating exceptions and handling them which are found in programming languages. By taking an abstract view and considering only the contrast between normal and exceptional termination and the interrupted flow of control given by the **exit** construct, we are left with a mechanism which has no bias towards any particular programming language. We can develop this simple scheme in various ways to produce mechanisms that are easy to translate into different programming languages. We will show

one such development in this chapter, and the translation into a programming language will come in Chapter 8.

There are two important features that are missing in the mechanism proposed in the previous chapter: the ability to distinguish between 'different' exceptions, and the ability to associate code fragments with exceptions. In any reasonably-sized system, there will be several ways in which exceptions can be raised. Indeed, within one procedure, it is quite possible that several 'errors' might occur. For example, if we model a bounded map as a partial function from *Keys* to *Values*.

$$map : Key \pfun Value$$
$$\textbf{inv } \#map \leq max \ ,$$

then an attempt to add a new pair to the map might be modelled as follows:

$$\textbf{procedure } add(\textbf{value } k : Key, v : Value) \mathrel{\widehat=}$$
$$map : \left[\begin{pmatrix} k \notin \mathrm{dom}\ map \\ \#map < max \end{pmatrix}, map = map_0 \cup \{k \mapsto v\}\right] \qquad (*)$$

When we make this procedure robust, by specifying the behaviour when the precondition of $(*)$ is not met, we would like to be able to distinguish between the case where there is already a value stored under the given key ($k \in \mathrm{dom}\ map$) and the case where the map has already reached its maximum size ($\#map = max$). Without this distinction, we cannot give useful error messages, for instance.

It can also be useful to associate code fragments with particular exceptions. In the case above, we might just want to give the user an informative error message, or we might want to attempt some sort of 'clean-up' action — if the exception has been raised in the middle of a sequence of operations, some of the operations may need to have their effect reversed in order to restore the system to a reasonable state.

The mechanism we propose involves the use a construct already in the language — procedures. Since procedures are already a method for giving names to program fragments, it seem natural to combine them with exits to give an exception-handling mechanism. In order to declare a handler, we write

$$\textbf{handler } H \mathrel{\widehat=} hhh$$

as an abbreviation for

$$\textbf{procedure } H \mathrel{\widehat=} hhh; \textbf{ exit} \ .$$

In order to distinguish, for the user, the procedures which are exception-handlers from the standard procedures, we write

$$\textbf{raise } H$$

which is simply an abbreviation for a call of procedure H. Thus, when exception H is raised, the associated code *hhh* is executed, and control passes to the

end of the smallest enclosing exception block.[1] Clearly, this scheme can deal with multiple exceptions without any further complication, simply by declaring several handlers. For instance, a user of the *add* procedure given above might declare a handler for *AlreadyThere* (to be executed when *add* is invoked with $k \in \text{dom } map$) and a handler for *Full* (for when $\#map = max$).

Another advantage of the scheme is that it enables the declaration of the handler to be separated from the raising of the exception, even to the extent that they might be the responsibility of different developers. In the case study which inspired this work — the use of a library of abstract data types — it will be seen (in Chapter 8) that many of the library operations have specifications that include the raising of exceptions: the writer of these specifications, and their implementor, can have no idea of what will be the most appropriate action to be taken when a specific exception is raised. It is the developer of the application which uses the library who has this knowledge, and it is his responsibility to declare the handlers for the exception.

4.2 Laws for raise and handler

As usual with new constructs, we propose some laws about the constructs which will be useful when carrying out developments. These laws can be proved correct using the properties of exceptions and procedures. In all of the laws, we assume that the name chosen for a handler is fresh — suitable renaming can be used, if necessary.

The first two laws are simply encapsulations of the definitions above, so that we can refer to them:

Law 4.1 *handler definition*

A declaration

handler $H \mathrel{\widehat{=}} hhh$

is an abbreviation for

procedure $H \mathrel{\widehat{=}} hhh$; **exit** .

Law 4.2 *raise definition*

Raising an exception

raise H

is an abbreviation for

H ,

a call of procedure H.

[1] That is, the smallest exception block enclosing the point at which H is raised, not the point at which it is declared.

We need to show how to introduce handlers into a program. A sequential composition of two **exit**-free programs can be implemented by turning the second half of the composition into an exception-handler:

Law 4.3 *sequential composition and raise*

$$
\begin{array}{l}
aaa;\ bbb \\
= \quad [\![\ \textbf{handler}\ H \mathrel{\widehat{=}} bbb \bullet \\
\qquad aaa;\ \textbf{raise}\ H \\
\quad]\!] \qquad\qquad \textit{provided } aaa \textit{ and } bbb \textit{ are } \textbf{exit}\textit{-free}
\end{array}
$$

Proof

$$
\begin{array}{l}
RHS = [\![\ \textbf{handler}\ H \mathrel{\widehat{=}} bbb \bullet \\
\qquad\qquad aaa;\ \textbf{raise}\ H \\
\qquad]\!] \\
\qquad = \text{"Copy rule, } \textit{raise definition } 4.2\text{"} \\
\qquad\quad [\![\ aaa;\ bbb;\ \textbf{exit}\ ]\!] \\
\qquad = \text{"}\textit{exit ending block } 3.2\text{"} \\
\qquad\quad [\![\ aaa;\ bbb]\!] \\
\qquad = \text{"}\textit{exception-free block } 3.3\text{"} \\
\qquad\quad aaa;\ bbb \\
\qquad = LHS
\end{array}
$$

□

If a program already contains aaa followed by **exit**, we can replace this with **raise** H:

Law 4.4 *introduce handler*

$$
\begin{array}{l}
[\![\ \mathcal{P}(aaa;\ \textbf{exit})\ ]\!] \\
= \quad [\![\ \textbf{handler}\ H \mathrel{\widehat{=}} aaa \bullet \\
\qquad \mathcal{P}(\textbf{raise}\ H) \\
\quad]\!]
\end{array}
$$

The validity of this law follows immediately from the Copy Rule.

A program containing an else construct can be refined to a choice with a **raise** in one branch:

Law 4.5 *introduce handler*

$$
\begin{array}{l}
[\![\ \mathcal{P}(aaa \gg (bbb;\ ccc))\ ]\!] \\
= \quad [\![\ \textbf{handler}\ H \mathrel{\widehat{=}} ccc \bullet \\
\qquad \mathcal{P}(aaa \parallel (bbb;\ \textbf{raise}\ H)) \\
\quad]\!]
\end{array}
$$

In this case, the proof follows from the definition of > and *introduce handler* 4.4.

If *aaa* and *bbb* are **exit**-free, then a nondeterministic choice between them can be implemented by an exception block with *aaa* as one branch of the choice, and **raise** H as the other branch, if the code associated with exception H is *bbb*:

Law 4.6 *introduce handler to choice*

$$
\begin{array}{l}
aaa \,[\!]\, bbb \\
= \quad [\![\textbf{handler } H \mathrel{\widehat{=}} bbb \bullet \\
\qquad aaa \\
\qquad [\!]\, \textbf{raise } H \\
\quad]\!] \qquad\qquad \textit{provided aaa and bbb are } \textbf{exit}\textit{-free}
\end{array}
$$

Proof

$$
\begin{array}{rl}
LHS = & aaa \,[\!]\, bbb \\
= & \text{“}\textit{choice-else } 3.10\text{”} \\
& [\![\, aaa > bbb \,]\!] \\
= & \text{“}\textit{introduce handler } 4.5\text{”} \\
& [\![\textbf{handler } H \mathrel{\widehat{=}} bbb \bullet \\
& \qquad aaa \,[\!]\, \textbf{raise } H \\
&]\!]
\end{array}
$$

□

We can always add more code after a **raise** construct, since it will never be executed.

Law 4.7 *raise-sequential composition*

$$
\begin{array}{l}
\textbf{raise } H \\
= \quad \textbf{raise } H ; aaa
\end{array}
$$

The proof of this is immediate from *program after exit* 3.1, and the definition of **raise**.

4.3 A development using named exits

In order to show some of the rules from the previous section in use, and in order to give a flavour of the developments which are possible using named exceptions

and error-handling routines, we work our way through a small example in this section. We will need some more laws as we progress.

Our task seems initially to be very simple: we have to find the sum of three numbers

$$sum := a + b + c \ .$$

The difficulty occurs because we will not be able to use the built-in addition operation of the programming language, but will have to use the following procedure instead, which in turn uses the type *valid*:

$$valid \ \widehat{=} \ -maxint..maxint$$

$$\begin{array}{l}\textbf{procedure } add \ (\textbf{value } i, j : \mathbb{N}, \textbf{result } s : \mathbb{N}) \ \widehat{=} \\ \quad i, j, i + j \in valid \ \rightarrow \ s := i + j \\ [] \ \left(\begin{array}{l} i, j \in valid \\ i + j \notin valid \end{array}\right) \rightarrow \textbf{raise } Overflow \\ [] \ \left(\begin{array}{l} i \notin valid \vee \\ j \notin valid \end{array}\right) \rightarrow \textbf{raise } BadInput\end{array}$$

The *add* procedure recognises the possibility of two forms of error: if either of the input numbers is not a valid integer (it is not between $-maxint$ and $+maxint$), then the exception *BadInput* is raised. If both inpnts are valid, but their sum is not, then *Overflow* is raised. Only if both inputs and their sum are valid is the result parameter s set to the sum of the inputs.

Given that we are using this procedure, we need to adjust our specification slightly to reflect the fact that we can only achieve our goal if certain conditions are met.

$$Spec \ \widehat{=} \ sum, r : [OK \vee B \vee OV] \ ,$$

where we have the following definitions:

$$OK \ \widehat{=} \ \left(\begin{array}{l} a, b, c \in valid \\ a + b, a + b + c \in valid \\ sum = a + b + c \\ r = Ok \end{array}\right)$$

$$B \ \widehat{=} \ \left(\begin{array}{l} (a \notin valid \vee \\ \ b \notin valid \vee \\ \ c \notin valid) \\ sum = 0 \\ r = Bad \end{array}\right)$$

$$OV \ \widehat{=} \ \left(\begin{array}{l} a, b, c \in valid \\ (a + b \notin valid \vee \\ \ a + b + c \notin valid) \\ sum = maxint \\ r = Over \end{array}\right)$$

If we cannot use *add* to obtain the sum, then we set the return code r accordingly, and set *sum* to either 0 or *maxint*.

While this example might seem a little contrived at first sight, it is not really: when we carry out developments which will eventually use the addition operator of a real machine, we often cheat because we assume that the addition will work correctly. In fact, its behaviour is very similar to that of the *add* procedure above — it can overflow or behave strangely if its inputs are 'out of range'.[2]

The obvious way to implement *Spec* is to use two calls of *add*, with appropriate handlers for *Overflow* and *BadInput*. The target of our development is something like

$$
\begin{array}{l}
[\![\ \textbf{handler}\ \ \mathit{Overflow} \mathrel{\widehat{=}} \mathit{sum} := \mathit{maxint};\ r := \mathit{Over} \\
\qquad\qquad\ \ \mathit{BadInput} \mathrel{\widehat{=}} \mathit{sum} := 0;\ r := \mathit{Bad} \bullet \\
\quad \mathit{add}(a, b, \mathit{sum}); \\
\quad \mathit{add}(\mathit{sum}, c, \mathit{sum}); \\
\quad r := \mathit{Ok} \\
]\!]
\end{array}
$$

The problems now is to manipulate *Spec* until it contains two consecutive copies of *add* with appropriate substitutions. It is not difficult to split OK into a sequential composition, but we also need to distribute the relevant parts of B and OV into the correct parts of tbe composition.

The first additional law that we need, easily proved by *wp* calculation, helps to turn a specification with a postcondition that is a disjunction into an exception block, with branches corresponding to the clauses of the disjunction.

Law 4.8 *disjunction-else distribution*

$$
\begin{array}{l}
w : [\alpha, \beta_1 \vee \ldots \vee \beta_n] \\
\sqsubseteq\ [\![\\
\qquad w : [\alpha, \beta_1] \\
\qquad > \\
\qquad w : [\alpha, \beta_2] \\
\qquad \ldots \\
\qquad > \\
\qquad w : [\alpha, \beta_n] \\
\quad]\!]
\end{array}
$$

Our *Spec* is already in a suitable form to apply this law, but, before we do so, it will be convenient to spht B and OV into two further disjunctions:

$$
B = \left(\begin{array}{l} (a \notin \mathit{valid} \vee \\ \ b \notin \mathit{valid}) \\ \mathit{sum} = 0 \\ r = \mathit{Bad} \end{array} \right) \vee \left(\begin{array}{l} c \notin \mathit{valid} \\ \mathit{sum} = 0 \\ r = \mathit{Bad} \end{array} \right)
$$

[2] Tbis is particularly true when the programming language contains several forms of nnmber: int, longint, float, real etc.

$$OV = \left(\begin{array}{l} a, b, c \in valid \\ a + b \notin valid \\ sum = maxint \\ r = Over \end{array}\right) \vee \left(\begin{array}{l} a, b, c \in valid \\ a + b + c \notin valid \\ sum = maxint \\ r = Over \end{array}\right)$$

We call these disjunctions $B1$ and $B2$ and $OV1$ and $OV2$, respectively.

Now we can apply *disjunction-else distribution* 4.8:

$$\begin{array}{ll} & Spec \\ = & sum, r : [OK \vee (B1 \vee B2) \vee (OV1 \vee OV2)] \\ = & sum, r : [OK \vee B1 \vee OV1 \vee B2 \vee OV2] \\ \sqsubseteq & \text{“}disjunction\text{-}else\ distribution\ 4.8\text{”} \\ & [\![\\ & \quad sum, r : [OK] \\ & \quad > \\ & \quad sum, r : [B1] \\ & \quad > \\ & \quad sum, r : [OV1] \\ & \quad > \\ & \quad sum, r : [B2] \\ & \quad > \\ & \quad sum, r : [OV2] \\ &]\!] \end{array}$$

◁

The next step is to develop the successful branch into a sequential composition:

$$\begin{array}{lll} = & sum, r : & \left[\begin{array}{c} a, b, c \in valid \\ a + b, a + b + c \in valid \\ sum = a + b + c \\ r = Ok \end{array}\right] \\ \sqsubseteq & sum, r : & \left[\begin{array}{c} a, b, c \in valid \\ a + b, a + b + c \in valid \\ sum = a + b \end{array}\right] ; \\ & sum, r : & \left[\begin{array}{cc} \vdots & \\ a, b, c \in valid & a, b, c \in valid \\ a + b, a + b + c \in valid , & a + b, a + b + c \in valid \\ sum = a + b & sum = a + b + c \\ & r = Ok \end{array}\right] \end{array}$$

Let us name these two specification statements $OK1$ and $OK2$.

Now we simply have to associate the error-case branches with the appropriate parts of the successful case. In order to move the branches around, we need these two laws:

Law 4.9 *else distribution*

$$\begin{array}{ll} & (aaa; bbb) > ccc \\ = & (aaa > ccc); bbb \end{array}$$

Law 4.10 *else distribution*

$$\begin{array}{ll} & (aaa;\ bbb) > ccc \\ \sqsubseteq & (aaa > ccc)\ ;\ (bbb > ccc) \qquad\qquad \textit{provided } ccc \sqsubseteq aaa;\ ccc \end{array}$$

Now the inside of the exception block can be refined as follows:

$$\begin{array}{ll} & (OK1;\ OK2) > sum, r : [B1] > sum, r : [OV1] \\ & > sum, r : [B2] > sum, r : [OV2] \\ \sqsubseteq & \text{“}\textit{else distribution}\ 4.9\text{”} \\ & ((OK1 > sum, r : [B1] > sum, r : [OV1]);\ OK2) \\ & > sum, r : [B2] > sum, r : [OV2] \\ \sqsubseteq & \text{“}\textit{else distribution}\ 4.10,\ \textit{take normal branch}\ 3.5(\text{twice})\text{”} \\ & (OK1 > sum, r : [B1] > sum, r : [OV1]); \qquad (1) \\ & (OK2 > sum, r : [B2] > sum, r : [OV2]) \qquad (2) \end{array}$$

The application of *else distribution* 4.10 is justified by the fact that

$$\begin{array}{ll} & sum, r : [B2] \ [\!]\ sum, r : [OV2] \\ \sqsubseteq & OK1;\ (sum, r : [B2] \ [\!]\ sum, r : [OV2])\ . \end{array}$$

take normal branch 3.5 allows us to remove the two extra else-branches that would otherwise appear at the end of (1).

Now we have our program in the shape required, and it is a simple matter to refine (1) and (2) using calls of *add*.

$$\begin{array}{ll} (1) = & sum, r : \left[\begin{array}{c} a, b, c \in valid \\ a + b, a + b + c \in valid \\ sum = a + b \end{array}\right] \\ & [\!]\ sum, r : \left[\begin{array}{c} (a \notin valid \vee \\ b \notin valid) \\ sum = 0 \\ r = Bad \end{array}\right] ; \textbf{exit} \\ & [\!]\ sum, r : \left[\begin{array}{c} a, b, c \in valid \\ a + b \notin valid \\ sum = maxint \\ r = Over \end{array}\right] ; \textbf{exit} \\ \sqsubseteq & \textbf{handler}\ BadInput \mathrel{\widehat{=}} sum := 0;\ r := Bad \\ & Overflow \mathrel{\widehat{=}} sum := maxint;\ r := Over \bullet \\ & \left(\begin{array}{c} a, b \in valid \\ a + b \in valid \end{array}\right) \rightarrow sum := a + b \\ & [\!] \left(\begin{array}{c} (a \notin valid \vee \\ b \notin valid) \end{array}\right) \rightarrow \textbf{raise}\ Badinput \\ & [\!] \left(\begin{array}{c} a, b \in valid \\ a + b \notin valid \end{array}\right) \rightarrow \textbf{raise}\ Overflow \\ \sqsubseteq & add(a, b, sum) \end{array}$$

The development for (2) is very similar, except that the assignment of Ok to r is moved to the end of the exception block:

$$(2) \sqsubseteq \quad add(sum, c, sum); \\ r := Ok$$

We have now developed the program for $sum := a + b + c$. It cannot be denied that it is a somewhat tortuous development: more experience with development involving exceptions and exception-handling is likely to reveal 'development idioms' -- patterns which appear frequently -- which can then be encapsulated into further laws. It is interesting to note that we actually end up with two declarations of the handlers in the above development: since they are identical, they can be merged to simplify the code.

It is also worth remarking on the change we made in our original specification, to allow for the possibility of the *add* procedure 'failing': we started with a specification consisting of a simple assignment to *sum*, and we ended up with a more complicated specification, formed from the three disjuncts. OK, B and OV. Clearly this transformation is not a refinement, since it allows different behaviours within the original precondition. It is in fact what Banach calls a retrenchment [7]. This sort of transformation occurs fairly often in system development: the top level 'specification' captures most of the desired behaviour of the system, but it needs to be transformed until the exact behaviour is captured, in all its gory detail: then the process of formal refinement can start.

4.4 A possible enhancement

One advantage of this procedure-based mechanism for exception-handling is that it can easily he extended to model an additional feature which is found in the exception-handling mechanisms of several programming languages.[3] Languages such as CLU [31] allow the programmer to pass a parameter to an exception handler. Since, as far as we are concerned, exception handlers are just procedures, we can model this with the normal procedure-passing mechanism for procedures. One application of this would be to give more informative error messages — a handler for *RecordNotFound* could say exactly which record could not be located, for instance.

4.5 Conclusion

In this chapter, we extended the simple **exit** mechanism defined previously, by introducing the idea of an exception handler. This encapsulated the actions to

[3]Section 10.1 contains details of some of the exception-handling mechanisms available in programming languages.

be taken when a particular exception was raised. It was defined using the procedure mechanism already found in the language. Several laws were proposed and proved, and sample developments showed how the laws could be used.

Part III

Iterators

Chapter 5

An iterator construct

This chapter begins our study of iterators and how they can be introduced into a development method based on the refinement calculus. The notation used is based partly on recent results from the functional programming community about homomorphisms on recursively-defined data types (see for example [9, 36]). These results are summarised in Section 5.3, after the introduction of a basic iterator construct over sequences. We then look at iterators over more complex recursive types, and at how we can use a combination of recursive and non-recursive types to specify the behaviour of a module. The final part of this chapter shows how some of the results from the functional programming theory can be used.

In subsequent chapters we will develop this work by investigating the use of procedure variables, and thus procedures as parameters. In Chapter 7 we will show how procedure parameters can be used to encapsulate iterators, and so enable them to be specified and used in developments based on a library of pre-defined abstract data types.

5.1 Introduction

Before looking at the functional programming ideas, we motivate the work with a brief introduction to iterators: what they are, why they are important and the history of their use. Eckart [19] has described iteration as 'the ability to consider every element of a data structure'. For instance, a company's personnel database might contain a list of records, each of which contains details of an employee, such as name, address, salary and so on. Given this structure, we

could use an iterator to define a procedure which would examine each record in turn, and reduce the salary of all those who earned more than £50K by 20%. Alternatively, we could obtain a list of all those employees earning more than £30K, or even the total wage bill for the company. These three examples illustrate three common forms of iteration: the first takes the form of an 'update in place'; the second is a filter, producing a smaller collection of records; and the third produces a scalar value.

The key fact about an iterator construct is that its user should need no knowledge of how the structure is implemented: it should be possible to abstract from such details as whether the list is singly or doubly linked and whether it is stored in main or secondary storage. The interesting challenge is that we want users, while working within the refinement calculus, to be able to use iterator constructs on elements of types they have themselves defined, rather than simply on built-in types. We therefore eventually need to pass, as parameters to such constructs, information about the action to be applied to each element of the structure: this motivates our interest in procedural parameters, explored further in Chapters 6 and 7.

Although iterators have appeared in programming languages for many years,[1] there has been some renewed interest recently. Wing [54] notes that two recent trends in technology are likely to cause future interest: persistent object repositories and large-scale distributed information systems. A persistent object repository [1] can be seen as a generalisation of a database: instead of records in a relation, there are objects in collections of different types — often user-defined abstract types. Just as for databases, users want to carry out queries over these collections, which are simply applications of iterators. The situation is slightly different for large-scale distributed information systems such as the World Wide Web, WAIS and gopher: when these systems were introduced, there was no direct support for iterators, so users were forced to follow hypertext links to achieve the effect of a query such as "Find me all the objects that ...". However, there are now several search-engines available, which provide a more user-friendly interface to these iteration-like abstractions.

5.2 The it..ti construct for sequences

Consider an iteration over a sequence s of type seq A, defined by

$$\textbf{type}\ \mathsf{seq}\, A \mathrel{\widehat{=}} \mathsf{Empty} \mid \mathsf{Cons}\, A\, (\mathsf{seq}\, A)\ ,$$

where we make use of the usual refinement calculus notation for disjoint union types: each element of the type is either an empty sequence, or it is constructed from an element of A and another sequence. (Further details may be found in [44, Chapter 15].) For brevity, and the convenience of an infix operator, we will

[1] Section 10.2 contains a review of iterator constructs in several programming languages.

often use the abbreviations

$$\begin{aligned} () &\;\widehat{=}\; \mathsf{Empty} \\ a{:}as &\;\widehat{=}\; \mathsf{Cons}\ a\ as\ . \end{aligned}$$

The purpose of the iteration is to perform some calculation which involves the consideration of each element of the seqnence in turn. We suppose that the result of the iteration is to be stored in a variable r, which is of some type R. A first example of the construct that we propose to use for the iteration is the following:

it s **into** r **with**
$\quad () \longrightarrow r := x$
$\quad [\!] \; a{:}as \longrightarrow r := f(a, as)$
ti

The **it..ti** construct begins with a statement of the variable over which the iteration is to be performed, s, and the variable where the result is to be stored, r. This is followed by a collection of branches, one for each part of the disjoint union definition of the type of s.[2] In this case, the first branch covers the empty sequence, and the second branch covers non-empty sequences made up of an element a, together with a sequence as. The interpretation which we intend for this construct is as follows: if the sequence s is empty, then the result variable r is updated with a (constant) value x; alternatively, if s is not empty, then the new value of r is found by applying a binary function f to the first element of s and to the result of the iteration over the remainder of s — it will be clear from the definition below that this result is determined by a recursion.

This brings us to one of the interesting points about this construct — the dual use of as in the second branch. On the left of the arrow, as is a pattern matcher, while on the right it is the result of a recursive call. Use of this abbreviation has the advantage that we do not have to give a name to the function we are applying to s.

We can now give the definition for the **it..ti** construct in terms of a recursive procedure.

Definition 5.1 *sequence iterator*

An iteration over a sequence s of the following form

it s **into** r **with**
$\quad () \longrightarrow bbb$
$\quad [\!] \; a{:}as \longrightarrow ccc$
ti

is defined as

[2] There are obvious similarities with the tagged alternation and iteration constructs of [44].

$I(s, r)$,

where

procedure I(**value** s, **result** r) $\widehat{=}$
  if s **is**
    $\langle\,\rangle \longrightarrow bbb$
  [] $a{:}as \longrightarrow$ |[**var** $l \bullet I(as, l);\ ccc[as\backslash l]$]|
  fi .

Notice that the local variable l is used to store the result of applying the procedure recursively to as, so that it can be used in the immediately following statement, by means of substitution. Notice that, since the recursive call is applied to the tail of s, we are guaranteed that the recursion will terminate.

For example, suppose we have the following declarations:

s : seq **N**
r : **N**

Then the following program fragment will obtain the sum of the sequence:

it s **into** r **with**
  $\langle\,\rangle \longrightarrow r := 0$
[] $n{:}ns \longrightarrow r := n + ns$
ti

Similarly, we obtain the length:

it s **into** r **with**
  $\langle\,\rangle \longrightarrow r := 0$
[] $n{:}ns \longrightarrow r := 1 + ns$
ti

5.3 Homomorphisms on initial algebras

To generalise iterators beyond sequences, and to expose the links between the two examples above, we use recent work in the functional programming community, which we summarise in this section. The work is based on homomorphisms — functions on recursively-defined data types whose inductive definition mimics the structure of the type. Further details may be found in [9, 36, 39].

First we give an informal indication of the direction of this work, before outlining its formal basis: since we are transferring work from functional programming into the refinement calculus, we give only a brief summary of the results required rather than the full details, which may be found in the papers cited.

An informal approach

Suppose we define a type T by

$$T \mathrel{\widehat{=}} \mathsf{a}\, A \mid \mathsf{b}\, B\, T \mid \mathsf{c}\, T\, C\, T \ .$$

This uses the previously defined types A, B and C and defines the constructor functions a, b and c, whose types are thus

$$\begin{aligned} \mathsf{a} &: A \longrightarrow T \\ \mathsf{b} &: B \longrightarrow T \longrightarrow T \\ \mathsf{c} &: T \longrightarrow C \longrightarrow T \longrightarrow T \ . \end{aligned}$$

Each element of T can be thought of as 'tagged' with a constructor function.

Now, suppose that we wish to define a function on T, which will give as result an element of some type R, say $f : T \longrightarrow R$. We can achieve this by defining three subsidiary functions, each designed to show the effect of f on the disjoint part of its domain corresponding to each of the constructor functions.

The simplest part of the domain is that formed by a: for this we define

$$f_{\mathsf{a}} : A \longrightarrow R$$

which maps every element of A to an element of R. Now to find the effect of f on an element of T of the form $\mathsf{a}\, a$, we merely apply f_{a} to a.

The functions corresponding to b and c are slightly more complex, but the types of their domains are derived from the domains of the constructor functions, with each instance of T replaced by R (since, as the function is recursively applied, all of the elements of T in the lower-level structure have already been transformed to elements of R). So the subsidiary functions we need to define have the following types:

$$\begin{aligned} f_{\mathsf{b}} &: B \longrightarrow R \longrightarrow R \\ f_{\mathsf{c}} &: R \longrightarrow C \longrightarrow R \longrightarrow R \ . \end{aligned}$$

Once we have given the functions f_{a}, f_{b} and f_{c}, we can combine them, using the so-called 'banana brackets' [36], to define a function from T to R:

$$f \mathrel{\widehat{=}} (\!|f_{\mathsf{a}}, f_{\mathsf{b}}, f_{\mathsf{c}}|\!) \ .$$

This function f can be applied to any element of T, however it has been constructed. Moreover, for any element of T, we can be sure that exactly one of the subsidiary functions is applicable.

We will use, as a concrete example running through this section, the type of non-empty lists of natural numbers. This is defined by

$$\mathit{Natlist} \mathrel{\widehat{=}} \mathsf{Single}\, \mathbb{N} \mid \mathsf{Cons}\, \mathbb{N}\, \mathit{Natlist} \ ,$$

using the previously defined type $\mathbb{N}$ and the constructors Single and Cons.

For every function we want to define on *Natlist*, we need to give two subsidiary functions to show the effect on a singleton list, and on a longer list. Thus if we want to map an element of *Natlist* to its sum, we can define the function in two parts:

$$\begin{aligned} sum(\mathsf{Single}\, n) &= n \\ sum(\mathsf{Cons}\, n\, ns) &= n + sum(ns) \ . \end{aligned}$$

So the function *sum* can be defined:

$$sum = (\!|id, (+)|\!) \ .$$

Similarly, to obtain the product of the elements of a list, we define

$$product = (\!|id, (*)|\!) \ .$$

To add one to every element of a list, we define

$$inc\_list = (\!|inc, cons - inc|\!) \ ,$$

where

$$\begin{aligned} inc\, n &= \mathsf{Single}\ n + 1 \\ cons - inc\, n\, ns &= \mathsf{Cons}\, (n+1)\, ns \ . \end{aligned}$$

Note that *inc_list* does not 'update' the list, but rather forms a new list of the desired values.

Applying these functions to the list $(1,2)$ gives

$$\begin{aligned} (\!|id, (+)|\!)\ (1,2) &= 1 + 2 = 3 \\ (\!|id, (*)|\!)\ (1,2) &= 1 * 2 = 2 \\ (\!|inc, cons - inc|\!)\ (1,2) &= (1+1, 2+1) = (2,3) \ . \end{aligned}$$

Formal definitions

It is well-known that recursively-defined types, such as *Natlist* above, can be viewed as initial algebras of an appropriate functor. For instance, if we define the functor F by its action on objects (sets) and functions:

$$\begin{aligned} \mathsf{F}(A) &= \mathbb{N} + (\mathbb{N} \times A) \\ \mathsf{F}(f) &= id + (id \times f) \ , \end{aligned}$$

then

$$\mathsf{F}(Natlist) = \mathbb{N} + (\mathbb{N} \times Natlist) \ .$$

Now the two constructor functions of *Natlist* can be combined into a single function with the join operator

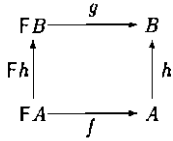
$$[\mathsf{Single}, \mathsf{Cons}] : \mathbb{N} + (\mathbb{N} \times Natlist) \longrightarrow Natlist \ ,$$

and we can deduce that we therefore have an F-algebra, which consists of F*Natlist*, *Natlist* and the function between them [Single, Cons].

For any two F-algebras $f : \mathsf{F}A \longrightarrow A$ and $g : \mathsf{F}B \longrightarrow B$, an F-homomorphism from f to g is an arrow $h : A \longrightarrow B$ snch that

$$h \cdot f = g \cdot \mathsf{F}h \ ,$$

which expresses equationally that the following diagram commutes:



The join [Single, Cons] is actually defined to be the initial algebra of F. It is therefore possible, given any other F-algebra -- say g -- to find a unique F-homomorphism from the initial algebra to g. This concept of the 'unique homomorphism from an initial algebra' is the basis of our iterator construct, and is called a catamorphism [36]. It is usually written with the 'banana brackets' mentioned above: $(\!|g|\!)$. A simple way of thinking about catamorphisms is that the functions given between the brackets $(\!|\ |\!)$ are used as 'replacements' for the constructor functions of the catamorphism's argument -- a form of 'organ transplant'.

In the next section we will make explicit the connection between catamorphisms and the **it..ti** construct, showing how the assignment of a catamorphism applied to an element of a datatype can be refined directly to an iterator. But first we explore the definitions of catamorphisms a little more, showing how they can be defined in two ways, either in functional programming terms, or with a collection of recursive equations.

If we consider again the examples above on *Natlist* we can see two forms of definition. For instance, the *sum* function can be defined, using the subsidiary functions of identity and addition, by

$$sum = (\!|id, (+)|\!) \ .$$

Alternatively, the following equations together define it:

$$\begin{aligned} sum\ (\mathsf{Single}\ n) &= n \\ sum\ (\mathsf{Cons}\ n\ ns) &= n + sum(ns) \ . \end{aligned}$$

Similarly, the *inc_list* function, which adds one to each element of a *Natlist*, is defined either by

$$inc\_list = (\!|inc, cons \cdot inc|\!)$$

or by

$$\begin{aligned} &inc\_list\ (\mathsf{Single}\ n) = \mathsf{Single}\ (n+1) \\ &inc\_list\ (\mathsf{Cons}\ n\ ns) = \mathsf{Cons}\ (n+1)\ inc\_list(ns) \ . \end{aligned}$$

In both of these cases, it is clear that the variant for the recursive definitions is given by structural induction over the datatype: the recursive call on the RHS of the definition is applied to the tail of the original argument.

Of course, it is not difficult to obtain the recursive equations from the functional definition. If a catamorphism c on *Natlist* is defined by

$$c = (\!|f, g|\!) \ ,$$

then the following recursive equations also define c:

$$c\,(\mathsf{Single}\ n) = f\ n$$
$$c\,(\mathsf{Cons}\ n\ ns) = g(n, c(ns)) \ .$$

5.4 Catamorphisms and the it..ti construct

We now give the connection between catamorphisms and the **it..ti** construct which we introduced earlier. From the definition of **it..ti** above in terms of a recursive procedure, we can prove a law which will allow us to carry out developments where we implement an assignment with an iterator. Note that we are now returning to work with the sequence type defined at the start of Section 5.2.

Law 5.2 *assignment iterator*

If the value to be assigned to a variable is formed by the application of a catamorphism to a sequence, then the whole assignment can be implemented with an **it..ti** construct.

$$r := (\!|f, g|\!)\ s$$
$$\sqsubseteq$$
it s **into** r **with**
$\quad \langle\,\rangle \ \longrightarrow\ r := f$
$\quad [\!]\ a{:}as \ \longrightarrow\ r := g(a, as)$
ti

Proof

By the definition of **it..ti**, it is enough to prove that

$$r := (\!|f, g|\!)\ s$$
$$\sqsubseteq$$
$$I(s, r)$$

where, as before,

procedure I(**value** s, **result** r) $\widehat{=}$
$\quad$ **if** s **is**
$\quad\quad \langle\,\rangle \ \longrightarrow\ r := f$
$\quad\quad [\!]\ a{:}as \ \longrightarrow\ [\![\ \mathbf{var}\ l \bullet I(as, l);\ r := g(a, l)\ ]\!]$
$\quad$ **fi** .

We therefore develop the assignment until it is transformed into a recursive procedure:[3]

$r := (\!|f, g|\!)\ s$

$\sqsubseteq$ **re** I(**value** s,**result** r) **variant** V **is** $\#s$ •
$\{ V = \#s \}$
$r := (\!|f, g|\!)\ s$

$\sqsubseteq$ $r : [V = \#s, r = F(s)]$

$\sqsubseteq$ "tagged alternation"
if s **is**
$\quad \langle\,\rangle \ \longrightarrow\ r : [s = \langle\,\rangle \wedge V = \#s, r = F(s)]$ ◁
$[\!]\ a{:}as \ \longrightarrow\ r : [s = a{:}as \wedge V = \#s, r = F(s)]$ (1)
fi

$\sqsubseteq$ "by definition of F, and conversion to recursive form"
$r := f$

(1) $\sqsubseteq$ "by definition of F, and conversion to recursive form"
$r : [s = a{:}as \wedge V = \#s\ ,\ r = g(a, F(as))]$

$\sqsubseteq$ **var** l •
$l : [s = a{:}as \wedge V = \#s\ ,\ l = F(as) \wedge s = a{:}as \wedge V = \#s]$; (2)
$r : [s = a{:}as \wedge V = \#s \wedge l = F(as)\ ,\ r = g(a, F(as))]$ ◁

$\sqsubseteq$ $r := g(a, l)$

(2) $\sqsubseteq$ $l : [V > \#as \geq 0\ ,\ l = F(as)]$

$\sqsubseteq$ $I(as, l)$

□

We can use this law to give some very simple examples of iterations over sequences. Suppose we have the following declarations:

s : seq $\mathbb{N}$
r : $\mathbb{N}$.

Then we can develop simple iterators by referring directly to the catamorphism-style definitions of the functions concerned. For example, for the sum of a sequence:

$r := \Sigma s$
$=$ $r := (\!|0, (+)|\!)\ s$
$\sqsubseteq$ "*assignment iterator* 5.2"
it s **into** r **with**
$\quad \langle\,\rangle \ \longrightarrow\ r := 0$
$[\!]\ n{:}ns \ \longrightarrow\ r := n + ns$
ti

[3]Note that we are using the refinement rule for recursion from the second edition of Morgan's text [44], rather than the original formulation — from the first edition [43] — which was used in the description of recursion with exceptions in Chapter 3.

What we are doing here is to import refinements like

$$\begin{array}{ll} & r := \Sigma s \\ = & r := (\!|0, (+)|\!) s \end{array}$$

directly from other theories: we are able to use the literature on catamorphisms to simplify the development of our own iterators.

The length of a sequence can also be obtained with an iterator:

$$\begin{array}{ll} & r := \#s \\ = & \text{"define } a \oplus b = 1 + b\text{"} \\ & r := (\!|0, \oplus|\!) s \\ \sqsubseteq & \text{"}\textit{assignment iterator}\ 5.2\text{"} \\ & \textbf{it } s \textbf{ into } r \textbf{ with} \\ & \quad ( ) \ \longrightarrow\ r := 0 \\ & \quad [\!] \ n{:}ns \ \longrightarrow\ r := n \oplus ns \\ & \textbf{ti} \\ \sqsubseteq & r := 1 + ns \end{array}$$

◁

5.5 Iterators over more general data types

We now work in a more general framework, with an arbitrary recursive data type. The type we use is defined schematically by

$$\textbf{type } T \mathrel{\widehat{=}} \mathsf{a} \mid \mathsf{b}\ X \mid \mathsf{c}\ Y\ T\ .$$

Thus an element of T is either a constant, identified by a, or it is the image of an element of some set X, tagged by b, or it is formed from an element of Y and some other element of T and is tagged by c. It will become clear how the definitions and refinement laws can be extended from a type with these three 'typical' branches to any other recursive type.

First we extend Definition *sequence iterator* 5.1.

Definition 5.3 *general iterator*

If t is any element of the type T defined above, then

$$\begin{array}{l} \textbf{it } t \textbf{ into } r \textbf{ with} \\ \quad \mathsf{a} \ \longrightarrow\ aaa \\ [\!] \ \mathsf{b}\, x \ \longrightarrow\ bbb \\ [\!] \ \mathsf{c}\, y\, t' \ \longrightarrow\ ccc \\ \textbf{ti} \end{array}$$

is defined to mean the same as

$$I(t, r)\ ,$$

where

$$
\begin{array}{l}
\textbf{procedure } I(\textbf{value } t, \textbf{result } r) \mathrel{\widehat{=}} \\
\quad \textbf{if } t \textbf{ is} \\
\qquad \begin{array}{ll} \mathsf{a} & \longrightarrow aaa \\ [\!] \, \mathsf{b}\, x & \longrightarrow bbb \\ [\!] \, \mathsf{c}\, y\, t' & \longrightarrow |[\ \textbf{var } l \bullet I(t', l);\ ccc[t'\backslash l]\ ]| \end{array} \\
\quad \textbf{fi}
\end{array}
$$

Notice that the local variable is only needed on the third branch, where there is a recursive occurrence of T in the type definition. If there were several occurrences within one branch, then the same number of local variables would be required: so a branch

$$\mathsf{m}\ T\ T$$

would correspond to a branch in the definition of the procedure I which had the form

$$\mathsf{m}\ t_1\ t_2 \longrightarrow |[\ \textbf{var } l_1, l_2 \bullet I(t_1, l_1);\ I(t_2, l_2);\ ccc[t_1, t_2\backslash l_1, l_2]\ ]| \ .$$

Having extended the definition of an iterator itself, we can also extend the law which introdnces an iterator as a refinement of an assignment:

Law 5.4 *assignment iterator*

$$
\begin{array}{l}
\quad r := (\!|P, Q, R|\!)\ t \\
\sqsubseteq \\
\quad \textbf{it } t \textbf{ into } r \textbf{ with} \\
\qquad \begin{array}{ll} \mathsf{a} & \longrightarrow r := P \\ [\!] \, \mathsf{b}\, x & \longrightarrow r := Q(x) \\ [\!] \, \mathsf{c}\, y\, t' & \longrightarrow r := R(y, t') \end{array} \\
\quad \textbf{ti}
\end{array}
$$

5.6 Refinement of branches

Now that we have these laws over more complicated **data types**, we can use them to develop some more sophisticated examples. The observant reader may be wondering about the point of having "$r :=$" in each branch of the **it..ti** construct. In fact, it is useful to have a program fragment (rather than an expression) in each branch, because it gives scope for further refinement: in cases where the expression being assigned to the result variable — for instance, $R(y, t')$ in the **it..ti** construct in Law *assignment iterator* 5.4 — cannot be easily evaluated in the target language, the assignment $r := R(y, t')$ can be refined until it is code. (In program developments where we are using libraries of abstract data types, it is also likely that we will want to refine branches until they can be replaced by calls of library procedures.)

We give two examples which show the idea of this refinement of branches. The first is based on an example in a paper about iterators in the CLU programming language [32]. The task is to count how many numeric characters are contained in a string which might also contain alphabetic characters.

We need to define two disjoint union types for characters, which are either alphabetic or numeric, and for strings, which are either empty, or contain a character and a string:

type
Char $\widehat{=}$ alph *Alpha* | num *Numeric*
String $\widehat{=}$ empty | ch *Char String*

(We assume that *Alpha* and *Numeric* have been suitably defined.)

We define first an infix operator $\oplus$ which will form part of the catamorphism:

$$(\mathsf{alph}\ c) \oplus n = n$$
$$(\mathsf{num}\ c) \oplus n = n + 1\ .$$

Now it is clear that, if *count_num* is the function which, when applied to a string, returns the desired number of numeric characters, then

$$count\_num = (\!|0, \oplus|\!)\ .$$

We can therefore immediately introduce an iterator, as follows:

$$i := (\!|0, \oplus|\!)\ s$$
$$\sqsubseteq \quad \text{"}assignment\ iterator\ 5.4\text{"}$$
it s **into** i **with**
empty $\longrightarrow$ $i := 0$
$[\!]$ ch $c\ cs$ $\longrightarrow$ $i := c \oplus es$ $\lhd$
ti

Since the expression $c \oplus cs$ is not immediately implementable, we need to refine the second branch, which is not difficult using a tagged alternation:

$\sqsubseteq$ "tagged alternation"
if c **is** num $\longrightarrow$
$i : [c = \mathsf{num}\ n, i = c \oplus cs]$ (1)
$[\!]$ c **is** alph $\longrightarrow$
$i : [c = \mathsf{alph}\ a, i = c \oplus cs]$ (2)
fi

The two branches of the tagged alternation are easily implemented, using the definition of $\oplus$:

(1) $\sqsubseteq$ $i := cs + 1$

(2) $\sqsubseteq$ $i := cs$.

This completes the development, giving overall:

$$
\begin{array}{ll}
 & i := count\_num(s) \\
\sqsubseteq & \textbf{it } s \textbf{ into } i \textbf{ with} \\
 & \quad \mathsf{empty} \longrightarrow i := 0 \\
 & \quad [\!] \ \mathsf{ch}\ c\ cs \longrightarrow \textbf{if } c \textbf{ is num} \longrightarrow i := cs + 1 \\
 & \qquad\qquad\qquad\qquad [\!] \ c \textbf{ is alph} \longrightarrow i := cs \\
 & \qquad\qquad\qquad\qquad \textbf{fi} \\
 & \textbf{ti}
\end{array}
$$

Our second example is concerned with the specification of part of a file system. One of the components of this system is a record of the last time a file was accessed. This access list is modelled as a mapping from file *Names* to *Dates*:

$$\textbf{type } Map[Index, Value] \mathrel{\hat{=}} \mathsf{empty} \mid \mathsf{pr}\ Index\ Value\ Map[Index, Value]$$

$$al : Map[Name, Date]$$

Periodically, it is required to produce, from this access list, two other lists, one of which is to contain all those files that were last accessed strictly before some given date, and the other is to contain the remaining files which have been accessed more recently. For convenience, the access dates are to be retained in both lists.

We can define two functions, *keep* and *reject*, which, when applied to a date and the access list, will return the required lists:

$$keep : Date \longrightarrow Map[Name, Date] \longrightarrow Map[Name, Date]$$

$$keep\ dt\ \mathsf{empty} = \mathsf{empty}$$
$$keep\ dt\ (\mathsf{pr}\ n\ d\ m) = \begin{cases} keep\ dt\ m & \text{if } dt \geq d \\ \mathsf{pr}\ n\ d\ (keep\ dt\ m) & \text{if } dt < d \end{cases}$$

$$reject : Date \longrightarrow Map[Name, Date] \longrightarrow Map[Name, Date]$$

$$reject\ dt\ \mathsf{empty} = \mathsf{empty}$$
$$reject\ dt\ (\mathsf{pr}\ n\ d\ m) = \begin{cases} reject\ dt\ m & \text{if } dt < d \\ \mathsf{pr}\ n\ d\ (reject\ dt\ m) & \text{if } dt \geq d \end{cases}$$

Assuming that s and t are the variables in which the results are to be stored, and that dt is the date about which the access list is being divided, our specification is:

$$s, t := keep\ dt\ al, reject\ dt\ al \ . \tag{1}$$

The simplest way to implement this, using the theory that we have already developed, is to divide the multiple assignment into two simple assignments

joined by sequential composition, and to implement each separately with an **it..ti**. This gives the following development:

$$
\begin{array}{ll}
(1)\sqsubseteq & \textbf{it } al \textbf{ into } s \textbf{ with} \\
& \quad \mathsf{empty} \longrightarrow s := \mathsf{empty} \\
& [\!]\ \mathsf{pr}\ n\ d\ m \longrightarrow \textbf{if } dt \geq d \longrightarrow s := m \\
& \qquad\qquad\qquad\quad [\!]\ dt < d \longrightarrow s := \mathsf{pr}\ n\ d\ m \\
& \qquad\qquad\qquad\quad \textbf{fi} \\
& \textbf{ti}; \\
& \textbf{it } al \textbf{ into } t \textbf{ with} \\
& \quad \mathsf{empty} \longrightarrow t := \mathsf{empty} \\
& [\!]\ \mathsf{pr}\ n\ d\ m \longrightarrow \textbf{if } dt < d \longrightarrow t := m \\
& \qquad\qquad\qquad\quad [\!]\ dt \geq d \longrightarrow t := \mathsf{pr}\ n\ d\ m \\
& \qquad\qquad\qquad\quad \textbf{fi} \\
& \textbf{ti}
\end{array}
$$

However, a much more interesting development is obtained by re-expressing the problem in functional programming terms, and using results developed by that community. Functional programmers would immediately recognise both *keep* and *reject* as examples of the *filter* function, which is defined on lists by

$$
\begin{aligned}
&\mathit{filter}\ p\ (\,) = (\,) \\
&\mathit{filter}\ p\ (x{:}xs) = \begin{cases} x{:}(\mathit{filter}\ p\ xs) & \text{if } p\ x \\ \mathit{filter}\ p\ xs & \text{if } \neg\ p\ x \end{cases} .
\end{aligned}
$$

Now if we define an infix operator $\oplus_p$, a form of 'conditional cons', by

$$
x \oplus_p xs = \begin{array}{ll} x{:}xs & \text{if } p\ x \\ xs & \text{if } \neg\ p\ x \end{array} ,
$$

we can immediately express *filter* as a catamorphism:

$$
\mathit{filter}\ p = (\!|(\,), \oplus_p|\!) \ .
$$

The final function that needs to be defined is one which divides a list into two halves, depending on some filtering predicate p:

$$
\mathit{split}\ p\ xs = (\mathit{filter}\ p\ xs, \mathit{filter}\ \bar{p}\ xs) \ ,
$$

where $\bar{p}$ is the negation of p. It is now easy to see that our original problem can be expressed as

$$
(s, t) := \mathit{split}\ p\ al \tag{2}
$$

where the predicate p is defined (on pairs of names and dates) by

$$
p(n, d) \mathrel{\widehat{=}} dt < d \ .
$$

Now we have defined *split* in terms of *filter*, and *filter* itself has been expressed as a catamorphism, but we cannot yet express *split* directly as a catamorphism,

which we will need if we are to implement (2) with a single iteration. Using the version of *filter* for the *Map* type, we know that

$$\begin{aligned} \mathit{split}\ p\ xs = {} & (\mathit{filter}\ p\ xs, \mathit{filter}\ \bar{p}\ xs) \\ & (([\mathsf{empty}, \oplus_p])\ xs, ([\mathsf{empty}, \oplus_{\bar{p}}])\ xs)\ . \end{aligned}$$

We can express this, in a point-free way, as

$$\mathit{split}\ p = ([\mathsf{empty}, \oplus_p]) \vartriangle ([\mathsf{empty}, \oplus_{\bar{p}}])$$

where $\vartriangle$ is the join operator defined by

$$(f \vartriangle g)\, x = (f\, x, g\, x)\ .$$

Now we can appeal to the functional programming literature for the result that we need. Specifically, in [10, Section 3.2] and [11, Section 3.1], we find the so-called banana-split law:

$$([h]) \vartriangle ([k]) = ([(h \times k) \cdot \mathit{unzip}])$$

where *unzip* is defined in terms of the functor F from the algebra which underlies the catamorphism, and two projection functions:

$$\mathit{unzip} = \mathsf{F}\pi_1 \vartriangle \mathsf{F}\pi_2\ .$$

Now the functor for the *Map* type is very similar to that for *Natlist* given above. Its effect on objects and functions is as follows:

$$\begin{aligned} \mathsf{F}_A(B) &= 1 + (A \times B) \\ \mathsf{F}_A(f) &= id_1 + (id_A \times f) \end{aligned}$$

In our case the parameter A is *Index* $\times$ *Value*, so

$$\mathsf{F}\pi_i = id_1 + (id_{I \times V} \times \pi_i) \qquad \text{for } i = 1, 2\ .$$

Now a little algebraic manipulation, in the Squigol fashion, allows us to express *split p* as a catamorphism:

$$\begin{aligned} \mathit{split}\ p &= ([\mathsf{empty}, \oplus_p]) \vartriangle ([\mathsf{empty}, \oplus_{\bar{p}}]) \\ &= \text{“by banana-split law”} \\ &\quad ([([\mathsf{empty}, \oplus_p]) \times ([\mathsf{empty}, \oplus_{\bar{p}}]) \cdot (\mathsf{F}\pi_1 \vartriangle \mathsf{F}\pi_2)]) \\ &= \text{“since } (h \times k) \cdot (l \vartriangle m) = h \cdot l \vartriangle k \cdot m\text{”} \\ &\quad ([([\mathsf{empty}, \oplus_p]) \cdot (id_1 + (id_{I \times V} \times \pi_1)) \vartriangle \\ &\qquad ([\mathsf{empty}, \oplus_{\bar{p}}]) \cdot (id_1 + (id_{I \times V} \times \pi_2))]) \\ &= \text{“since } [f, g] \cdot (h + k) = [f \cdot h, g \cdot k]\text{”} \\ &\quad ([([\mathsf{empty}, \oplus_p \cdot (id_{I \times V} \times \pi_1)] \vartriangle [\mathsf{empty}, \oplus_{\bar{p}} \cdot (id_{I \times V} \times \pi_2)])]) \\ &= \text{“since } [f, g] \vartriangle [h, k] = [f \vartriangle h, g \vartriangle k]\text{”} \\ &\quad ([\mathsf{empty} \vartriangle \mathsf{empty}, \oplus_p \cdot (id_{I \times V} \times \pi_1) \vartriangle \oplus_{\bar{p}} \cdot (id_{I \times V} \times \pi_2)]) \end{aligned}$$

If we examine the second function in this catamorphism more closely, we can see that each part of the join expects to be applied to a pair of pairs, the first

of which is an $(index, value)$ pair, and the second is a pair of maps — the result of the recursive application of the catamorphism to the remainder of the map. The full iterator development is now given by:

$$
\begin{array}{ll}
 & s, t := split\ p\ al \\
\sqsubseteq & s, t := (\!|\mathsf{empty} \vartriangle \mathsf{empty}, \oplus_p \cdot (id_{I \times V} \times \pi_1) \vartriangle \oplus_{\bar{p}} \cdot (id_{I \times V} \times \pi_2)|\!)\ al \\
\sqsubseteq & \mathbf{it}\ al\ \mathbf{into}\ s, t\ \mathbf{with} \\
 & \quad \mathsf{empty} \longrightarrow s, t := \mathsf{empty}, \mathsf{empty} \\
 & \quad [\!]\ \mathsf{pr}\ n\ d\ m \longrightarrow s, t := (n, d) \oplus_p m_1, (n, d) \oplus_{\bar{p}} m_2 \\
 & \mathbf{ti}
\end{array}
\qquad \triangleleft
$$

where m_1 and m_2 are the first and second components of the pair m.

The second branch can be developed in the obvious way with an alternation:

$$
\begin{array}{ll}
\sqsubseteq & \mathbf{if}\ p(n, d) \longrightarrow s := \mathsf{pr}\ n\ d\ m_1; t := m_2 \\
 & \quad [\!]\ \neg\, p(n, d) \longrightarrow t := \mathsf{pr}\ n\ d\ m_2; s := m_1 \\
 & \mathbf{fi}
\end{array}
$$

5.7 Conclusion

In this chapter, we have introduced the iterator construct **it..ti**, which forms the basis of our work on iterators. The construct was based on the idea of a catamorphism, but was formally defined as a recursive procedure. Several examples were given, showing how particular functions can be seen as catamorphisms, and therefore implemented by an **it..ti** construct.

In Section 5.5, we explained how the **it..ti** construct for sequences could be extended to act on a more general data type. The type T used there is intended to he a typical example of a type generated by a polynomial functor. This form of functor — formed from constants, products and coproducts — is general enough for our purposes, and we are guaranteed the existence of an initial object in the category of F-algebras (see [38]).

At the start of the chapter, we gave illustrations of three common forms of iteration: the 'update in place', the filter and the scalar result. During the chapter. we showed how the second and third of these are related to the new construct: the file-system example in the last section was an example of a filter, while the sum or the length of a sequence was an example of a scalar result. The first form of iteration was not explicitly illustrated, but it is not hard to see that following an **it..ti** by an assignment of the result back to the original variable could have the desired effect. For instance, in Section 5.3, we introdnced the *inc_list* catamorphism:

$$inc\_list = (\!|inc, cons \leftarrow inc|\!)\ ,$$

where *inc* simply forms a singleton list from tbe increment of its argument. Suppose that we had to implement the following 'update in place' form of

iteration:

$$s := inc\_list(s) \ ,$$

adding one to each element of the non-empty list of numbers s. It is easy to see that this is achieved by the following program:

```
var r : Natlist •
it s into r with
     Single n    → r := Single n + 1
  [] Cons n ns   → r := Cons (n + 1) ns
ti;
s := r
```

Chapter 6

Higher-order programs

We now extend the language of the refinement calculus to cover procedure variables, explaining first the syntax of the new constructs, then their predicate transformer semantics. The semantics is given by showing how the normal Copy Rule semantics for procedure constants can be replaced by an equivalent formulation, which involves considering procedure meanings as values. We then show how procedure variables can take these values, and give some refinement laws.

The basis of this semantic definition is Naumann's work, reported in [48], but the well-formedness proof and all of the laws here are original.

The ideas developed here will be used in the next chapter to allow us to define an encapsulated iterator procedure, which will need to take procedure values as parameters.

In this chapter, unless otherwise stated, pv will be used to represent a procedure variable, and pe an expression of procedure type. We will also use semantic brackets $[\![\ ]\!]$.

6.1 Syntax

The only parametrisation mechanisms allowed are **value** and **result**. We allow procedure types to be named. For example:

type
binproc = **proc** (**value** $a, b : \mathbb{N}$)
comp = **proc** (**value** $x, y : \mathbb{N}$, **result** b : *Boolean*)

We can declare variables of procedure type in the normal way, using named types or explicit type expressions:

var
p : *binproc*
q : **proc** (**value** $x, y : \mathbb{N}$, **result** b : *Boolean*)

Although the parameters of a procedure type are specified, it is also possible for the bodies of procedures to refer directly to external (global) variables. These do not have to be specified and this is one of the major technical complications in the semantics. However, in languages where procedure variables are not allowed to refer to global variables, the expressive power of procedure variables is limited.

Procedure constants and variables may be executed, using the keyword **call**:

call q .

Actual parameters are supplied for the formal parameters, as usual.

When we consider the program fragments which make up procedure expressions, we see the first significant syntactic restriction: such expressions must always be parametrised. Any variable not mentioned in the expression's parameter list must be declared globally. Thus, unless x and y are global variables, we are not allowed to assign the procedure value $x := y + 1$ to a procedure variable; instead we have to use the value (**value** $y : \mathbb{N}$, **result** $x : \mathbb{N} \bullet x := y + 1$). Once again, any external variables are not explicitly mentioned.

Naumann gives two further restrictions on the syntax of the language containing procedure variables, which are included at the end of this chapter for completeness.

Having declared procedure variables, we can assign values to them:

$q := pe$

where pe is an expression — parametrised as necessary — of the correct procedure type. However, assignment to procedure variables causes some interesting problems — this is investigated further below.

6.2 Semantics

Notation

Before giving the predicate transformer semantics of procedure variables, we need to define some notation.

Since keeping track of the state spaces on which the predicate transformers are acting is one of the key parts of these semantics, we need notation which allows us to restrict and extend the states. Suppose σ is a state, that is a function from variable names to values. Then $\sigma \setminus x$ is the restriction of σ to all variables except x (which may be a list). So, if y is a state component — in the domain of σ — and distinct from x, then $(\sigma \setminus x).y = \sigma.y$.

We also need the inverse image function $\dagger x$ of restriction. Suppose ϕ is a predicate over state space Σ: for now, we can regard ϕ as any member of $\mathbf{P}\,\Sigma$. If x (of type T) is not a component of the state, then $\phi \dagger x$ is a predicate over the state space extended by x. It is therefore a set of states — a subset of $\Sigma \times T$ — and is defined by

$$\sigma \in \phi \dagger x \;\equiv\; \sigma \setminus x \in \phi \ .$$

The extension of state space Σ by the fresh variable $x : T$ is denoted by $\Sigma, x : T$. So $\phi \dagger x$ is a predicate over $\Sigma, x : T$.

The final piece of notation required is also concerned with the state spaces over which predicate transformers act. Various authors [5, 47] have shown that the product of two predicate transformers can be used to model their combined action: in the case, as here, where we simply want to extend the state space of a predicate transformer, we can take its product with the identity transformer on the additional components. If f is a predicate transformer over state space Σ, and x is not a component of Σ, then we define $f \otimes id_x$ (over $\Sigma, x : T$) by its action on predicates ϕ over $\Sigma, x : T$:

$$\begin{array}{l} \sigma \in (f \otimes id_x).\phi \\ \equiv \\ (\exists \psi \mid (\forall \tau \mid \tau \in \psi \bullet \tau[x \mapsto \sigma.x] \in \phi) \bullet \sigma \setminus x \in f.\psi) \end{array}$$

where ψ ranges over predicates over Σ, and $\tau[x \mapsto \sigma.x]$ denotes overriding — agreeing with τ except at x where it takes value $\sigma.x$. In other words, the weakest condition for $f \otimes id_x$ to establish ϕ from some initial state σ is that f should establish a condition ψ on Σ from the relevant part $\sigma \setminus x$ of σ, and that every state τ in ψ should satisfy ϕ, when joined with the unchanging component x. Alternatively, τ is the largest postcondition which when extended at x with $\sigma.x$ lies within ϕ.

We note in passing that for predicates (over $\Sigma, x : T$) which are independent of x — that is of the form $\alpha \dagger x$ — we have from above that

$$(f \otimes id_x).(\alpha \dagger x) \;=\; (f.\alpha) \dagger x \ ,$$

and hence

$$f \otimes id_z \sqsubseteq g \otimes id_z$$

implies

$$f \sqsubseteq g \ .$$

(The other direction follows directly from monotonicity.) This result will enable us to prove the transitivity of the order relation on procedure values.

Procedures as values

The first step in defining the semantics of procedure variables is to define the values which such variables may take — we need to determine the set which corresponds to a procedure type, say **proc** (**value** $v : V$, **result** $r : R$). We take this to be the set of tuples $\langle f, v, r, g \rangle$ where f is a predicate transformer over $(v : V, r : R, g : G)$, g is a list of the global variables of f, v, r and g are disjoint and the rank[1] of G is at most that of V and R -- this restriction is needed to be sure that the set is properly defined in well-founded set theory. (A procedure value with no parameters is therefore not allowed to refer to any global variables of procedure type.) Although it is uuusual to see the names of the parameters in the values. they are needed in the semantic definitions which come later, and their effect is reduced by the definition of type equivalence for procedure types:

Definition 6.1 *procedure type equivalence*

> We extend the normal rules about type equivalence by explaining when two procedure types are type equivalent: types **proc** (**value** $v : V$, **result** $r : R$) and **proc** (**value** $v' : V'$, **result** $r' : R'$) are equivalent (written $\equiv$) exactly when $V \equiv V'$ and $R \equiv R'$. In other words, the parameter names are not significant, and neither are the global variables.

Having defined the set of values corresponding to a procedure type, we can now give the order relation on it, which we represent by $\dot{\sqsubseteq}$. Basically, this is just the refinement order on the predicate transformers, but we need to be careful about the state spaces involved. In the case where both values have the same formal parameters, we have

$$\begin{array}{l} \langle f, v, r, g \rangle \ \dot{\sqsubseteq} \ \langle f', v, r, h \rangle \\ \text{iff} \\ (f \otimes id_h) \sqsubseteq (f' \otimes id_g) \ , \end{array}$$

provided g is distinct from h. Since f is a predicate transformer over v, r and g, and f' acts over v, r and h, we are extending each predicate transformer to act

[1] The rank of a type is the maximum level of nesting of the procedure constructor.

on the same state space, and saying that the two procedure values are related when these extensions of their predicate transformer components are related. It is easy to see that the transitivity of $\dot{\sqsubseteq}$ follows immediately from the result above.

The definition is only slightly more complicated when there is an overlap between the global variables of the two procedure values, say $P1$ and $P2$. The globals are then partitioned into three lists, g, h and i. These are respectively those that appear only in $P1$, those that appear only in $P2$, and those that appear in both. The refinement ordering is defined as follows:

$$\begin{array}{l}(f, v, r, (g, i)) \mathrel{\dot{\sqsubseteq}} (f', v, r, (h, i)) \\ \text{iff} \\ (f \otimes id_h) \sqsubseteq (f' \otimes id_g) \ .\end{array}$$

Again, the predicate transformers are extended to a common state space.

We omit the case where renaming of parameters is required.

Procedure calls

We can now proceed to give the meaning of procedure calls. We deal with calls of procedure constants, procedure variables and explicit procedure expressions together, since the mechanisms for giving their meanings are very similar, the only distinction being where the value is stored. For procedure constants, it is stored in the euvironment; for procedure variables, it is stored in the state; and for explicit expressions, it does not need to be stored at all. For constants and explicit expressions, the rather formidable formula given below is equivalent to the standard semantics as given by the Copy Rule (with the addition of parameters).

The motivation for the definition of **call** $P(e, w)$ is obtained by considering the standard result for procedure constants about the replacement of **value** and **result** parameters by local variables, with assignments to those variables before and after execution of the procedure body:

$$[\![\textbf{call } P(e, w)]\!] = [\![\textbf{var } v, r \bullet v := e; f; w := r]\!]$$

where P is a procedure constant with associated body f which has formal parameters v and r, and e and w contain no occurrences of v or r. As is usual in such cases, we elide all mentions of the environment η, where it is of no real importance: both sides of the above equation should really be parametrised by η and the expression e should be evaluated in η in the derivation below.

An almost-standard application of refinement calculus laws shows

$$
\begin{array}{ll}
 & wp([\![\textbf{var } v, r \bullet v := e; f; w := r]\!], \phi) \\
= & \text{"local variable law: see below"} \\
 & \forall v, r \bullet wp(v := e; f; w := r, (\phi \dagger v, r)) \\
= & \text{"sequential composition, assignment"} \\
 & \forall v, r \bullet wp(f; w := r, (\phi \dagger v, r))[v \backslash e] \\
= & \text{"sequential composition, assignment"} \\
 & \forall v, r \bullet wp(f, (\phi \dagger v, r))[w \backslash r])[v \backslash e] \ .
\end{array}
$$

In order to be houest about the states ou which f acts, we have had to change the *local variable* law slightly to be sure that the predicate on which the transformer acts is of the correct 'type':

Law 6.2 *introducc local variable*

$$
\begin{array}{l}
wp(\textbf{var } x \bullet aaa, \phi) \\
\quad = \forall x \bullet wp(aaa, \phi \dagger x) \\
\textit{provided } \phi \textit{ contains no } x
\end{array}
$$

This distinction is not usually needed in the presentation of the local variable law. (A similar adjustment needs to be made to the *introduce local constant* law.)

Now we consider a statement **call** $P(e, w)$, where P might be a procedure constant, a procedure variable or an explicit procedure expression. Wherever it may be stored, the meaning of P is a procedure value, say

$$\langle f, v, r, g \rangle \ .$$

At the point of call, the state must contain g (the global variables of P) and w (the actual result parameters). We suppose that the remainder of the state is given by a list t. (v and r must not appear in t.) Then the meaning of the call is given by its effect on a predicate ϕ, as calculated above:

$$(\forall v, r \bullet ((f \otimes id_{w,t}).(\phi \dagger v, r)[w \backslash r])[v \backslash e])$$

We shall call this formula Φ, so that we will be able to refer to it later. In order to justify it, we note that f is a predicate transformer over v, r and g, and therefore $f \otimes id_{w,t}$ is a predicate transformer over v, r, g, w and t. ϕ is a predicate over the complete state space — g, w and t — and so $(\phi \dagger v, r)[w \backslash r]$ is also a predicate over g, w, t, v and r. Thus Φ is a predicate over g, w and t as required.

We can now use this formula for **call** $P(e, w)$ in the definitions of the three different forms of procedure call mentioned above. The meaning of a procedure constant is stored in the environment η, and refers to global variables at the point of declaration.

Definition 6.3 *procedure constant call*

$$
\begin{array}{l}
[\![\textbf{call } P(e,w)]\!]_\eta.\phi \\
= \\
\exists f, v, r, g \bullet \\
\quad \eta.P = (f, v, r, g) \wedge \\
\quad \Phi
\end{array}
$$

It is not hard to see that this definition of **call** $P(e,w)$ has the same effect as the traditional Copy Rule for procedure constants.

For an explicit procedure expression there is no need to store the value at all, and references to global variables refer to the point of use. Here, the environment η has been omitted.

Definition 6.4 *explicit procedure expression call*

$$
\begin{array}{l}
[\![\textbf{call } (\textbf{value } v, \textbf{result } r \bullet p)(e,w)]\!].\phi \\
= \\
\exists f, g \bullet \\
\quad [\![p]\!] = (f, v, r, g) \wedge \\
\quad \Phi
\end{array}
$$

Finally, the most interesting case is that of a call of a procedure variable. Here the value is stored in the state σ.

Definition 6.5 *procedure variable call*

$$
\begin{array}{l}
\sigma \in [\![\textbf{call } pv(e,w)]\!].\phi \\
= \\
\exists f, v, r, g \bullet \\
\quad \sigma.pv = (f, v, r, g) \wedge \\
\quad \sigma \in \Phi
\end{array}
$$

In the case of procedure variables, we note that t must contain pv itself: the constraint on rank given above means that pv cannot appear in g. We also note that f, g and t all depend on σ: in different states, pv can take on different procedure values (though the parameters must be of the correct type) with different globals referred to, and the remainder of the state, t, will depend on the result parameters used.

Assignment and monotonicity

Having given meaning to calls of procedure variables, we now consider assignments to such variables. The semantic framework which we use to describe

assignments is complex at first sight, but it soon becomes clear that the restrictions we make actually have no effect on most datatypes — and we can therefore inherit all the normal refinement calculus results, rather than having to re-prove them.

The reason for the unusual semantic framework is the need for monotonicity. Monotonicity is a fundameutal property of all constructs in the refinement calculus, and is the basis for the development strategy known as 'stepwise refinement'. It is because all the constructs are monotonic that we can refine specifications in isolation, assemble the code with (monotonic) constructs, and remain sure that the resulting program is a valid refinement of the combined specifications. However, the procedure variable assignment $pv := pe$ has pe as a sub-program — pv is a variable and not a sub-program, hence not subject to refinement — but $pe \sqsubseteq pe'$ does not imply that $pv := pe \sqsubseteq pv := pe'$. Thus assignment to a procedure variable is not monotonic.

Naumann's suggestion [48] to solve this problem was to use a geueralised assignment statement $pv :\sqsupseteq pe$, defined by analogy with Morgan's Simple Specification ahbreviation [44, Abbreviation 8.1][2]. Tbis construct assigns to pv any program wbich is a refinement of the expression pe. Monotonicity of this construct follows immediately from transitivity of $\sqsubseteq$, and consideration of the behaviour of some compilers also makes this construct seem reasonable: in the case wbere pv is merely a pointer to the code of pe, then $pv := pe$ will indeed establish $pv = pe$, but compilers often 'optimize' programs by making them more deterministic or by making them terminate more often (by monitored execution, subscript range checking etc). So it is not clear that $pv := pe$ will establish $pv = pe$ in such cases anyway. However, as long as the compiler is correct, it should at least be the case that the value of pv will he at least as good as pe — that $pv \sqsupseteq pe$.

Although we now have a monotonic construct as desired, it turns out that, in the traditional powerset model, only very weak refinement laws can be proved about $:\sqsupseteq$ — for instance, it is no longer the case that $pv :\sqsupseteq pv$ and **skip** are equivalent: althougb it is easy to show that $pv :\sqsupseteq pv \sqsubseteq \mathbf{skip}$, tbe other direction is not true. In other words, for some predicate ϕ,

$$wp(\mathbf{skip}, \phi) \not\Rightarrow wp(pv :\sqsupseteq pv), \phi) \ .$$

The underlying cause of the problems with these laws is that predicates may not be monotonic with respect to refinement: for instance, $pv = S$ is not monotonic in pv because it is satisfied by S but not by any proper refinement of S.

The solution to this difficulty proposed by Naumann [48] is to banish all non-monotonic predicates, thus restoring tbe important refinement laws. The non-monotonic predicates are removed by taking as predicates not all possible sets of states, but only those which are up-closed under the relevant ordering: if X is a poset (with respect to $\leq$), tben a subset ϕ of X is up-closed exactly when

$$\forall a, b : X \bullet a \in \phi \wedge a \leq b \Rightarrow b \in \phi \ .$$

[2] Morgan points out that the notation is due originally to Jean-Raymond Abrial.

There are two useful observations we can make about up-closed subsets:

- for any $a \in X$, the set $\{x : X \mid a \leq x\}$ is up-closed; and
- if X is discretely ordered ($x \leq y$ iff $x = y$), then every subset of X is up-closed, and every subset of $X \times X$ is also up-closed.

The first observation will be useful when we later consider statements about refinement as state predicates, but the second is important now. Every type T is interpreted as a pre-ordered set. For most datatypes — all except procedure types — the ordering is simply equality, giving a discretely ordered set. The second observation above then tells us that, for predicates not involving procedure variables, the restriction of predicates to up-closed subsets is vacuous — we can still use all the results of the traditional powerset model. It is only for procedure types that we have to be careful to use up-closed predicates (in particular, we cannot use equality). Thus for ordinary variables — those not of procedure type — we use the standard assignment construct with its usual (substitution) semantics, while for procedure type variables, we use the generalised assignment $:\sqsupseteq$, which is formally defined below.

Having defined our semantic framework, we now carry out some investigations to ensure that the framework has the right properties:

- that the standard programming constructs of the refinement calculus maintain up-closure;
- that the new constructs which deal with procedure variables also maintain up-closure;
- that the conditions which guarantee that a specification statement maintains up-closure are reasonable; and
- that the framework is sufficiently well-behaved that recursive constructs are well-defined.

Standard programming constructs We deal first with the case where the postcondition contains no reference to state variables of procedure type. In this case, the ordering that concerns us for up-closure is equality, and so we can appeal to the usual results about the monotonicity of the constructs of the refinement calculus [44].

For postconditions which do refer to state variables of procedure type, we proceed by structural induction. For each statement S, we must show that if ϕ is an up-closed set of states, then so too is $wp(S, \phi)$, given that the subcomponents of S also preserve up-closure.

From the definition of up-closure, we have, for a procedure variable pv,

$\phi(pv)$ is up-closed
iff $\phi(pv) \wedge pv \sqsubseteq pv' \Rightarrow \phi(pv')$

We now look at the standard programming constructs of the refinement calculus in turn:

- **skip**

  $$wp(\mathbf{skip}, \phi) = \phi$$

  so we have nothing to prove.

- **abort**

  $$wp(\mathbf{abort}, \phi) = \mathit{false}$$

  which is up-closed.

- sequential composition

  $$wp(aaa;\ bbb, \phi) = wp(aaa, wp(bbb, \phi))$$

  Thus the up-closure of $wp(aaa;\ bbb, \phi)$ follows immediately from the up-closure of ϕ and the inductive hypothesis that aaa and bbb preserve up-closure.

- assignment to a simple (non-procedure) variable $x := E$;

  $$wp(x := E, \phi) = \phi[x \backslash E]$$

  up-closure follows directly from the up-closure of ϕ.

- alternation

  $$\begin{aligned} &wp(\mathbf{if}\ [\!]\ \alpha_i \rightarrow aaa_i\ \mathbf{fi}, \phi) \\ &\quad = \bigvee \alpha_i \wedge \\ &\qquad (\bigwedge(\alpha_i \Rightarrow wp(aaa_i, \phi))) \end{aligned}$$

  We give names to the formulae in the hypothesis and conclusion:

  $$\begin{aligned} H1: &\quad \bigvee \alpha_i \wedge (\bigwedge(\alpha_i \Rightarrow wp(aaa_i, \phi))) \\ H2: &\quad pv \sqsubseteq pv1 \\ H3: &\quad (\forall pv2 \bullet \phi(pv) \wedge pv \sqsubseteq pv2 \Rightarrow \phi(pv2)) \\ C1: &\quad (\bigvee \alpha_i)[pv \backslash pv1] \\ C2: &\quad (\bigwedge(\alpha_i \Rightarrow wp(aaa_i, \phi)))[pv \backslash pv1] \end{aligned}$$

  We need to prove that $H1 \wedge H2 \wedge H3 \Rightarrow C1 \wedge C2$. $C1$ follows immediately from $H1$, once we insist that no procedure variable may appear in a guard. For $C2$, we can distribute the substitution through the conjunction to get

  $$\bigwedge(\alpha_i \Rightarrow wp(aaa_i, \phi)[pv \backslash pv1])\ ,$$

  again assuming that pv does not occur in any α_i. Now let i be any index. If α_i is false, then we can immediately conclude that

  $$\alpha_i \Rightarrow wp(aaa_i, \phi)[pv \backslash pv1]\ .$$

If α_i is true, then from $H1$, we can see that $wp(aaa_i, \phi)$ must also be true. By the inductive hypothesis, this is an up-closed set, and hence $wp(aaa_i, \phi)[pv\backslash pv1]$ must be true, giving us once again

$$\alpha_i \Rightarrow wp(aaa_i, \phi)[pv\backslash pv1] \ .$$

But i was arbitrary, so we can couclude

$$\bigwedge(\alpha_i \Rightarrow wp(aaa_i, \phi)[pv\backslash pv1]) \ ,$$

and, from that, C2 follows.

New constructs for procedure variables There are two constructs that we need to consider: assignment and procedure call:

- assignment to a procedure variable $pv :\sqsupseteq pe$. The generalised assignment is defined as follows[3]:

$$wp(pv :\sqsupseteq pe, \phi)$$
$$= (\forall\, pv' \bullet pv' \sqsupseteq pe \Rightarrow \phi[pv\backslash pv'])$$

(pv' must be a fresh variable in this formula.) We call this formula $\psi(pw)$ for arbitrary procedure variable pw. We need to prove

$$\psi(pw) \wedge pw \sqsubseteq pw1 \Rightarrow \psi(pw1)$$
given that ϕ is up-closed.

We give names to the various formulae:

$H1 : \psi(pw)$	$(\forall\, pv' \bullet pv' \sqsupseteq pe \Rightarrow \phi[pv\backslash pv'])$
$H2 :$	$pw \sqsubseteq pw1$
$H3 :$	ϕ is up-closed
$C : \psi(pw1)$	$(\forall\, pv' \bullet pv' \sqsupseteq pe \Rightarrow \phi[pv\backslash pv'])[pw\backslash pw1]$

We have to prove that $H1 \wedge H2 \wedge H3 \Rightarrow C$, and we start by re-writing C, taking the outer substitution inside the quantification:

$$C : \psi(pw1) \qquad (\forall\, pv' \bullet pv' \sqsupseteq pe[pw\backslash pw1] \Rightarrow \phi[pv\backslash pv'][pw\backslash pw1]) \ .$$

Now we choose an arbitrary pv' such that $pv' \sqsupseteq pe[pw\backslash pw1]$. If such a pv' cannot be found, then C is trivially true. Monotonicity of pe tells us that

$$pw \sqsubseteq pw1 \Rightarrow pe \sqsubseteq pe[pw\backslash pw1] \ ,$$

[3]This definition agrees with that given by Morgan's Simple Specification abbreviation:

$pv :\sqsupseteq pe = pv : [pv \sqsupseteq pe[pv\backslash pv_0]]$.

and we know that $pw \sqsubseteq pw1$ from $H2$. So we have

$$pe \sqsubseteq pe[pw \backslash pw1]$$

Thus by transitivity of $\sqsubseteq$, we have

$$pv' \sqsupseteq pe \ .$$

and, by $H1$,

$$\phi[pv \backslash pv'] \ .$$

The up-closure of ϕ also tells us that $\phi[pv \backslash pv']$ is up-closed. Thus

$$\phi[pv \backslash pv'] \wedge pw \sqsubseteq pw1 \Rightarrow \phi[pv \backslash pv'][pw \backslash pw1] \ .$$

Both antecedents are true (from above and $H2$), and so the conclusion is true, giving us C as required.

- procedure variable call **call** $pv(e, w)$. Although the formula for the weakest precondition given in Definition 6.5 is rather formidable, most of the complications come from the parameter passing. Without that, we simply have to show that $f \otimes id_{w,t}$ preserves up-closure, given that f does. If the variable we are considering is pw, then a case analysis gives us the desired result: if pw appears in the list w, t, then $f \otimes id_{w,t}$ has no effect on it; and if pw is not in that list, then it is covered by f and the inductive hypothesis tells us that up-closure is maintained.

Specification statements Since the specification statement is an additional construct which does not appear in Dijkstra's language of guarded commands, we do not have to ensure that it always maintains up-closure. Instead we can investigate the conditions under which it does so, and insist that those conditions are met in the new language which includes procedure variables. The conditions for maintenance of up-closure will be conditions on *pre* and *post*, where these are the two predicates which form the specification statement. Specifically, we show that the up-closure of *pre* and the up-closure of $\neg$ *post* — or, equivalently, the down-closure of *post* — are sufficient together to guarantee that $wp(w : [pre, post], \phi)$ maintains up-closure of ϕ.

We may assume:

$H1$: $pv \sqsubseteq pv1$
$H2$: pre
$H3$: $(\forall w \bullet post \Rightarrow \phi) \upharpoonright w$

We have to prove[4]:

$C1$: $pre[pv \backslash pv1]$
$C2$: $((\forall w \bullet post \Rightarrow \phi) \upharpoonright w)[pv \backslash pv1]$

[4]Note that the *wp* definition of a specification statement is slightly different from the standard form given in Chapter 2, as we have to take account of the state spaces.

C1 follows directly from H1, H2 and the up-closure of *pre*.

For C2, we note that w must be different from pv, since we forbid specification statements over procedure variables. Hence the $\dagger w$ and the substitution for pv can be safely interchanged. We therefore need to show that

$$\neg\ post[pv\backslash pv1] \vee \phi[pv\backslash pv1] \ .$$

From H3, we know that, for a particular w, either $\neg\ post$ holds or ϕ does:

- if $\neg\ post$ holds, then its up-closure and H1 give us $\neg\ post[pv\backslash pv1]$
- if ϕ holds, then its up-closure and H1 give us $\phi[pv\backslash pv1]$.

In either case, $\neg\ post[pv\backslash pv1] \vee \phi[pv\backslash pv1]$ holds, giving the desired result.

Recursion The set over which we need recursive definitions to be well-defined is the set of programs, and, by the results above, we can restrict our attention to programs which preserve up-closure. We need to show that this set, together with the refinement relation, forms a complete partial order (cpo), and thus that any monotonic function on the set — a program context — has a least fixed-point.

So our task is to show that every chain in the set has a least upper bound (in the set). Suppose that C_i is such a chain of increasingly-refined up-closure-preserving programs. The existence of the least upper bound as a program follows from standard results, but we need to show that $\bigsqcup_i C_i.\bot$ also preserves up-closure. In other words, we need

$$(\bigsqcup_i C_i.\bot).\phi.x \wedge x \sqsubseteq x' \Rightarrow (\bigsqcup_i C_i.\bot).\phi.x' \qquad (\dagger)$$

Now, by definition, $(\bigsqcup_i C_i.\bot).\phi.x = (\bigsqcup_i C_i.\bot.\phi).x$, which is either true or false for a particular value of x. If it is false, then we are finished, since the antecedent of $(\dagger)$ is false. If it is true, then, since the C_i are increasing, there must be some j for which $C_j.\bot.\phi.x$ is true. Since C_j preserves up-closure, we therefore know that $C_j.\bot.\phi.x'$ is also true, and hence that $(\bigsqcup_i C_i.\bot.\phi).x'$, as required. Thus the least upper bound of the chain preserves up-closure.

Connectives Having examined the semantic framework, and noted that it has the desired properties, it is also interesting to look at the predicate connectives to find out which of them maintain the property of upward-closure: when their arguments are upward-closed, so should their result be. Conjunction and disjunction of predicates are given set-theoretically by union and intersection, respectively, and these do indeed preserve upward-closure. However, negation is more complicated, because the complement of an up-closed set need not be up-closed. We first define a version of implication which preserves up-closure by

$$\phi \Rightarrow \psi = \cup(\delta \bullet (\delta \text{ is up-closed}) \wedge \delta \cap \phi \subseteq \psi) \ .$$

Now we can define negation in terms of implication:

$$\dot{\neg}\,\phi = \phi \Rightarrow \varnothing \ .$$

It is easy to see that $\dot{\neg}\,\phi$ is the largest up-closed set that is disjoint from ϕ. For discretely-ordered X, $\dot{\neg}\,\phi$ is simply the set-theoretical complement $X - \phi$.

Quantifications are defined in terms of the projection operator introduced above:

$$\sigma \in (\exists\, x \bullet \phi) \equiv (\exists\, \tau \mid \sigma = \tau \lfloor x \bullet \tau \in \phi)$$
$$\sigma \in (\forall\, x \bullet \phi) \equiv (\forall\, \tau \mid \sigma = \tau \lfloor x \bullet \tau \in \phi)$$

Now we are in a position where we can prove the following lemma:

Lemma

If ϕ is up-closed, then $wp(\mathbf{skip}, \phi) \Rightarrow wp(pv :\sqsupseteq pv, \phi)$

Proof

$$\begin{array}{ll} & wp(pv :\sqsupseteq pv, \phi) \\ = & \text{“definition of } :\sqsupseteq\text{”} \\ & (\forall\, pv' \bullet pv' \sqsupseteq pv \Rightarrow \phi[pv \backslash pv']) \end{array}$$

Now $wp(\mathbf{skip}, \phi) = \phi$, and so we must show that, for any pv',

$$\phi \wedge pv \sqsubseteq pv' \Rightarrow \phi[pv \backslash pv'] \ .$$

But this follows directly from the definition of up-closure of ϕ, and so the lemma is proved. □

This gives us the equivalence of **skip** and $pv :\sqsupseteq pv$.

State predicates involving refinement and some basic laws

The final step before we can prove the correctness of various laws involving procedure variables is to consider exactly what is meant by state predicates which use the refinement relation. These might appear for instance in the pre- or post-condition of a specification statement or in a guard. Wherever such a predicate may appear, the mechanism for evaluation is the same: the two operands should be extended, by taking their product with appropriate identity transformers, until they both act on the same state space.

For instance, suppose our state comprises the two natural numbers x and y, and consider the state predicate

$$x : [x = x_0 + 1] \sqsubseteq (\mathbf{proc}\, x := x + 1) \ .$$

The meaning of the LHS is a predicate transformer over (x, y), whereas the RHS acts only on x. In order to compare the two, we take the product with id_y:

$$\begin{array}{l}\text{for any } \phi \text{ (over } x \text{ and } y\text{)},\\ wp(x : [x = x_0 + 1], \phi) \Rightarrow wp((x := x + 1) \otimes id_y, \phi) \ ,\end{array}$$

which simplifies to

$$\phi[x\backslash x + 1] \Rightarrow \phi[x\backslash x + 1] \ .$$

Alternatively, suppose there is also a procedure variable pv in the state — for simplicity, we assume it takes no parameters. To evaluate

$$x : [x = x_0 + 1] \sqsubseteq pv \ .$$

we need to find out which variables the current value of pv acts on. Suppose that

$$\sigma.pv = \langle f, \ , x\rangle \ ,$$

where σ is the current state. Thus the current value of pv is a predicate transformer f which acts only on x. Then, to evaluate the predicate above, we must stipulate that y and pv should remain unchanged:

$$\begin{array}{l}\text{for any } \phi \text{ (over } x \text{ and } y\text{)},\\ wp(x : [x = x_0 + 1], \phi) \Rightarrow wp(f \otimes id_{y,pv}, \phi) \ .\end{array}$$

We are now able to prove three basic laws about the execution of procedure variables. The first law that we prove is the simplest one, where the procedure variable has no parameters:

Law 6.6 *introduce procedure variable execution*

$$w : \left[\begin{array}{c} w : [pre, post] \sqsubseteq pv \\ pre \end{array}, post\right] \ \sqsubseteq \ \textbf{call } pv$$

Proof In order to prove this law, we must show that the weakest-precondition of the left-hand side implies the weakest precondition of the right-hand side. Looking back at the definitions given earlier, we can see that the right-hand side gives us

$$(f \otimes id_t).\phi$$

where f is the predicate transformer part of the current value of pv — again we assume no parameters for now — and t is the part of the state on which f does not act. On the left-hand side, we get, by the definition of wp for a specification

statement

$$(w : [pre, post] \sqsubseteq pv) \wedge pre$$
$$\wedge\ (\forall w \bullet post \Rightarrow \phi) \dagger w \ .$$

Expanding the predicate involving $\sqsubseteq$, we get

for any ψ,
$$(pre \wedge (\forall w \bullet post \Rightarrow \psi) \dagger w) \Rightarrow (f \otimes id_t).\psi$$
$$\wedge\ pre$$
$$\wedge\ (\forall w \bullet post \Rightarrow \phi) \dagger w \ .$$

Taking ψ to be ϕ gives us the required result immediately.

□

Law 6.7 *procedure variable value assignment*

If the procedure variable pv has heen declared as **procedure** (**value** v), then we have the following refinement:

$$w : \left[\begin{array}{c} pre \\ w := E \sqsubseteq pv \end{array}, post \right] \sqsubseteq \textbf{call}\ pv(A) \qquad (1)$$
$$provided\ w : [pre, post] \sqsubseteq w := E[v\backslash A]$$

where A contains no v

Proof Suppose that the value of pv in the current state is $\langle f, v, , g \rangle$, so f is a predicate transformer over v and the global variables g. Taking weakest preconditions on the left of (1), with respect to a predicate ϕ over the whole state (g and t), we get

$$H1: \quad pre$$
$$H2: \quad w := E \sqsubseteq pv$$
$$H3: \quad (\forall w \bullet post \Rightarrow \phi) \dagger w \ ,$$

and on the right we get

$$C: \quad (\forall v \bullet ((f \otimes id_t).(\phi \dagger v))[v\backslash A]) \ .$$

Since v does not appear in A, the universal quantification in C is vacuous, so our revised goal is

$$C': \quad ((f \otimes id_t).(\phi \dagger v))[v\backslash A] \ .$$

We can re-express $H2$ in a more useful way as

for any ψ over the whole state g and t
$$(\psi \dagger v)[w\backslash E] \Rightarrow (f \otimes id_t).(\psi \dagger v) \ .$$

Now the antecedent here is equivalent to $\psi[w\backslash E]$, so we have

$$H2': \quad \psi[w\backslash E] \Rightarrow (f \otimes id_t).(\psi \dagger v) \ .$$

The proviso to the law can be expressed as

For any η over g and t,
$pre \wedge (\forall w \bullet post \Rightarrow \eta) \dagger w \ \Rightarrow \ \eta[w\backslash E[v\backslash A]]$

The consequent here is equivalent to $\eta[w\backslash E][v\backslash A]$, since the only place v can appear in $\eta[w\backslash E]$ is in E itself.

Now from $H1$, $H3$ and the proviso (witb η instantiated to ϕ), we get

$$\phi[w\backslash E][v\backslash A] \ .$$

By monotonicity of textual substitution and $H2'$ (with ψ instantiated to ϕ), we get

$$\phi[w\backslash E][v\backslash A] \Rightarrow ((f \otimes id_t).(\phi \dagger v))[v\backslash A] \ .$$

Putting these two together, we get C' as required.

□

Law 6.8 *procedure variable result assignment*

If the procedure variable pv has been declared as **procedure** (**result** r), then we have the following refinement:

$$a : \left[\begin{matrix} pre \\ r := E \sqsubseteq pv \end{matrix} \ , post \right] \sqsubseteq \ \textbf{call}\ pv(a) \qquad (2)$$

provided $a : [pre, post] \sqsubseteq a := E$

where r does not occur in E.

Proof Suppose that the value of pv in the current state is $(f, , r, g)$, so f is a predicate transformer over r and the global variables g. Taking weakest preconditions on the left of (2), with respect to a predicate ϕ over the whole state (g, a and t), we get

$$\begin{aligned} H1: &\quad pre \\ H2: &\quad r := E \sqsubseteq pv \\ H3: &\quad (\forall a \bullet post \Rightarrow \phi) \dagger a \ , \end{aligned}$$

and on the right we get

$$C: \quad (\forall r \bullet ((f \otimes id_{a,t}).(\phi \dagger r)[a\backslash r]) \ .$$

The proviso can be expressed as

For any η over g, a and t,
$pre \wedge (\forall a \bullet post \Rightarrow \eta) \dagger a \ \Rightarrow \ \eta[a\backslash E]$

$H2$ is equivalent to

for any ψ over the whole state g, a and t
$(\psi \dagger r)[r \backslash E] \Rightarrow (f \otimes id_{a,t}).(\psi \dagger r)$,

which in turn is equivalent to

$$(\psi[r \backslash E]) \dagger r \Rightarrow (f \otimes id_{a,t}).(\psi \dagger r) \ .$$

Taking ψ to be $\phi[a \backslash r]$, we get

$$(\phi[a \backslash r][r \backslash E]) \dagger r \Rightarrow (f \otimes id_{a,t}).(\phi[a \backslash r] \dagger r) \ ,$$

which can be simplified to

$$(\phi[a \backslash E]) \dagger r \Rightarrow (f \otimes id_{a,t}).(\phi \dagger r)[a \backslash r] \ .$$

By the lemma below, we can simplify this again to

$$\phi[a \backslash E] \Rightarrow \forall r \bullet (f \otimes id_{a,t}).(\phi \dagger r)[a \backslash r] \ .$$

From $H1$, $H3$ and the proviso, with η instantiated to ϕ, we can conclude $\phi[a \backslash E]$, giving us the desired result.

□

Lemma For any predicates α (over some state) and β (over the state extended with a fresh variable x), if we know that

$$(\alpha \dagger x) \Rightarrow \beta(x)$$

then

$$\alpha \Rightarrow (\forall x \bullet \beta(x)) \ .$$

Proof Suppose that

$$(\alpha \dagger x) \Rightarrow \beta(x) \ .$$

Then, if σ is a state in the set $\alpha \dagger x$, it must also be in the set $\beta(x)$. So, by the definition of $\dagger$,

$$(\sigma \lfloor x \in \alpha) \Rightarrow (\sigma \in \beta) \ . \tag{3}$$

Now suppose that σ' is any state in α. We must show that $\sigma' \in \forall x \bullet \beta(x)$. By the definition of $\forall$ over our up-closed sets of states, we therefore need

$$(\forall \tau \mid \sigma' = \tau \lfloor x \bullet \tau \in \beta) \ .$$

Consider any τ such that $\sigma' = \tau \lfloor x$. Then we know that $\tau \lfloor x \in \alpha$, since $\sigma' \in \alpha$. Therefore, by (3), we know that $\tau \in \beta$. But τ was arbitrary, so we have the required result about τ, and we can conclude that $\sigma' \in \forall x \bullet \beta(x)$.

□

6.3 Laws for procedure variables

Proofs of these laws follow directly from the definitions.

Law 6.9 *procedure variable value specification*

If the procedure variable pv has been declared as **procedure** (**value** f), then we have the following refinement:

$$w : \left[\begin{array}{c} pre \\ w : [pre_1, post_1] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(A)$$

$$\textit{provided } w : [pre, post] \sqsubseteq w : [pre_1[f \backslash A], post_1[f_0 \backslash A_0]]$$

where A_0 is $A[w \backslash w_0]$ and $post_1$ contains no f

Law 6.10 *procedure variable result specification*

If the procedure variable pv has been declared as **procedure** (**result** f), then we have the following refinement:

$$a : \left[\begin{array}{c} pre \\ f : [pre_1, post_1[a \backslash f]] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(a)$$

$$\textit{provided } a : [pre, post] \sqsubseteq a : [pre_1, post_1]$$

where f does not occur in pre_1, and neither f nor f_0 occur in $post_1$.

6.4 Naumann's syntactic restrictions

Naumann's first restriction, which he called the *Global Variable Constraint* is intended to simplify the implementation of procedure variables using stack allocation, by ensuring that external variables of stored procedures — those assigned to variables or passed as parameters — are visible at every point of call. Explicitly, he states that

> 'no variable free in the body of a procedure assigned to a procedure variable (or passed as an actual parameter) is bound by *var*, *aux* or *pro*'.

(Naumann's *var*, *aux* and *pro* denote local variables, logical constants and procedure expressions.)

The second constraint ensures the absence of aliasing. It has two parts:

- in calls of procedure constants and procedure expressions, the free variables of the called procedure do not appear in the actual parameter list;
- in calls of procedure variables (and formal procedure type parameters) only variables bound by **var** may appear as actual result parameters; thus by the Global Variable Constraint, they are distinct from the externals of the procedure variable.

6.5 Conclusion

In this chapter, we have stepped back from the work on iterators to look at how variables of procedure type can be incorporated into the language of the refinement calculus. The semantics of such variables was described using weakest preconditions, following Naumann's work, while the original work in this chapter showed the well-formedness of Naumann's constructions, and the validity of several new refinement laws about procedure variables. The work described in this chapter will form the basis of Chapter 7.

Chapter 7

Encapsulating iterators

In this chapter we bring together the work of the previous two chapters — Chapter 5 on the **it..ti** construct, and Chapter 6 on procedure variables — to show how we can encapsulate the iterator construct into a procedure itself. This euables us to put forward a development method which is based on the refinemeut calculus and which uses a pre-defined library of abstract data types. We start by considering the use of procedures as parameters, based on the theory of procedure variables, hefore giving an example, and showing how to encapsulate the iterator construct. Subsequent chapters will show the use of the libraries of abstract data types on a larger scale.

7.1 Procedures as parameters

Now that we have given syntax and semantics to procedure variables, thereby putting procedure values on the same level as values of other types, it is only a small extension to allow procedure variahles as parameters, thus permitting procedure values to be passed to and from other procedures just as other values are passed.

Syntax

We allow parameters of procedure type to appear in parameter lists for a procedure, just as simple (non-procedure) parameters do. These procedural parameters may be passed by value, result or value-result. When a procedural

parameter is used, its own parameters must be specified in the parameter list.[1] For example, consider the following procedure definition:

$$\begin{array}{l}\textbf{procedure } P\ (\textbf{value } x, y : \mathbb{N};\ fp : \textbf{procedure } (\textbf{value } v : V);\\ \qquad\qquad\qquad \textbf{result } op : \textbf{procedure } (\textbf{value } w : W))\\ \widehat{=}\\ \qquad \ldots\end{array}$$

This declares P to be a procedure which has three value parameters and one result parameter. Of the value parameters, two are simple parameters (x and y, both numbers) and the other, fp, is a procedure which itself takes a single value parameter. When P is called, an actual procedure value of the correct type must be supplied for fp, just as numeric values must be supplied for x and y. The result of executing P is communicated through the result parameter op: again the name of an actual procedure variable of the correct type must be supplied on the call.

It is worth recalling here that Naumann gave some additional syntactic constraints in [48] to simplify implementation and prevent aliasing. These are summarised in Section 6.4.

Semantics

We recall that the definition of substitution by value for non-procedure variables was constructed so that the following equality would hold:

$$\begin{array}{ll} & P[\textbf{value } f \backslash A]\\ = & |[\ \textbf{var } l \bullet\\ & \quad l := A;\\ & \quad P[f \backslash l]\\ &]|\end{array}$$

where P is a program, f a variable, A a term and l a fresh local variable. In order to deal with substitutions for procedure variables, we will use very much the same approach. However, there is a problem with the program fragment above, when we replace f by a procedure variable pv: we must also replace l by a fresh local procedure variable lp. We then have an assignment to lp, and we recall that assignment to procedure variables is a non-monotonic construct. So instead, we want a definition that maintains the following equality:

$$\begin{array}{ll} & P[\textbf{value } fp \backslash AP]\\ = & |[\ \textbf{var } lp \bullet\\ & \quad lp :\sqsupseteq AP;\\ & \quad P[fp \backslash lp]\\ &]|\end{array}$$

Now fp is a procedure variable, AP a term suitable for assignment to such a variable — a program fragment — and lp a fresh local procedure variable of suitable type.

[1] Remember however that global variables are not specified for a procedure type.

The definition of this substitution is as follows:

Definition 7.1 *procedure value substitution*

$$\begin{array}{ll} & wp(P[\textbf{value } fp \backslash AP], \phi) \\ = & \forall X \bullet X \sqsupseteq AP \Rightarrow wp(P, \phi)[fp \backslash X] \end{array}$$

It should be noted that this definition is consistent with the standard definition of substitution by value: for a variable x of discrete (non-procedural) type, an assignment $x :\sqsupseteq e$ is equivalent to the normal $x := e$. A similar remark applies to the definition of result substitution which follows.

Similarly, the definition of (non-procedural) substitution by result was chosen to give the following equality:

$$\begin{array}{ll} & P[\textbf{result } r \backslash a] \\ = & |[\ \textbf{var } l \bullet \\ & \quad P[r \backslash l]; \\ & \quad a := l \\ &]| \end{array}$$

In the case of procedure parameters, r, a and l must all be procedure variables, and so the assignment to a is a problem. Instead we aim at the following equality:

$$\begin{array}{ll} & P[\textbf{result } rp \backslash ap] \\ = & |[\ \textbf{var } lp \bullet \\ & \quad P[rp \backslash lp]; \\ & \quad ap :\sqsupseteq lp \\ &]] \end{array}$$

Now rp, ap and lp must all be procedure variables of the same type.

The definition of this substitution is as follows:

Definition 7.2 *procedure result substitution*

$$\begin{array}{ll} & wp(P[\textbf{result } rp \backslash ap], \phi) \\ = & \forall lp \bullet wp(P[rp \backslash ap], (\forall ap \bullet ap \sqsupseteq lp \Rightarrow \phi) \dagger ap) \end{array}$$

While these definitions give precise semantics to the procedure variable versions of the two major substitution forms, they are not particularly user-friendly. However, as an alternative to using the laws, we can also manipulate the specification under consideration until it matches the program fragments above, and then immediately replace it with the appropriate procedure call. Experience has shown that this is actually more helpful than proposing and proving other laws using the weakest-precondition definitions.

7.2 Example

Our first example[2] of the use of procednres as parameters concentrates on the use of value parameters. Later we'll show how to nse result parameters to combine two iterators to form a third. We consider the development and use of a procedure, *findmax*, to find the maximum of an array a of elements of type T, where the ordering relation R is not hard-coded into the procedure, but passed as a parameter: for any two values of T, say x and y, the procedure parameter rp should return a boolean value b to show whether or not they are related; that is, execution of rp will establish $b \Leftrightarrow x\ R\ y$. The development of *findmax* gives a further example of the use of procedure variables — in the development of a procedure which has a procedure parameter, that parameter is treated just as a local procedure variable — while the use of *findmax* shows particular procedure values heing passed as actual parameters.

There are two value parameters and one result parameter to *findmax*:

$$\begin{array}{l} \mathit{findmax}\ (\mathbf{value}\ \ a : \mathit{array}[0\,..\,N-1]\ \mathit{of}\ T; \\ \qquad\qquad\qquad\quad rp : \mathbf{procedure}\ (\mathbf{value}\ x, y : T, \mathbf{result}\ b : \mathit{Bool}); \\ \qquad\quad \mathbf{result}\ \ m : T) \end{array}$$

The postcondition that we want *findmax* to establish is that m is the maximum value (in the R-ordering) in a:

$$\forall j : 0\,..\,N \bullet a[j]\ R\ m \wedge m\ \mathbf{in}\ a \tag{1}$$

The precondition, from which we have to establish (1), must contain the fact that R is a total order. We must also assume that rp computes the relation R, which can be expressed as follows:

$$b : [b \Leftrightarrow x\ R\ y] \ \sqsubseteq\ rp\ . \tag{2}$$

We note that, in the definition of the type of rp, the variables x, y and b would normally be taken as place-holders, but we need to use them in specification statements. Specifically, we expect predicates about rp, such as (2), to include specification statements with x, y and b free in the pre- and/or postconditions.

So the specification of *findmax* is

$$\mathit{findmax} \mathrel{\widehat{=}} m : [R \text{ is a total order} \wedge (2), (1)]$$

For brevity, we omit, from here on, the requirement that R should he a total order — formally, it should be carried through each precondition.

[2]The example is taken from [49] where it is expressed in terms of Hoare triples, rather than the refinement calculus.

Standard development steps take us to the following program:

$$
\begin{array}{l}
\mathit{findmax}\,(\mathbf{value}\ \ a : \mathit{array}[0\,..\,N-1]\ \mathit{of}\ T; \\
\qquad\qquad\qquad\quad \mathit{rp} : \mathbf{procedure}\ (\mathbf{value}\ x, y : T, \mathbf{result}\ b : \mathit{Bool}); \\
\qquad\quad \mathbf{result}\ m : T) \\
\widehat{=} \\
\mathbf{var}\ i : \mathit{int} \bullet \\
\quad m, i := a[0], 1; \\
\quad \mathbf{do}\ i \neq N \rightarrow \\
\qquad \mathbf{var}\ c : \mathit{Bool} \bullet \\
\qquad\quad c : [I, J]; \\
\qquad\quad \mathbf{if}\ c\ \mathbf{then}\ m := a[i]; \\
\qquad i := i + 1 \\
\quad \mathbf{od}\ ,
\end{array}
\tag{3}
$$

where the loop invariant I and the postcondition of (3) J are given by:

$$
\begin{array}{l}
I \mathrel{\widehat{=}} (b : [b \Leftrightarrow x\,R\,y] \sqsubseteq \mathit{rp}) \wedge (\forall j : 0\,..\,i-1 \bullet a[j]\,R\,m) \wedge (m\ \mathbf{in}\ a) \\
J \mathrel{\widehat{=}} (\forall j : 0\,..\,i-1 \bullet a[j]\,R\,m) \wedge (m\ \mathbf{in}\ a) \wedge (c \Leftrightarrow m\,R\,a[i])\ .
\end{array}
$$

Of course, the interesting part of this development for us comes in justifying the replacement of (3) with a call of rp. Expanding I and J and removing the conjuncts which occur in both precondition and postcondition[3] gives

$$
(3) \sqsubseteq \quad c : [b : [b \Leftrightarrow x\,R\,y] \sqsubseteq \mathit{rp}, c \Leftrightarrow m\,R\,a[i]\,]
$$

Since rp has both value and result parameters, we need a combination of two laws given in the last chapter, *procedure variable value specification* 6.9 and *procedure variable result specification* 6.10[4]:

Law 7.3 *proeedure variable value and result specification*

If the procedure variable pv has been declared as **procedure** (**value** v, **result** r), then we have the following refinement:

$$
w, ar : \left[\begin{array}{c} pre \\ w, r : [pre_1, post_1[ar\backslash r]] \sqsubseteq pv \end{array}, post \right] \sqsubseteq \mathbf{call}\ pv(A, ar)
$$

$$
\mathit{provided}\ w, ar : [pre, post] \sqsubseteq w, ar : [pre_1[v\backslash A], post_1[v\backslash A]\,]
$$

where r does not occur in pre_1, and neither r nor r_0 occurs in $post_1$.

Comparing the left-hand side of this rather formidable law with (3), it is not

[3] First the two conjuncts are removed from the postcnndition, since they can be derived from the precondition. Then they are removed from the precondition by simple weakening.

[4] The law given here is slightly simplified. the full version is given at the end of this chapter.

too difficult to match up the variables and predicates:

Law 7.3	(3)
w	
ar	c
pre	*true*
v	x, y
r	b
pre_1	*true*
$post_1$	$c \Leftrightarrow x\ R\ y$
pv	rp
$post$	$c \Leftrightarrow m\ R\ a[i]$
A	$m, a[i]$

The variable freedom provisos do not cause any problem. We need to have r and r_0 not occurring in $post_1$, which in this case means that b and b_0 must not occur in $c \Leftrightarrow x\ R\ y$. We must also have r not occurring in pre_1, which here is *true*.

Similarly, the side condition is satisfied: in order for the law to be valid, we need

$$w, ar : [pre, post] \sqsubseteq w, ar : [pre_1[v\backslash A], post_1[v\backslash A]\ ]\ .$$

In this case, A is the pair $m, a[i]$, and so the condition is

$$\begin{array}{ll} & c : [c \Leftrightarrow m\ R\ a[i]\ ] \\ \sqsubseteq & c : [(c \Leftrightarrow x\ R\ y)[x, y\backslash m, a[i]]\ ]\ , \end{array}$$

which is clearly true.

Since the conditions are all satisfied, we can conclude that

$$(3) \sqsubseteq \begin{array}{l} \text{"}\textit{procedure variable value and result specification}\ 7.3\text{"} \\ \textbf{call}\ rp(m, a[i], e) \end{array}$$

This completes the development. Collecting the code gives the following program:

```
findmax (value  a : array[0 .. N − 1] of T;
                rp : procedure (value x, y : T, result b : Bool);
         result m : T)
≙
var i : int •
    m, i := a[0], 1;
    do i ≠ N →
        var c : Bool •
            call rp(m, a[i], c);
            if c then m := a[i];
        i := i + 1
    od
```

Having defined *findmax*, we can now use it in various ways. For instance, suppose we need to find the spread (the difference between the minimnm and the maximum) of an array of integers. We can achieve this — admittedly not very efficiently — by calling *findmax* twice, passing different procedure parameters to find the maximum and minimum, and then subtracting one result from the other. Suppose two procedure constants are defined as follows:

$$\begin{array}{l}\textbf{procedure } lte(\textbf{value } x, y : int, \textbf{result } b : Bool) \\ \quad \widehat{=}\ b := (x \leq y) \\ \textbf{procedure } gte(\textbf{value } x, y : int, \textbf{result } b : Bool) \\ \quad \widehat{=}\ b := (x \geq y)\ .\end{array}$$

These can then be passed to *findmax*:

$$\begin{array}{l}spread : int \\ as : array[0..N]\ of\ int \\ spread := max(as) - min(as) \\ \sqsubseteq \\ \textbf{var } mx, mn : int \bullet \\ \quad mx : [mx = max(as)]; \qquad (1) \\ \quad mn : [mn = min(as)]; \qquad (2) \\ \quad spread := mx - mn\end{array}$$

(1) $\sqsubseteq$ *findmax*(as, lte, mx)
(2) $\sqsubseteq$ *findmax*(as, gte, mn)

Looking in a little more detail at the refinement of (1), we have the following:

$$\begin{array}{ll}(1) = & mx : [\forall j : 0\,..\,N \bullet as[j] \leq mx \wedge mx \textbf{ in } as] \\ \sqsubseteq & \textbf{var } lp : \textbf{procedure } (\textbf{value } x, y : int, \textbf{result } b : Bool) \bullet \\ & \quad lp : [b : [b \Leftrightarrow x \leq y] \sqsubseteq lp]; \\ & \quad mx : [b : [b \Leftrightarrow x \leq y] \sqsubseteq lp, \forall j : 0\,..\,N \bullet as[j] \leq mx \wedge mx \textbf{ in } as] \\ \sqsubseteq & findmax(as, lte, mx)\end{array}$$

using the code expansions for **value** and **result** substitutions for procedure parameters.

7.3 An iterator procedure

Having set up all the machinery of procedural parameters, we now consider iterators over sequences, recalling that an **it..ti** construct over a sequence takes the following form:

$$\begin{array}{l}\textbf{it } s \textbf{ into } r \textbf{ with} \\ \quad \langle\,\rangle \ \rightarrow\ r := x \\ \quad [\!]\ a{:}as \ \rightarrow\ r := f(a, as) \\ \textbf{ti}\ .\end{array}$$

Our aim is to encapsulate this construct within a procedure, which will then form part of a library module specifying the behaviour of sequences:

module *Seq*
var
  s : seq A
procedure *seqiter*(...) $\widehat{=}$ ...

The important gaps remaining in this definition are the parameters to be passed to *seqiter* and the procedure body. The **it..ti** construct above clearly has three important parts which need to be passed as parameters of some sort. The first (and the easiest) is the variable r in which the result is to be stored. This is passed to *seqiter* as a result parameter, and we suppose it has type X. The other two parameters correspond to the two branches of the iteration -- we must get across the action to be taken if the sequence is empty, and if it is non-empty. We use procedures for both of these parameters. Dealing first with the branch for the empty sequence, we pass (by value) a procedure,

ep : **procedure** (**result** $er : X$) ,

which has a single result parameter er, which will store the value x.

For the non-empty brauch, the procedure passed, cp, must take two value parameters, for the first element of the sequence and the value of the iteration applied to the remainder of the sequence. It also stores its output in a result parameter. So we have

cp : **procedure** (**value** $a : A, as : X$; **result** $cr : X$) .

Note that the type of the second value parameter is X: this parameter represents the *result* of the iteration on the tail of the sequence (as), rather than as itself. This simply reflects the way the corresponding branch of the **it..ti** construct was defined.

Putting this all together, we can give a specification of *seqiter*:

procedure *seqiter*(
  value ep : **procedure** (**result** $er : X$),
    cp : **procedure** (**value** $a : A, as : X$; **result** $cr : X$);
  result $r : X$) $\widehat{=}$
  it s **into** r **with**
    $\langle\,\rangle \rightarrow ep(r)$
  $[]$ $a{:}as \rightarrow cp(a, as, r)$
  ti

By the definition of **it..ti** (see *sequence iterator* 5.1), we can express the body

of *seqiter* as a recursive procedure $I(s, r)$, where

procedure I(**value** s : seq X, **result** $r : X$) $\widehat{=}$
  if s **is**
    $\langle\,\rangle \longrightarrow ep(r)$
  $[]\ a{:}as \longrightarrow$ **var** $l \bullet$
      $I(as, l)$;
      $cp(a, l, r)$
  fi

Looking back at the examples of iterators at the end of Section 5.2, we can now express them as calls to *seqiter*. The first example obtained the sum of a sequence:

it s **into** r **with**
  $\langle\,\rangle \longrightarrow r := 0$
  $[]\ n{:}ns \longrightarrow r := n + ns$
ti

As a call to *seqiter*, this would be

seqiter((**result** $er : \mathbb{N} \bullet er := 0$),
  (**value** $a, as : \mathbb{N}$; **result** $cr : \mathbb{N} \bullet cr := a + as$),
  r
) .

Similarly, the length of a sequence was obtained by

it s **into** r **with**
  $\langle\,\rangle \longrightarrow r := 0$
  $[]\ n{:}ns \longrightarrow r := 1 + ns$
ti ,

which we can now write as

seqiter((**result** $er : \mathbb{N} \bullet er := 0$),
  (**value** $a, as : \mathbb{N}$; **result** $cr : \mathbb{N} \bullet cr := 1 + as$),
  r
) .

These results can be seen as particular cases of the following law, which is easily obtained by combining the definition of *seqiter* above with *assignment iterator* 5.2.

Law 7.4 *assignment seqiter*

If the value to be assigned to a variable is formed by the application of a catamorphism to a sequence, then the whole assignment can be implemented by a call to *seqiter*.

$$
\begin{array}{l}
\quad r := (\!|f, g|\!)\ s \\
\sqsubseteq \\
\quad \mathit{seqiter}(\ \ (\textbf{result}\ er \bullet er := f), \\
\qquad\qquad (\textbf{value}\ a, as; \textbf{result}\ cr \bullet cr := g(a, as)), \\
\qquad\qquad r \\
\qquad\ \ )\ .
\end{array}
$$

The types of the parameters of the procedure parameters in the call to *seqiter* are as follows: er, as and cr have the same type as the overall result r, and a has the same type as each element of the sequence s.

7.4 Merging iterators

In Section 7.1, when we gave definitions for procedures as parameters, we defined the meanings of both value and result substitutions for parameters of procedure type, but our example in Section 7.2 used only the value substitution. We now remedy this by showing how to form an iterator combinator — a higher-order procedure which basically merges two iterators to form another. This combinator uses result parameters of procedure type.

In order to motivate this construction, consider the following problem: suppose that we have stored a set of values in an array, and that we need to calculate both the sum of the values and the sum of their squares, in order to perform some statistical calculation. A naive program would be:

$$
\begin{array}{l}
\textbf{it}\ s\ \textbf{into}\ \mathit{sum}\ \textbf{with} \\
\quad \langle\,\rangle \ \ \longrightarrow\ \mathit{sum} := 0 \\
\quad [\!]\ n{:}ns\ \longrightarrow\ \mathit{sum} := n + ns \\
\textbf{ti}; \\
\textbf{it}\ s\ \textbf{into}\ \mathit{sqsum}\ \textbf{with} \\
\quad \langle\,\rangle \ \ \longrightarrow\ \mathit{sqsum} := 0 \\
\quad [\!]\ n{:}ns\ \longrightarrow\ \mathit{sqsum} := n^2 + ns \\
\textbf{ti}
\end{array}
$$

However, it would clearly be more efficient to make just a single pass over the sequence, producing both of the required values in one go. The iteration will then take the following form:

$$
\begin{array}{l}
\textbf{it}\ s\ \textbf{into}\ \mathit{sum}, \mathit{sqsum}\ \textbf{with} \\
\quad \langle\,\rangle \ \ \longrightarrow\ \mathit{sum}, \mathit{sqsum} := 0, 0 \\
\quad [\!]\ n{:}ns\ \longrightarrow\ \mathit{sum}, \mathit{sqsum} := n + ns.1, n^2 + ns.2 \\
\textbf{ti}
\end{array}
$$

There are various points to note about this:

- instead of a single result parameter, we now have a pair; looking back at the definition of the iterator as a recursive procedure (*sequence iterator*

5.1), we can see that this means simply that the recursive procedure now has two result parameters, and that two local variables need to be defined to store the intermediate results;

- in the second branch of the iterator, which corresponds to the case where the sequence is non-empty, we have to refer to the result of the iteration on the tail of the sequence: in the simpler iterators above, we were able to use *ns*, but here *ns* is a pair of values so we have to use the projections *ns*.1 and *ns*.2 in the assignment of the second branch.

So our aim is now to define an iterator combinator, which takes as input two iterators over a sequence — in the form of the procedures which form their branches — and produces as output another iterator over a sequence, the effect of which is similar to a parallel combination of the two inputs. The output iterator will also be produced in the form of the two procedures which form its branches. Thus we are aiming to replace a program of the form

```
var rp1, rp2 : procedure ... •
    seqiter(ep1, cp1, r1);
    seqiter(ep2, cp2, r2)
```

by a program of the form

```
mergeiter(ep1, cp1, ep2, cp2, rp1, rp2);
seqiter(rp1, rp2, (r1, r2)) ,
```

where *rp*1 and *rp*2 are the two procedures output from the iterator combinator *mergeiter*.

It now remains to define the combinator *mergeiter*. As mentioned above, the **value** parameters are the procedures which form the branches of the first iterator (*ep*1 and *cp*1) and the second iterator (*ep*2 and *cp*2), while the result

parameters (*rep* and *rcp*) form the branches of the resulting combined iterator.

```
procedure mergeitcr
    (value  ep1 : procedure (result er : X)
            cp1 : procedure (value a : A, as : X; result cr : X)
            ep2 : procedure (result er : Y)
            ep2 : procedure (value a : A, as : Y; result cr : Y)
     result rep : procedure (result r : X × Y)
            rcp : procedure (value a : A, as : X × Y:
                             result r2 : X × Y)
    )
≙
rep :⊒ ||[ var l, m •
           ep1(l);
           ep2(m);
           r := (l, m)
        ]| ;
rcp :⊒ ||[ var n, o •
           cp1(a, as.1, n);
           cp2(a, as.2, o);
           r2 := (n, o)
        ]||
```

The observant reader may remember a similar example towards the end of Chapter 5, where we used the so-called banana-split law to convert two iterators into a single iterator. The technique used here is very similar, except that we can use procedures to encapsulate the functions required, and we can use *mergeiter* to generate automatically the procedures needed for the combined iterator.

We recall that the banana-split law gave us the equivalence of an assignment of the form

$$x, y := (\!|f1, g1|\!)\, s, (\!|f2, g2|\!)\, s$$

and one of the form

$$x, y := (\!|f1 \vartriangle f2\ ,\ g1 \cdot (id_A \times \pi_1) \vartriangle g2 \cdot (id_A \times \pi_2)|\!)\, s \quad , \qquad (*)$$

where $\vartriangle$ is the join operator — $(f \vartriangle g)x = (f\, x, g\, x)$ — and the π_i are projections.

By *assignment seqiter* 7.4, this second assignment $(*)$ can be implemented by a call to *seqiter*. But our major interest lies in the parameters to this call.

Returning to our example, we can write the original specification as

$$sum, sqsum := (\!|f1, g1|\!)\, s, (\!|f2, g2|\!)\, s \quad ,$$

where $f1$ and $f2$ are both the constant function which returns zero, while $g1$ and $g2$ are given by

$$g1(x, y) = x + y$$
$$g2(x, y) = x^2 + y \quad .$$

So the first parameter to *seqiter* is

$$(\textbf{result } r : \mathbb{N} \times \mathbb{N} \bullet r := (0,0)) \ .$$

The second parameter must encapsulate the second part of the catamorphism in (*). Expanding the rather complicated function gives

$$(\textbf{value } a : \mathbb{N}, as : \mathbb{N} \times \mathbb{N}; \textbf{result } r2 : \mathbb{N} \times \mathbb{N} \bullet \\ r2 := (a + as.1, a^2 + as.2)) \ .$$

We have thus shown that

$$sum, sqsum := (\!|f1, g1|\!)\ s, (\!|f2, g2|\!)\ s$$

$$\sqsubseteq \quad \text{"}assignment\ seqiter\ 7.4\text{"} \\ seqiter(\ (\textbf{result } r : \mathbb{N} \times \mathbb{N} \bullet r := (0,0)), \\ (\textbf{value } a : \mathbb{N}, as : \mathbb{N} \times \mathbb{N}; \textbf{result } r2 : \mathbb{N} \times \mathbb{N} \bullet \\ r2 := (a + as.1, a^2 + as.2)), \\ (sum, sqsum) \\ ) \ .$$

Now we can introduce local procedure variables, and initialise them to the required values so that they can be used as parameters to *seqiter*:

$$\sqsubseteq \quad \textbf{var } lep : \textbf{procedure } (\textbf{result } r : \text{seq } \mathbb{N} \times \text{seq } \mathbb{N}) \\ lcp : \textbf{procedure } (\textbf{value } a : \mathbb{N}, as : \text{seq } \mathbb{N} \times \text{seq } \mathbb{N}; \\ \textbf{result } r2 : \text{seq } \mathbb{N} \times \text{seq } \mathbb{N}) \bullet$$

$$lep, lcp :\sqsupseteq (\quad (\textbf{result } r : \mathbb{N} \times \mathbb{N} \bullet r := (0,0)), \\ (\textbf{value } a : \mathbb{N}, as : \mathbb{N} \times \mathbb{N}; \textbf{result } r2 : \mathbb{N} \times \mathbb{N} \bullet \\ r2 := (a + as.1, a^2 + as.2)) \\ ) \ ; \\ seqiter(lep, lcp, (sum, sqsum))$$

Now, using the definition of *mergeiter* and the code expansions of value and result parametrisations given above in Section 7.1, it is possible to show that the first statement of these two can be implemented by a call to *mergeiter* with the right valne parameters, which will assign procedure values to the result parameters which can then be passed directly to *seqiter*. The details of this are gruesome and unenlightening — except as evidence that it can be done — and are therefore omitted.

To snmmarise, what we have done is to define *mergeiter*, and then to show that

our original problem can be implemented by the following program:

$$
\begin{array}{l}
\textbf{var } lep : \textbf{procedure } (\textbf{result } r : \text{seq } \mathbb{N} \times \text{seq } \mathbb{N}) \\
\quad lcp : \textbf{procedure } (\textbf{value } a : \mathbb{N}, as : \text{seq } \mathbb{N} \times \text{seq } \mathbb{N}; \\
\qquad\qquad \textbf{result } r2 : \text{seq } \mathbb{N} \times \text{seq } \mathbb{N}) \bullet \\
\quad mergeiter((\textbf{result } r : \mathbb{N} \bullet r := 0), \\
\qquad\qquad (\textbf{value } a : \mathbb{N}, as : \mathbb{N}; \textbf{result } cr : \mathbb{N} \bullet cr := a + as), \\
\qquad\qquad (\textbf{result } r : \mathbb{N} \bullet r := 0), \\
\qquad\qquad (\textbf{value } a : \mathbb{N}, as : \mathbb{N}; \textbf{result } cr : \mathbb{N} \bullet cr := a^2 + as), \\
\qquad\qquad lep, \\
\qquad\qquad lcp): \\
\quad seqiter(lep, lcp, (sum, sqsum))
\end{array}
$$

As we anticipated, this is simply a call to *mergeiter*, followed by a call to *seqitcr*, and the essence of the banana-split law for sequences has been encapsulated.

7.5 A more general law

As promised, we give here the unsimplified version of *procedure variable value and result specification* 7.3. The slight generalisation allowed here is that the value parameter is allowed in the frame of the specification in the second conjunct of the precondition (with a change to the corresponding sidecondition).

Law 7.5 *procedure variable value and result specification*

If the procedure variable pv has been declared as **procedure** (**value** v, **result** r), then we have the following refinement:

$$
w, ar : \left[\begin{array}{c} pre \\ w, v, r : [pre_1, post_1[ar\backslash r]] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(A, ar)
$$

$$
provided\ w, ar : [pre, post] \sqsubseteq w, ar : [pre_1[v\backslash A], post_1[v_0\backslash A_0]]
$$

where r does not occur in pre_1, and neither v, r nor r_0 occur in $post_1$, and A_0 is $A[w, ar\backslash w_0, ar_0]$.

7.6 Conclusion

In this chapter we have brought together the earlier work on iterators and procedure variables to show how iterators can be 'packaged' into procedures in their own right, using procedural parameters to pass information to the iterator about the actions to be taken on each branch of the data type. The definitions of substitutions for procedure parameters are very similar to those for the simple non-procedure parameters, with slight adjustments to deal with assignments to procedure variables.

Part IV

Applications

Chapter 8

Applications 1: exceptions

8.1 Introduction

While developiug theory and notation, it is sometimes all too easy to forget the original reason for the work that we are doing: our aim is to enable the development of correct programs from their specifications. This chapter and the following one are intended therefore to show how real programs are developed using the notations previonsly introduced.

There was a choice, for these chapters, hetween developing a single large program and working on several smaller ones. In the end, it was decided to follow the latter course: although the development of a large program could show how the techniques would scale-up to industrial-sized problems, there was the danger that it would not be possible, in a large development, to see the merits of the particular techniques proposed for exceptions and iterators, as they would get lost in the mass of development details. So this chapter and the following one contain several small developments, with some of them being later combined into larger programs.

The context in which these sample programs will be developed is that of IBM's Collection Class Library for C++. This is a set of C++ classes that implement commonly-used abstract data types, such as sets, maps and sequences. We give a short summary below of the main features of the Collection Class Library.[1]

Section 8.2 describes how the exception-handling mechanism introduced earlier

[1]The motivation for using the Collection Class Library is that the author was supported by IBM in the early stages of this research.

can be related to exception handling in C++ and the Collection Class Library. In general, we give 'guidelines' on how to develop C++ programs, rather than fully justified 'laws', since the justification of such laws would require a formal semantics for C++, which is beyond the scope of this thesis!

Since C++ is not an ideal language for which to develop programs in the refinement calculus style, we have found it useful to introduce a few abbreviations to make the jump from guarded commands to C++ slightly easier. These are concerned with the relationship between C++ functions and refinement calculus procedures, and are found in Section 8.3 and Appendix A. We also give some additional laws that are used in the sample developments: rather than disrupt the developments by giving the laws 'in-line', they are collected in this section.

After all this extra notation, we are finally able to show how the individual data types from the Collection Class Library can be specified. For this purpose, we take sequences as our example, giving a specification which consists of a state model, followed by descriptions of the many operations provided in the library to manipulate sequences. Several sample programs are developed.

Chapter 9 will show how the iterator mechanism we have described can be used with the iterator mechanism of the Collection Class Library. Further examples will be developed.

An introduction to the Collection Class Library

IBM's Collection Class Library for C++ is a set of C++ classes that implement commonly used abstract data types, including sets, maps, sequences, relations, trees, stacks, bags, queues and priority queues. Most collection classes exist in several forms, depending on whether the collection is sorted, whether elements can be accessed by keys, whether there is an equality relation defined for elements and whether elements must be unique or if multiple occurrences are allowed.

For each collection class, many operations are defined: each takes the form of a C++ function, some having side-effects, some giving return values, and some both. If certain preconditions are not met when the operation is called, an exception may be raised. The informal specification [22] defines what the possible exceptions are for each function, but not the exact circumstances in which they are raised. This important omission is rectified in the formal specifications later in this chapter, for every relevant exception except for the *lOutOfMemory* exception. This exception can be raised by any of the variations of the *add* operation when the operating system is short of memory. It has to be treated differently since we cannot describe, at our level of abstraction, exactly when this will happen. Having experimented with a specification which merely stated that the *add* operations could raise this exception non-deterministically, under any circumstances, it was instead decided to omit it completely: the non-deterministic specification was not particularly helpful since it didn't describe when the exception would be raised, and it made developing programs which

used the *add* operation particularly difficult. While not ideal, the pragmatic solution of not mentioning the *IOutOfMemory* exception at all means that is much more practical to use the refinement calculus to argue about the correctness of an implementation, including the circumstances under which any other exception may be raised. However, the developer must remember to use another form of argument to reason about the possibility of the operating system running out of memory, and what actions should be taken if that possibility arises. This seems to be a case where formalism needs to be combined with pragmatism.

The Collection Class Library also has a built-in method of indirectly accessing the elements of a collection: the user of a collection class, once he has declared an instance of the class, can declare a 'cursor', which is then associated with that particular collection. This cursor can be used to access the elements of the collection: there are several operations which take a cursor as input, or return a cursor as output, and, for ordered collections, there are operations which access the elements in cursor order. For example, *addAsNext* takes an element and a cursor as input; provided that the cursor is valid and associated with the collection that is being operated on, the element is added to the collection at the position after that pointed to by the cursor. Although the cursor mechanism provides a reasonably efficient way of programming, by removing the need to copy or move possibly large pieces of data, it suffers from one major drawback:[2] whenever the collection is altered in any way, by the addition or removal of elements, all of the cursors are invalidated — that is, the programmer cannot rely on their still pointing to the same elements, or even to any elements at all. However, the description of the sequence library component given below does not deal at all with cursors: a decision was made that the cursor behaviour was not sufficiently linked with either exceptions or iterators to merit its inclusion. Although the operations which involved cursors could raise exceptions, there was no significant difference in the use of exceptions between these operations and non-cursor operations. Inclusion of cursors would have meant a significantly larger specification, but without significantly more interesting material.

Cursors can be used to program iterations over a data structure, but there is also a more abstract mechanism, the *allElementsDo* operation. Both of these are described in the next chapter.

Several of the operations found in the Collection Class Library also exist in alternative versions where, instead of a value being supplied to or returned by the operations, a pointer to the value is used. We have not described any of these alternative versions, since the focus of our interest is exceptions and iterators.

[2] This is the case in the current implementation, at least future versions may change this.

8.2 Exceptions and the Collection Class Library

In Chapter 3, we introduced a simple exception mechanism and showed how we could extend the idea of weakest precondition semantics to give a meaning to exceptional termination, and to justify the laws we stated about the constructs. In Chapter 4, we extended this mechanism by considering named exceptions, using procedures to give a form of exception handling, whereby different actions could he associated with different exceptions. Our task now is to show how we can nse these ideas to develop C++ programs which use the exception handling facilities of the Collection Class Library. Since the library makes direct use of the exception mechanism of the C++ language, we look first at this, before deciding how much of the C++ notation we will model with our exception handling mechanism. Our aim is always to develop programs rigorously using the refinement calculus, and then to translate them into the target programming language — or a subset of it. This translation process defines our view of a 'safe' subset of the programming language, since, although we cannot possibly guarantee to be able to develop formally every conceivable program in the language, we can guarantee that any program that is the end-product of a refinement calculus development and a translation will meet its specification.

Exception handling in C++

The C++ exception handling mechanism allows a programmer to recognise when a function has been called in an nnusual situation, and to pass control back to the caller of the function. The caller is then able to handle the exception in an appropriate way. The language constructs which implement this exception handling are:

- `throw` expressions:
- `try` blocks;
- `catch` blocks.

In the body of a function, the programmer can signal an unusual situation with a `throw` expression. This expression can contain information to be passed back to the caller, perhaps an indication of which object caused the exception. Alternatively it might just be a signal that the unusual situation has arisen. The concrete syntax consists of the keyword `throw`, followed by an assignment-expression. In the context of a declaration of *aobj* as an element of a class A, the two possible forms of a throw statement might be:

```
throw (aobj);
throw IInvalidCursor;
```

The first of these shows information being passed back to the caller, in this case the variable name `aobj`. In the second example, a constant valne is passed, which is just the name of the exception

Try blocks and catch blocks are used together to show the scope of definition of some handling routines, and the contents of those rontines. respectively. Thus a single try block is followed immediately by one or more catch blocks:

```
try{
   statements
   }
catch(ex1){
   statements
   }
catch(ex2){
   statements
   }
```

The statements enclosed in braces after the `try` keyword are the scope of the succeeding catch blocks: if a function which is called in these statements throws an exception which matches any of the catch blocks (`ex1` or `ex2` above), then the corresponding handler is executed. If the catch block terminates normally, control passes to the statement after the final handler.

At the start of each catch block, there is a parenthesised expression which declares the type of object that the exception handler may catch, and optionally a variable name to identify the object thrown within the succeeding code. The rules to determine which catch block is executed are:

- if the object thrown matches the type of the catch expression of the first block, control passes to tbat block;
- if the object thrown does not match the type of the catch expression of the first block, then subsequent blocks are searched for a matching type;
- the special catch expression `catch(...)` will match any thrown expression (and should therefore only appear in the last of a sequence of catch blocks);
- if no match is found, the search is continued in all enclosing try blocks and then in the caller of the current function;
- if no match is found after all this, a call to the `terminate()` function is made.

C++ exceptions and the refinement calculus

In its very simplest form, C++ exception handling is not very difficult to incorporate into the refinement calculus development method, as we bave extended

it to cover exceptional postconditions. The restriction that we put on the general scheme outlined above is that we don't allow objects to be passed to catch blocks.[3] Instead we insist that the catch expression which determines which exceptions are caught by a block should consist of a type name, and that type should be a single element type (formed from a class with no data or methods). So we might have:

```
Exception1 class { /* ... no methods ... */ }

try {
      ...
      throw Exception1;
      ...
    }
catch (Exception1) { ... }
```

Another restriction that we place on the general C++ exception mechanism is that we do not cover the case where the catch argument is a public base class (ie subtype) of the thrown class object: we insist that the catch argument should be exactly equal to the thrown object.

Having described the very restricted form of C++ exception handling which we are going to use, we are uow in a position to show how we can develop programs in our extended version of the refinement calculus, and then transliterate them into C++.

Using the rules given in previous chapters for **exits**, we aim to develop a program which has the form

$$
\begin{array}{l}
[\![\ \textbf{handler}\ E1 \mathrel{\widehat{=}} aaa \\
\qquad\qquad\ E2 \mathrel{\widehat{=}} bbb \\
\qquad\qquad\ \vdots \\
\qquad\qquad\ En \mathrel{\widehat{=}} nnn \bullet \\
\qquad zzz \\
]\!]
\end{array}
$$

Assuming all the subprograms (*aaa* to **nnn**, and *zzz*) are code, we would be finished, since we can transliterate this into C++ as:

```
class E1 { /* ... */ };
class E2 { /* ... */ };
 ...
class En { /* ... */ };

void main() {
```

[3] As was mentioned in Chapter 4, we could model this, but the Collection Class Library exception mechanism does not include this feature, so we do not need to consider it.

```
  try { zzz }
  catch (E1) { aaa }
  catch (E2) { bbb }
  ...
  catch (En) { nnn }
}
```

So each exception `Ei` is declared as a class with no members, and the handlers become catch blocks while the main program body `zzz` becomes a try block.

A simple development with exceptions

In this section, we give a formal development of a C++ program which contains exceptions. The example is very closely based on a sample program from the IBM C++ Reference Manual [22]. The original program is shown in Figure 8.1 overleaf, but we will slightly alter some of the input and output statements to simplify the development.

Our starting point is the following specification:

$$\begin{array}{l}\textbf{var } n : \mathbb{N} \\ \quad o : \mathbb{R} \\ \quad r : good \mid bad \\ \quad Z \mathrel{\widehat{=}} n = 0 \wedge r = bad \\ \quad NZ \mathrel{\widehat{=}} n \neq 0 \wedge o = 1/n \wedge r = good \\ o, r : [Z \vee NZ]\end{array}$$

If n is non-zero, the output o is to be set to the reciprocal of n. The response code r indicates whether or not the output is a valid reciprocal.

The development starts by introducing an exception block and duplicating the normal postcondition as an exceptional postcondition:

$$\begin{array}{ll}\sqsubseteq & \text{“}\textit{exceptional specification}\ 3.4\text{”} \\ & [\![\\ & o, r : [true, Z \vee NZ > Z \vee NZ] \\ &]\!]\end{array} \quad \triangleleft$$

Using the law for sequential composition, we can split this into two:

$$\begin{array}{lll}\sqsubseteq & \text{“}\textit{sequential composition}\ 3.13\text{”} & \\ & o, r : [true, n \neq 0 > Z \vee NZ]\,; & (1) \\ & o, r : [n \neq 0, Z \vee NZ > Z \vee NZ] & (2)\end{array}$$

The second branch is implemented by choosing the non-exceptional route:

$$\begin{array}{lll}(2) \sqsubseteq & \text{“}\textit{take normal branch}\ 3.5\text{”} & \\ & o, r : [n \neq 0, Z \vee NZ] & \\ \sqsubseteq & o, r : [n \neq 0, NZ] & \\ = & o, r : [n \neq 0, n \neq 0 \wedge o = 1/n \wedge r = good] & \\ \sqsubseteq & o, r := 1/n, good &\end{array}$$

```
/*************************************************************
*
The following example illustrates the basic use of try, catch,
and throw. The program prompts for a numerical user input and
determines the input's reciprocal. Before it attempts to print
the reciprocal to standard output, it checks that the inpnt
value is nonzero, to avoid a division by zero. If the input is
zero, an exception is thrown, and the catch block catches the
exception. If the input is nonzero, the reciprocal is printed
to standard output.
                                                            *
*************************************************************/

#include <iostream.h>
#include <stdlib.h>
class IsZero { /* ... */ };
void ZeroCheck( int i )
{
        if (i==0)
              throw IsZero();
}
void main()
{
        double a;

        cout << "Enter a number: ";
        cin >> a;
        try
        {
             ZeroCheck( a );
             cout << "Reciprocal is " << 1.0/a << endl;
        }
        catch ( IsZero )
        {
             cout << "Zero input is not valid" << endl;
             exit(1);
        }
        exit(0);
}
```

Figure 8.1: IBM's original program showing the use of exceptions

The first branch is developed by introducing an alternation, before deciding whether to aim for the normal or exceptional postcondition.

$$
\begin{array}{ll}
(1) \sqsubseteq & \textbf{if } n = 0 \rightarrow \\
& \quad o, r : [n = 0, n \neq 0 > Z \vee NZ] \qquad (3) \\
& [\!] \; n \neq 0 \rightarrow \\
& \quad o, r : [n \neq 0, n \neq 0 > Z \vee NZ] \qquad (4) \\
& \textbf{fi}
\end{array}
$$

The first branch of the alternation is implemented by choosing the exceptional postcondition -- any attempt to develop the normal postcondition would lead to a miracle.

$$
\begin{array}{ll}
(3) \sqsubseteq & \text{“}take\ exceptional\ branch\ 3.6\text{”} \\
& o, r : [n = 0, Z \vee NZ]; \qquad \triangleleft \\
& \textbf{exit} \\
\sqsubseteq & o, r : [n = 0, Z] \\
\sqsubseteq & r := bad
\end{array}
$$

The second branch of the alternation is easily implemented by choosing the normal postcondition, and noticing that we are already finished!

$$
\begin{array}{ll}
(4) \sqsubseteq & \text{“}take\ normal\ branch\ 3.5\text{”} \\
& o, r : [n \neq 0, n \neq 0] \\
\sqsubseteq & \textbf{skip}
\end{array}
$$

Collecting the code together we get

$$
\begin{array}{l}
o, r : [Z \vee NZ] \\
\sqsubseteq \; [\!] \\
\qquad \textbf{if} \quad n = 0 \rightarrow r := bad; \textbf{exit} \\
\qquad [\!] \quad n \neq 0 \rightarrow \textbf{skip} \\
\qquad \textbf{fi}; \\
\qquad o, r := 1/n, good \\
[\!]
\end{array}
$$

We can re-structure this slightly by using a handler:

$$
\begin{array}{l}
\sqsubseteq \quad \text{“}introduce\ handler\ 4.4\text{”} \\
\qquad [\!] \; \textbf{handler } IsZero \mathrel{\widehat{=}} r := bad \bullet \\
\qquad\quad \textbf{if} \quad n = 0 \rightarrow \textbf{raise}(IsZero) \\
\qquad\quad [\!] \quad n \neq 0 \rightarrow \textbf{skip} \\
\qquad\quad \textbf{fi}; \\
\qquad\quad o, r := 1/n, good \\
\qquad [\!]
\end{array}
$$

We now translate to C++, using the translation guidelines introdnced above, and include a few extra statements to reveal the results of the calculation on the console, and to set return codes.

```
#include <iostream.h>
#include <stdlib.h>
class IsZero { /* ... */ };
void main()
{
        double n;

        cout << "Enter a number: ";
        cin >> n;
        try
        {
             if (n==0) throw IsZero();
             o = 1.0/n; r = good;
             cout << "r is " << r << endl;
             cout << "o is " << o << endl;
        }
        catch ( IsZero )
        {
             r = bad;
             cout << "r is " << r << endl;
             exit(1);
        }
        exit(0);
}
```

8.3 Additional notation and laws

The end-product of the refinement calculus development method is a program in the language of Dijkstra's guarded commands [17]. It is relatively straightforward to translate such a program into a language such as Pascal or Modnla-2 for compilation and execution. However, the Collection Class Library is designed for the C++ language, and the mapping from guarded commands to C++ is not quite so simple. We have taken a pragmatic view where possible, since the focus of our work is on exceptions and iterators in the refinement calculus, not on the particular problems caused by the choice of C++ as the target programming language.

Appendix A contains details of the mismatch between C++ functions and the usual procedures of the refinement calculus. However, this mismatch occurs in only one place in the case studies in this chapter, and so the new notation is not very significant.

More importantly, we give some additional laws of the refinement calculus, which will be used in the example developments.

Additional laws

The standard refinement calculus development technique is to manipulate the specification, using the laws of the calculus, until we have a program consisting of executable code. In our case, because we intend to use the Collection Class Library, we can use the copy rule to replace a procedure body with a procedure call, so we will be aiming either towards executable code, or towards fragments of program which match (with suitable parametrisation) the bodies of procedures defined in the library modules. As we follow this development method, it is clear that the presence of exceptions in the specifications of the collection class operations means that we need to use the laws from Chapter 4 which show how to manipulate **exit**, **raise** and else constructs in the context of the standard programming constructs. We also need to perform some development steps which seem strange at first — their purpose is to change the program fragment we are currently working on so that it corresponds to a procedure body. This gives rise to the need for some additional laws, which are easy to verify using weakest preconditions.

We need another law to show how the else construct interacts with sequential composition.

Law 8.1 *else distribution*

$$\begin{array}{ll} (aaa:\ bbb) > ccc & \\ \sqsubseteq \quad aaa;\ (bbb > ccc) & \textit{provided } ccc \sqsubseteq aaa;\ ccc \end{array}$$

We can implement a specification statement whose postcondition consists of a disjunction, by introducing an exception block and taking the disjuncts as the normal and exceptional postconditions of an extended specification statement inside the block.

Law 8.2 *disjunction-else*

$$\begin{array}{l} w : [\alpha, \beta \vee \gamma] \\ = \quad [\![\\ \qquad\quad w : [\alpha, \beta > \gamma] \\ \quad]\!] \end{array}$$

We can turn a specification into a choice between guarded commands, using any program at all in the new branch, if we know that the branch will never be taken:

Law 8.3 *superfluous choice*

$$\begin{array}{ll} w : [\alpha, \beta] & \\ \sqsubseteq \quad \gamma \ \rightarrow w : [\alpha, \beta] & \\ \quad [\!] \ \neg\gamma \ \rightarrow aaa & \\ & \textit{provided } \alpha \Rightarrow \gamma \end{array}$$

It is sometimes convenient to transform a guarded specification statement into one where the guard has been absorbed:

Law 8.4 *absorb guard*

$$\begin{aligned} & \alpha \rightarrow w : [\beta, \gamma] \\ = \quad & w : [\alpha \Rightarrow \beta, \alpha[w \backslash w_0] \wedge \gamma] \end{aligned}$$

provided w_0 *is not free in* α

8.4 A collection class specification

Having set up all the extra notational machinery that we need, we are now able to give a specification of a sample collection class, for which we use sequences. We first describe all of the operations on sequences, and then give a small example to show how the new pieces of notation introduced above can be used. As was mentioned above, there is one omission from the description below: we do not concern ourselves with the `IOutOfMemory` exception. This is an exception which can be raised by any of the *add* operations, reflecting the possibility that the operating system might report that it has run out of memory. We would not be able to describe the exact circumstances under which this might happen, and including the possibility of running out of memory makes the specification unnecessarily complicated.

The sequence class

The only state variable is the sequence itself.[4]

module *ISeq*[*Element*] $\widehat{=}$
var
s : seq[*Element*]

There are four different ways to *add* an element to the sequence. In the simplest case, the new element is added to the end of the sequence, and a return code indicates whether the operation has completed successfully.[5]

procedure $add(e : Element, \textbf{result } r : Boolean) \mathrel{\widehat{=}}$
$s, r := s \frown \langle e \rangle, true$

We have explicit operations to add the new element to the beginning or end of the sequence.

procedure $addAsFirst(e : Element) \mathrel{\widehat{=}}$
$s := \langle e \rangle \frown s$

procedure $addAsLast(e : Element) \mathrel{\widehat{=}}$
$s := s \frown \langle e \rangle$

[4] As in Z, our sequences are indexed from 1 to the length of the sequence.

[5] For brevity, we assume that all parameters are passed by value, unless explicitly noted otherwise.

We can also specify the position at which we want the element to be added. If this position is invalid, an exception is raised. Otherwise, the uew element is added at the specified position, with subsequent elements being 'shunted down' as required. ($\uparrow$ and $\downarrow$ are functions which return parts of a sequence: $s \uparrow n$ gives the first n elements of a sequence s, and $s \downarrow n$ returns the seqnence with the first n elements removed.)

$$\begin{array}{l}\textbf{procedure } \mathit{addAtPosition}(i : \mathbb{N}, e : \mathit{Element}) \mathrel{\widehat=} \\ \quad 1 \le i \le \#s + 1 \quad \rightarrow s := s \uparrow (i-1) \frown \langle e \rangle \frown s \downarrow (i-1) \\ [] \ (i = 0) \vee (i > \#s + 1) \ \rightarrow \textbf{raise}(\mathit{IPosttionInvalidException})\end{array}$$

The *anyElement* operation returns a randomly-chosen element of the sequence. provided that the sequence is not empty.

$$\begin{array}{l}\textbf{procedure } \mathit{anyElement}(\textbf{result } e : \mathit{Element}) \mathrel{\widehat=} \\ \quad s \neq \langle\rangle \ \rightarrow e : [e \in \operatorname{ran} s] \\ [] \ s = \langle\rangle \ \rightarrow \textbf{raise}(\mathit{IEmptyException})\end{array}$$

We can examine the element at any particular position in the sequence.

$$\begin{array}{l}\textbf{procedure } \mathit{elementAtPosition}(i : \mathbb{N}, \textbf{result } e : \mathit{Element}) \mathrel{\widehat=} \\ \quad 1 \le i \le \#s \quad \rightarrow e := s\,i \\ [] \ (i = 0) \vee (i > \#s) \ \rightarrow \textbf{raise}(\mathit{IPosttionInvalidException})\end{array}$$

We can look at the first element of the sequence, provided that the sequence is not empty.

$$\begin{array}{l}\textbf{procedure } \mathit{firstElement}(\textbf{result } e : \mathit{Element}) \mathrel{\widehat=} \\ \quad s \neq \langle\rangle \ \rightarrow e := s\,1 \\ [] \ s = \langle\rangle \ \rightarrow \textbf{raise}(\mathit{IEmptyException})\end{array}$$

There are several enquiry operations for sequences. For compatibility with other classes in the Collection Class Library, the operations *isBounded* and *isFull* are provided: since this class is not bounded, the former operation always returns False. Similarly the sequence can never he full. The *isEmpty* operation determines whether the sequence is currently empty.

$$\begin{array}{l}\textbf{procedure } \mathit{isBounded}(\textbf{result } r : \mathit{Boolean}) \mathrel{\widehat=} \\ \quad r := \mathit{false}\end{array}$$

$$\begin{array}{l}\textbf{procedure } \mathit{isFull}(\textbf{result } r : \mathit{Boolean}) \mathrel{\widehat=} \\ \quad r := \mathit{false}\end{array}$$

$$\begin{array}{l}\textbf{procedure } \mathit{isEmpty}(\textbf{result } r : \mathit{Boolean}) \mathrel{\widehat=} \\ \quad r := (s = \langle\rangle)\end{array}$$

We can access the last element of the sequence, provided that the sequence is not empty.

$$\begin{array}{l}\textbf{procedure } \mathit{lastElement}(\textbf{result } e : \mathit{Element}) \mathrel{\widehat=} \\ \quad s \neq \langle\rangle \ \rightarrow e := s\,(\#s) \\ [] \ s = \langle\rangle \ \rightarrow \textbf{raise}(\mathit{IEmptyException})\end{array}$$

Since this class provides a data type of unbounded sequences, the operation which might be expected to return the maximum number of elements that the sequence can contain will always raise an exception.

procedure *maxNumberOfElements*(**result** $n : \mathbb{N}$) $\widehat{=}$
raise(*INotBoundedException*)

We can determine the current size of the sequence.

procedure *numberOfElements*(**result** $n : \mathbb{N}$) $\widehat{=}$
$n := \#s$

There are two forms of 'multiple removal' operations: the first removes all of the elements of the sequence unconditionally, and so the sequence becomes empty. The second form removes all those elements of the sequence which satisfy a given property. Here we model this property as a set of elements, and use the *squash* function to 'close up' the gaps in the sequence which are caused by the removal of those elements in the set.

procedure *removeAll*1 $\widehat{=}$
$s := \langle\rangle$

procedure *removeAll*2($b : \mathbb{P}$ *Element*) $\widehat{=}$
$s := squash(s \rhd b)$

There are three ways to remove a single element from the sequence: by specifying which element is to be removed, or by taking the first or last element. An exception is raised if the position is not valid, or if the sequence is empty.

procedure *removeAtPosition*($i : \mathbb{N}$) $\widehat{=}$
$1 \leq i \leq \#s \quad \rightarrow s := s \uparrow (i-1) \frown s \downarrow i$
$[\!]\ (i = 0) \vee (i > \#s) \rightarrow$ **raise**(*IPositionInvalidException*)

procedure *removeFirst* $\widehat{=}$
$s \neq \langle\rangle \rightarrow s := s \downarrow 1$
$[\!]\ s = \langle\rangle \rightarrow$ **raise**(*IEmptyException*)

procedure *removeLast* $\widehat{=}$
$s \neq \langle\rangle \rightarrow s := s \uparrow (\#s - 1)$
$[\!]\ s = \langle\rangle \rightarrow$ **raise**(*IEmptyException*)

Example

We now show how programs can be developed using the specification of a library module, such as the sequence module above. We follow the usual technique of using the procedure rules to replace a procedure body by a call (with the appropriate parametrisation), but some unusual development steps have to be taken because the procedure specifications above give the possibility of raising an exception. Although this example is fairly simple, it shows the techniques needed to introduce the exceptions.

We will work on a sequence of numbers, and we will expect some sort of return code:

$$s : \text{seq } \mathbf{N}$$
$$r : OK \mid TooSmall$$

The specification of our problem is

$$Spec \mathrel{\widehat=}$$
$$\textbf{if } \#s \geq 2 \rightarrow s, r := (s \uparrow \#s - 1) \downarrow 1, OK$$
$$[\!] \ \#s < 2 \rightarrow r := TooSmall$$
$$\textbf{fi}$$

If the sequence s has at least two elements, then the first and last elements are removed. Otherwise, s is left unchanged and a return code indicates that the sequence is too small.

The implementation that we are aiming at involves using *removeFirst* and *removeLast* to take off the first and last elements of s. If the sequence is empty, then obviously a call to the first operation will immediately fail. However, if s is a singleton sequence, then a call to one of these procedures will succeed and then the other will fail. So we need to store the initial value of s in a local variable, so that it can be restored later, if necessary. Thus the first development step is to introduce the local variable l, and a logical constant for the initial value of s: we also work with naked guarded commands rather than alternations, since this matches the library procedure definitions:

$$Spec$$
$$\sqsubseteq \quad \textbf{var } l : \text{seq}\,\mathbf{N} \bullet$$
$$\textbf{con } S \bullet$$
$$l := s;$$
$$\left(\begin{array}{l} \{l = S\}\, \#s \geq 2 \rightarrow s, r := (s \uparrow \#s - 1) \downarrow 1, OK \\ [\!] \ \{l = S\}\, \#s < 2 \rightarrow s, r : [r = TooSmall \wedge s = S] \end{array} \right) \qquad \triangleleft$$

The next step of the development is to introduce an exception block, and to change the non-deterministic choice into an 'else' construct:

$$\sqsubseteq \quad \text{“}choice\text{-}else\ 3.10\text{”}$$
$$[\!]$$
$$\{l = S\}\, \#s \geq 2 \rightarrow s, r := (s \uparrow \#s - 1) \downarrow 1, OK$$
$$>$$
$$\{l = S\}\, \#s < 2 \rightarrow s, r : [r = TooSmall \wedge s = S]$$
$$[\!]$$

The assignment of OK to r can be moved to the end of the exception block:

$$\sqsubseteq \quad \text{“}else\ distribution\ 4.9\text{”}$$
$$\left(\begin{array}{l} \{l = S\}\, \#s \geq 2 \rightarrow s := (s \uparrow \#s - 1) \downarrow 1 \\ > \\ \{l = S\}\, \#s < 2 \rightarrow s, r : [r = TooSmall \wedge s = S] \end{array} \right) ; \qquad \triangleleft$$

$$r := OK$$

The exceptional construct is split into two with sequential composition, using the somewhat complicated law introduced earlier (page 24).

$$\sqsubseteq \quad \text{"}\textit{sequential composition}\text{ 3.12"}$$

$$\left(\begin{array}{l} \{l = S\}\ s \neq \langle\rangle \ \rightarrow s := (s \uparrow \#s - 1) \\ > \\ \{l = S\}\ \#s < 2 \rightarrow s, r : [r = \mathit{TooSmall} \wedge s = S] \end{array}\right) ; \qquad (1)$$

$$\left(\begin{array}{l} \{l = S\}\ s \neq \langle\rangle \ \rightarrow s := (s \downarrow 1) \\ > \\ \{l = S\}\ \#s < 1 \rightarrow s, r : [r = \mathit{TooSmall} \wedge s = S] \end{array}\right) \qquad (2)$$

(The use of this law is justified by noting that

$$\{l = S\}\ \#s \geq 2 \rightarrow s := (s \uparrow \#s - 1) \downarrow 1$$

$$\sqsubseteq$$

$$\{l = S\}\ s \neq \langle\rangle \rightarrow s := (s \uparrow \#s - 1);$$
$$\{l = S\}\ s \neq \langle\rangle \rightarrow s := (s \downarrow 1)$$

and that

$$\{l = S\}\ \#s < 2 \rightarrow s, r : [r = \mathit{TooSmall} \wedge s = S]$$

$$\sqsubseteq$$

$$\{l = S\}\ s \neq \langle\rangle \ \rightarrow s := (s \uparrow \#s - 1);$$
$$\{l = S\}\ \#s < 1 \ \rightarrow s, r : [r = \mathit{TooSmall} \wedge s = S]$$

as required.)

The program now looks very promising because the two halves of the sequential composition are similar to the specifications of the *removeLast* and *removeFirst* operations, respectively. Converting each exceptional branch into an assignment, then removing the unnecessary assumptions, and slightly re-writing the guards makes the match even more clear:

$$(2) \sqsubseteq \quad s \neq \langle\rangle \ \rightarrow s := s \downarrow 1$$
$$> \ s = \langle\rangle \ \rightarrow r, s := \mathit{TooSmall}, l$$

$$(1) \sqsubseteq \quad s \neq \langle\rangle \ \rightarrow s := s \uparrow \#s - 1$$
$$> \ \#s < 2 \rightarrow r, s := \mathit{TooSmall}, l$$

$$\sqsubseteq \quad \text{"}\textit{strengthen guard}\text{"}$$
$$s \neq \langle\rangle \ \rightarrow s := s \uparrow \#s - 1$$
$$> \ s = \langle\rangle \ \rightarrow r, s := \mathit{TooSmall}, l$$

We can make the match to the library specifications complete by defining a handler for the exception which is raised when the operations are applied to an empty sequence. For this we need to return to the top level of the development:

$$
\begin{array}{l}
Spec \\
\sqsubseteq \quad \text{"}introduce\ handler\ 4.5\text{"} \\
\quad \textbf{var}\ l : \mathrm{seq}\ \mathbb{N} \bullet \\
\quad l := s; \\
\quad [\![\ \textbf{handler}\ IEmptyException \mathrel{\widehat=} r, s := TooSmall, l \bullet \\
\qquad (s \neq \langle\rangle \rightarrow s := s \uparrow \#s - 1 \\
\qquad\quad [\!]\ s = \langle\rangle \rightarrow \textbf{raise}\ IEmptyException); \\
\qquad (s \neq \langle\rangle \rightarrow s := s \downarrow 1 \\
\qquad\quad [\!]\ s = \langle\rangle \rightarrow \textbf{raise}\ IEmptyException); \\
\qquad r := OK \\
\quad]\!]
\end{array}
$$

Notice that the handler for *IEmptyException* should actually be declared twice: however the handlers are identical and so the declarations can be merged.

Finally we use the copy rule to insert calls to the *removeLast* and *removeFirst* operations, to give the following program:

$$
\begin{array}{l}
\textbf{var}\ l : \mathrm{seq}\ \mathbb{N} \bullet \\
l := s; \\
[\![\ \textbf{handler}\ IEmptyException \mathrel{\widehat=} r, s := TooSmall, l \bullet \\
\quad s.removeLast; \\
\quad s.removeFirst; \\
\quad r := OK \\
]\!]
\end{array}
$$

What's missing?

This completes our specification of the sequence class, but we should be honest and admit that there are a few parts of the commercially-available class that we have not specified:

- We have not given a specification of the iterator operation *allElementsDo*: this would not be too difficult, but we have concentrated on exceptions in this chapter. The next chapter shows how to use the *allElementsDo* operation.
- We have not described the *addAllFrom* operation, which allows the user to form a combination of two collections, by adding the elements of a second collection to the current one. It should be possible to specify this in terms of an iteration over the second collection, but there are some interesting questions: for instance, what happens if an exception is raised (perhaps because of lack of memory) midway through the iteration? The manual is very unclear on this!

- As mentioned earlier, we have not described the use of cursors, or the *IOutOfMemory* exception. We have also not described the operations which use pointers.

8.5 Sample developments

We can now give some larger sample developments which use the specification set out above.

Mapping a function

In order to show how we can use the specification of the sequence class above, we start by developing a program to transform a given text, t, a sequence of words, into another text, s, by applying an as-yet-unspecified function f to each element of t. The development will be tackled in two different ways: using ordinary sequence operations in this chapter, and using iterators in the next chapter.

The specification of the problem stipulates that the sequences should have the same length, and that f should be applied to each element of t to produce the corresponding element of s.

$$\textbf{var } s, t : ISeq[Word]$$
$$s : [(\#s = \#t) \wedge \forall j : 1..\#t \bullet s\,j = f(t\,j)]$$

The first step is to introduce a local variable and split the specification statement into an initialisation, followed by what will become a loop:

$$\begin{array}{ll} \sqsubseteq & \textbf{var } i : \mathbb{N} \bullet \\ & \text{“}I \mathrel{\widehat=} (\#s = i) \wedge (i \leq \#t) \wedge \forall j : 1..i \bullet s\,j = f(t\,j)\text{”} \bullet \\ & s, i : [true, I]; \qquad (1) \\ & s, i : [I, I \wedge i = \#t] \qquad (2) \end{array}$$

The initialisation is easily implemented as two assignments:

$$(1) \sqsubseteq \quad s, i := (\,), 0$$

and the assignment to s is implemented with a call to the *removeAll* procedure:

$$\begin{array}{ll} \sqsubseteq & s.removeAll1; \\ & i := 0 \end{array}$$

Returning to (2), we can see that the iteration is guarded by $i \neq \#t$, which we can represent as $i \neq t.numberOfElements!$, remembering[6] that this is an

[6] See Appendix A.

abbreviation for the declaration of a local variable, say $numT$, followed by a call to $t.numberOfElements$ with the result stored in $numT$, and the use of $i \neq numT$ as the real guard:[7]

$$
\begin{array}{ll}
(2) \sqsubseteq & \text{"iteration"} \\
& \textbf{do } i \neq t.numberOfElements! \rightarrow \\
& \quad s, i : \left[\begin{array}{cc} I & I \\ i \neq \#t \;\cdot\; 0 \leq \#t - i < \#t - i_0 \end{array} \right] \\
& \textbf{od}
\end{array}
\qquad \triangleleft
$$

Since it is clear that we will need to increment i, we use the *following assignment* law on the loop body:

$$
\begin{array}{ll}
\sqsubseteq & \text{"following assignment"} \\
& s : \left[\begin{array}{cc} I & I[i \backslash i+1] \\ i \neq \#t \;\cdot\; 0 \leq \#t - (i+1) < \#t - i \end{array} \right]; \\
& i := i + 1
\end{array}
\qquad (3)
$$

It is tempting to implement the first half of the loop body with a simple assignment:

$$
(3) \sqsubseteq \quad s := s \frown \langle t(i+1) \rangle \quad ,
$$

but this would lead to trouble, since the only way we can access the $(i+1)$th element of t is to use the *elementAtPosition* operation. In order to use it successfully, we need to be sure that the supplied position — $i+1$ in this case — lies within the bounds of the sequence. So here we need $1 \leq i+1 \leq \#t$. Therefore, a development that proceeded straight to an assignment as above would soon run into problems. So instead, we introduce another local variable to store the relevant element of t:

$$
\begin{array}{ll}
(3) \sqsubseteq & \textbf{var } w : Word \bullet \\
& w, s : \left[\begin{array}{cc} I & I[i \backslash i+1] \\ i \neq \#t \;\cdot\; 0 \leq \#t - (i+1) < \#t - i \end{array} \right] \\
\sqsubseteq & \text{"sequential composition"} \\
& w, s : \left[\begin{array}{cc} I & I \\ i \neq \#t \;\cdot & i \neq \#t \\ & w = t(i+1) \end{array} \right]; \qquad \triangleleft \\
& w, s : \left[\begin{array}{cc} I & \\ i \neq \#t & , I[i \backslash i+1] \\ w = t(i+1) & \end{array} \right]
\end{array}
\qquad (4)
$$

The first half of this is implemented by introdncing a nondeterministic choice. We know from the precondition that the first branch of the choice must be taken, so we have complete freedom of choice for the second branch. We choose to raise an exception so that this construct matches exactly the specification of

[7] In the body of the loop, we continue to use $\#t$, since these occurrences are not in code, and will disappear later in the development

$t.elementAtPosition(i+1, w)$.

$$
\begin{array}{ll}
\sqsubseteq & \text{"superfluous choice 8.3"} \\
 & \quad 1 \le i+1 \le \#t \qquad\qquad \rightarrow w := t(i+1) \\
 & \quad [\!] \ (0 = i+1) \vee (i+1 > \#t) \ \rightarrow \textbf{raise}(IPositionInvalidException) \\
\sqsubseteq & t.elementAtPosition(i+1, w)
\end{array}
$$

Returning to the second half of the sequential composition, we can expand the definition of I and it is then easy to see that it can be implemented by applying f to w, and then appending w to the end of s.

$$
(4) = \quad w, s : \left[\begin{array}{ll} \#s = i & \\ i \le \#t & \#s = i+1 \\ \forall j : 1..i \bullet sj = f(tj), & i+1 \le \#t \\ i \ne \#t & \forall j : 1..i+1 \bullet sj = f(tj) \\ w = t(i+1) & \end{array} \right]
$$

$$
\begin{array}{ll}
\sqsubseteq & w := f(w); \\
 & s := s \frown \langle w \rangle \qquad\qquad \lhd \\
\sqsubseteq & s.addAsLast(w)
\end{array}
$$

This completes the first development of the problem. Collecting the code gives the following program, where we have also defined a dummy handler for the exception, even though we know that the exception can never be raised within the block -- this is to prevent an over-zealous compiler complaining that there is an undefined procedure:

```
[| handler IPositionInvalidException ≙ skip •
    var i : N •
    s.removeAll1;
    i := 0;
    do i ≠ t.numberOfElements! →
        var w : Word •
            t.elementAtPosition(i + 1, w);
            w := f(w);                                   (*)
            s.addAsLast(w);
            i := i + 1
    od
|]
```

The assignment marked (*) is still not code, since we don't yet know the definition of the function f.

A particular function

The development above is clearly generic, in that we do not specify what function should be applied to each element of the list. We now look at a particular

function f, which is motivated by an exercise from a standard programming text.[8] We do not go into any great detail in the development, since the problem does not make any great use of exceptions, which are our chief concern in this chapter. However, several of the refinement steps are motivated by the need to use operations from the sequence collection class for manipulating the sequences involved, and this is the purpose of including the development here: we can see how the development has to be geared towards these class operations.

The problem requirements are as follows: we are given a text t in the form of a sequence of words, and two lists of words a and b. The task is to transform the text t into a text s by replacing each occurrence of a word a_i with the corresponding word b_i. Thus, for instance, if a is $\langle one, two, three \rangle$ and b is $\langle eleven, twelve, thirteen \rangle$, then a text $\langle it, is, two, minutes, past, three \rangle$ would be transformed to $\langle it, is, twelve, minutes, past, thirteen \rangle$. Clearly we can use the 'map' code developed above, and the only work left is to define the function f which will serve in this case, and then to develop code to replace the assignment marked $(*)$ in the development of the previous section. From the informal statement of requirements, we can see that the function f that we are interested in is the one which maps a word onto its transform:

$$\begin{array}{l}\textbf{procedure } transform(in : Word, \textbf{result } out : Word) \mathrel{\widehat=} \\ \quad out := f(in) \\ \text{where} \\ \quad f : Word \to Word \\ \quad \forall w : Word \bullet f(w) = \begin{cases} b(a^{-1}\, w) & \text{if } w \in \operatorname{ran} a \\ w & \text{if } w \notin \operatorname{ran} a \end{cases}\end{array}$$

Expressing this as a specification, we get the starting point for our development:

$$\begin{array}{ll} & out := f(in) \\ \sqsubseteq & out : \left[\begin{pmatrix} in \in \operatorname{ran} a \\ out = b(a^{-1}\, in) \end{pmatrix} \vee \begin{pmatrix} in \notin \operatorname{ran} a \\ out = in \end{pmatrix} \right] \end{array}$$

The algorithm chosen is a sequential search, where a local variable p is introduced, the purpose of which is to store an index such that, if in does actually appear in a, then p will point to it, and if in doesn't appear in a then p will be set to one more than the length of a. Once p has been set to this value, it is simple to achieve the desired postcondition with an alternation, using the *elementAtPosition* operation.[9]

$$\begin{array}{ll} \sqsubseteq & \textbf{var } p : \mathbb{N} \bullet \\ & p : \left[\begin{pmatrix} in \in \operatorname{ran} a \\ in = a\, p \end{pmatrix} \vee \begin{pmatrix} in \notin \operatorname{ran} a \\ p = \#a + 1 \end{pmatrix} \right]; \\ & out : \left[\begin{pmatrix} in \in \operatorname{ran} a \\ in = a\, p \end{pmatrix} \vee \begin{pmatrix} in \notin \operatorname{ran} a \\ p = \#a + 1 \end{pmatrix} \right], \end{array} \tag{5}$$

[8] The problem appears in [55] as exercise 1.7, where it is expressed in terms of arrays rather than sequences.

[9] As in the previous example, *superfluous choice* 8.3 is also required to introduce the 'superfluous' branch of *elementAtPosition*.

$$\left(\begin{array}{c} in \in \operatorname{ran} a \\ out = b(a^{-1}in) \end{array}\right) \vee \left(\begin{array}{c} in \notin \operatorname{ran} a \\ out = in \end{array}\right)\Bigg] \qquad \triangleleft$$

$$\sqsubseteq \quad \textbf{if } 1 \leq p \leq \#a \rightarrow b.elementAtPosition(p, out)$$
$$[\!] \; p = \#a + 1 \rightarrow out := in$$
$$\textbf{fi}$$

Setting p to the correct index value is achieved with an iteration, which uses yet another local variable v to store the most recently accessed element of a.

$$(5) \sqsubseteq \quad \textbf{var } v : Word$$
$$\text{“}I \mathrel{\widehat{=}} (1 \leq p \leq \#a + 1) \wedge (in \notin a[1..p-1])$$
$$\wedge (1 \leq p \leq \#a \Rightarrow v = a[p])\text{”} \bullet$$
$$p := 1;$$
$$a.elementAtPosition(p, v);$$
$$\textbf{do } p \neq \#a + 1 \wedge in \neq v \rightarrow$$
$$\textbf{if } 1 \leq p \leq \#a - 1 \rightarrow a.elementAtPosition(p+1, v)$$
$$[\!] \; p = \#a \rightarrow \textbf{skip}$$
$$\textbf{fi} \; ;$$
$$p := p + 1$$
$$\textbf{od}$$

This completes the development of the procedure *transform*. The collected code shows three calls to the *elementAtPosition* operation from the sequence class:

$$\textbf{var } p : \mathbb{N}, v : Word$$
$$p := 1;$$
$$a.elementAtPosition(p, v);$$
$$\textbf{do } p \neq \#a + 1 \wedge in \neq v \rightarrow$$
$$\textbf{if } 1 \leq p \leq \#a - 1 \rightarrow a.elementAtPosition(p+1, v)$$
$$[\!] \; p = \#a \rightarrow \textbf{skip}$$
$$\textbf{fi} \; ;$$
$$p := p + 1$$
$$\textbf{od};$$
$$\textbf{if } 1 \leq p \leq \#a \rightarrow b.elementAtPosition(p, out)$$
$$[\!] \; p = \#a + 1 \rightarrow out := in$$
$$\textbf{fi}$$

8.6 Conclusion

In this chapter, we have completed our study of exceptions. Chapter 3 started with a very abstract approach, which simply differentiated between normal and exceptional termination. Then, in Chapter 4, we extended this by introducing handlers and multiple exceptions. We have now shown how to relate these ideas to the very specific exception mechanisms which exist in the Collection Class Library. We gave a specification of one of the classes, and showed how it is possible to develop programs which use that specification. In particular, we showed several developments which use the exception-handling mechanisms of the Collection Class Library.

Chapter 9

Applications 2: iterators

9.1 Introduction

We investigate in this chapter how we can apply the ideas about iterators which were introduced in earlier chapters to the development of programs which use the iterator facility of the Collection Class Library. We are therefore not particularly concerned with exceptions in this chapter.

Our applications in this chapter are based on the sequence Collection Class introduced in the last chapter: this has the advantage that we do not need to go through a long specification before we actually get to the more interesting sections on how the iterator concepts are used with the class.

We start by giving an introduction to the iterator mechanisms available in the Collection Class Library. By taking a slightly unusual definition of the sequence type, we then show how one of these mechanisms can be related to the **it..ti** construct. We give some examples to show how programs can be developed using iterators over sequences, and we finish by returning to the sequence example — mapping a function — from the previous chapter.

9.2 Iterators in the Collection Class Library

The Collection Class Library incorporates two iteration methods: a method which uses cursors and a method which uses iterator functions. We will describe the cursor method briefly, and then concentrate on the use of iterator functions,

because this is the method that relates best to onr it. ti construct.

Iterating using cursors

An iteration over a collection can be achieved using the standard cursor operations, and the C++ for construct. Consider the following example.

```
ISet <int> coll;
ISet <int>::Cursor current(coll);
for ( current.setToFirst(); current.isValid();
      current.setToNext())
{
 // ...
 coll.elementAt(current);
 // ...
}
```

coll is first declared to be a set of integers, and current is declared (using the nested class Cureor) as a cursor for the set coll. The for constrnct is initialised with current.setToFirst, continues as long as current.isValid is true. and uses current.setToNext to advance to the next element. In the body of the for construct, coll.elementAt(current) is used to obtain (a reference to) the element pointed to by current.

In order to make programming slightly easier, the Collection Class Library provides a macro forCursor:

```
#define forCursor (c)        \
    for ((c).setToFirst();   \
         (c).isValid();      \
         (c).setToNext())
```

With coll and current defined as above, the program now becomes

```
forCursor(current)
{
 // ...
 coll.elementAt(current)
 // ...
}
```

There are warnings in the manual [22] about not adding or removing elements from a collection during an iteration, 'or all elements may not be visited once'. One reason for this is that any addition or removal from a collection causes the invalidation of all cursors.

This is all that we will say about iteration using cursors, other than to note the similarity with the iteration schemes of various object-oriented languages (see Section 10.2).

Iteration using iterator functions

The second, and more interesting, method of iteration in the Collection Class Library is to use the `allElementsDo` operation. The Collection Class Library reference manual [22] gives two reasons why the cursor method might not be acceptable:

- for unordered collections, it might be stylistically undesirable to have an explicit (yet arbitrary) order; and
- it is possible that it might be more efficient to carry out an iteration in an arbitrary order, using something other than cursors. For instance, if a tree implementation is being used, a recursive descent iteration might be more efficient, despite the extra function calls.

In order to use the `allElementsDo` operation, the user has to supply a function which is to be applied to each element of the collection in turn. For ordered collections, the iteration order is the same as the order of the collection, and, for unordered collections, the iteration order is arbitrary. The function to be applied to each element of the collection also gives a Boolean return value. This value can be used to terminate the iteration prematurely, since the iteration will only move on to the next element if this return value is true.

For example, the sequence class contains the following declaration:

```
Boolean allElementsDo (Boolean (*function) (Element&, void*),
                       void* additionalArgument = 0) ;
```

This function could be used to sum the elements of a sequence of integers as follows:

```
typedef ISeq <int> IntSeq ;

Boolean sumUpFunction (int const& i, void* sum){
   *(int*)sum += i;
   return True;
}

IntSeq s;
\\ ...
int sum = 0;
s.allElementsDo (sumUpFunction, &sum)
```

The `sumUpFunction` is declared: it takes two parameters, an integer `i` and the cumulative total `sum`. Its effect is to add `i` to the total. A sequence, `s`, of integers is declared. The variable `sum` is initialised to 0, and then `allElementsDo` is applied to `s` with `sumUpFunction` as the function to be applied to each element, and the result stored in `sum`.

In what follows, we will make no use of the Boolean value returned by the function supplied as a parameter to `allElementsDo`, but it could clearly be used to provide an exit mechanism from the middle of an iteration.

One of the drawbacks of this method of iteration can be seen in the above example: if the function to be applied to each element requires additional arguments, perhaps an accumulation parameter, these must be supplied as a second parameter to `allElementsDo`, and they are therefore not well encapsulated. In the example above, we have to supply `sum` as the extra parameter. The Collection Class Library provides yet another way of performing iteration to get around this: there is a form of the `allElementsDo` operation which takes as its parameter an 'iterator class', rather than the function to be applied to each element. These iterator classes must contain a function called `applyTo`, and the class must be derived from an abstract base class `IIterator`. Now additional arguments that are needed for the iteration can be passed as arguments to the constructor of the derived iterator class. However, we will not be using this form of `allElementsDo`.

9.3 Collection Class iterators and it..ti

We must now explain how we can relate the iterator mechanism described above — `allElementsDo` with an iterator function — to the **it..ti** construct from Chapter 5. We will describe this relationship, just as we described the mapping from guarded command programs with exceptions and handlers to C++ programs, in terms of an informal collection of guidelines, rather than any formally-defined translation.

The key to the relationship between `allElementsDo` and **it..ti** lies in our view of the type underlying the sequence class. The traditional view — and the one taken in Section 5.2 — is that sequences are constructed as either empty or by applying the Cons function to an element and another sequence:

$$\textbf{type}\ \mathrm{seq}\, A \mathrel{\widehat{=}} \mathsf{Empty} \mid \mathsf{Cons}\, A\, (\mathrm{seq}\, A)\ .$$

When seen like this, sequences are often called 'Cons-lists'. However, for reasons that will become clear, it is much more convenient for us to use an alternative, but isomorphic, view, where we treat a sequence as a 'Snoc-list'[11, Chapter 1]:

$$\textbf{type}\ \mathrm{seq}\, A \mathrel{\widehat{=}} \mathsf{Empty} \mid \mathsf{Snoc}\, (\mathrm{seq}\, A)\, A\ .$$

With this complementary definition of the sequence type, **it..ti** constructs now take a correspondingly different form, with the branches as always correspond-

ing to the branches of the type definition: we can find the sum of a sequence s with

it s **into** r **with**
  Empty $\rightarrow r := 0$
  [] Snoc ns n $\rightarrow r := ns + n$
ti ,

or the length of a sequence with

it s **into** r **with**
  Empty $\rightarrow r := 0$
  [] Snoc ns n $\rightarrow r := ns + 1$
ti .

Now we must show how to encode this **it..ti** in terms of `allElementsDo`. The technique used is to treat `r`, the result variable of the iteration, as an 'accumulation parameter': the initialisation of this parameter is derived from the Empty branch, while the function to be passed to `allElementsDo` is derived from the Snoc branch by replacing occurrences of the *front* of the sequence — ns in the examples above — with the result variable, and making the *last* element of the sequence — n above — a value parameter to the function.

Examples may make this transformation a little clearer. The summation iterator above becomes

```
r = 0 ;
s.allElementsDo (value n, value-result r . r = r + n) ;
```

where we have used the standard refinement calculus descriptions of parameter passing mechanisms, rather than C++'s more cryptic * and &. The initialisation `r = 0` is taken directly from the Empty branch, while the function `r = r + n` comes from the Snoc branch, $r := ns + n$, with ns replaced in our usual way by the result variable r.

Similarly, the length iterator above becomes

```
r = 0 ;
s.allElementsDo (value n, value-result r . r = r + 1) ;
```

Generalising slightly, we obtain the following guidelines for implementing an **it..ti** construct over a Snoc-list, s:

it s **into** r **with**
  Empty $\rightarrow r := c$
  [] Snoc as a $\rightarrow r := f(as, a)$
ti

is transliterated into C++ as

```
r = c ;
s.allElementsDo (value a, value-result r . r = f(r,a)) ;
```

The reason for treating sequences as Snoc-lists rather than Cons-lists is so that the accumulation will 'start at the right end'. In functional programming terms, our definition of **it..ti** (once the recursion is removed) corresponds to a *foldr*. rather than a *foldl*: we therefore needed a definition of the sequence type which matches this.

9.4 A simple development

We can now return to the example of Section 8.5, and fulfil the promise made there to repeat the development, this time using iterators. The specification of the problem is that a sequence t should be transformed to another sequence s. by applying an as-yet-unspecified function f to each element[1].

$$\textbf{var } s, t : ISeq[Word]$$
$$s : [(\#s = \#t) \wedge \forall j : 1..\#t \bullet s\,j = f(t\,j)] \qquad (1)$$

It turns out that this development using iterators is much simpler than the one given earlier, since all the details of local variables and **do..od** constructs are neatly encapsulated in the iterator construct. The first step is to re-express the specification as a catamorphism:

$$(1) = \quad s := (\!|m1, m2|\!)\ t \ ,$$

where

$$\begin{aligned} m1\,() &= \mathsf{Empty} \\ m2\ as\ a &= \mathsf{Snoc}\ as\ (f\ a) \ . \end{aligned}$$

So we can immediately implement the catamorphism with an **it..ti**:

$$\begin{array}{ll} \sqsubseteq & \text{"}\textit{assignment iterator}\ 5.2,\ \mathsf{Snoc}\ \text{version"} \\ & \textbf{it}\ t\ \textbf{into}\ s\ \textbf{with} \\ & \qquad \mathsf{Empty} \quad \rightarrow s := \mathsf{Empty} \\ & \quad [\!] \ \mathsf{Snoc}\ as\ a \ \rightarrow s := \mathsf{Snoc}\ as\ (f\ a) \\ & \textbf{ti} \end{array}$$

We can translate this into C++, using the guidelines above and the definition of **s.removeAll1**:

```
s.removeAll1 ;
t.allElementsDo ( value a, value-result s . s = Snoc s (f a) ) ;
```

[1] Both sequences happen to contain words, but that is not relevant here.

Once it is known which function f is required for a particular development, it is clear that the C++ function passed to `t.allElementsDo` can be further developed. Indeed, it is likely that it will be necessary to use another library operation, such as `s.add`, to add the new element to the end of the sequence s as it is being constructed. However, these are not our concerns here.

9.5 More complex types

We conclude this chapter with a brief description of how the above scheme for translation to the Collection Class Library might be extended to more complex types such as trees. There is an obvious reason why the scheme cannot deal with trees in its present form: the trick of accumulating the result of a *foldr* in a variable cannot work with a type like trees because there is more than one recursive occurrence on the right hand side — we need to accumulate the result of applying the function in question to the left sub-tree and to the right sub-tree, so we need two local variables. Of course, at the next level of unwinding, we need four variables, and so on.

However, we can get round this problem by splitting the problem into two parts: first we *flatten* the tree, then we iterate over the flattened structure. Of course, the flattened structure should be of a type for which we can easily convert to a call of `allElementsDo`, such as Snoc-lists. We can often appeal to the fusion law [11, Equation 2.12] as a way of transforming a catamorphism over trees to one over the new type, such as Snoc-lists.

Of course, we should also consider the question of efficiency: this approach to the implementation of iterators over complex structures — by flattening the structures and iterating over the flattened version — is only going to be acceptable if we can maintain the efficiency of an algorithm on the flattened structures when it is translated back to the more complex structure. This is a topic which remains open for further research, which will inevitably involve an investigation into the relationship between data refinement and the **it..ti** construct.

9.6 Conclusion

In this chapter, we have shown how the iterator construct introduced earlier can be related to the iterator mechanism which is built into the Collection Class Library. For sequences, we showed how treating a sequence as a Snoc-list meant that there was an easy way to turn an **it..ti** construct into an initialisation followed by a call of the `allElementsDo` operation. We have seen that the **it..ti** construct is actually very flexible, since we can choose whatever 'view' of the type is most convenient. This choice can be made at a late stage, as it does not need to be fixed.

Part V

Epilogue

Chapter 10

Related work and conclusions

In this final chapter, we set our work in context by surveying other published work on exceptions and iterators. We suggest some areas for future work, and end by drawing some conclusions.

10.1 Related work on exceptions

In this section, we look at the varieties of exception mechanisms available in a selection of programming languages, before examining some proposed techniques for the formalisation of these mechanisms.

A variety of exception mechanisms

When we looked, in Chapter 3, at why exceptions were needed in program development, we mentioned two models of exception handling: the termination model and the resumption model. These models reflect the different views of the actions possible when an exception is raised or signalled — whether the signaller should be ended and control passed to a handling routine, or whether control should be passed to the handler and then back to the original signaller at the point where the exception was raised, after some sort of attempt to 'clean up'. In fact, when we examine the literature, we find a few other proposals for possible actions: the signalling procedure could be 're-tried' from the beginning, or the exception could be propagated to allow a higher level of procedure to respond to the error. This wide choice of actions is reflected in the variety of mechanisms

available for exceptions in programming languages. Some languages permit more than one sort of response, while others restrict what is allowed. In general, the more complex mechanisms are, not surprisingly, the cause for more complex formalisations.

Exception handling in PL/I [35] is based on the resumption model. For some built-in exceptions, it is possible to specify that an operation should be re-tried, though this feature is not extended to user-defined exceptions. Exceptions are automatically propagated until an appropriate handler is found, and the scope of exceptions is global.

In contrast, exception handling in CLU [31] is based on the termination model, and exceptions must be explicitly propagated along the invocation chain. The language allows only statements and procedures to raise exceptions, not expressions. These exceptions can be parametrised, and, since exceptions that might be raised by a procedure must appear in its declaration, a certain amount of type-checking is possible.

Goodenough [20] has proposed a notation for exceptions that is extremely flexible, allowing several forms of response to the raising of an exception including both resumption and termination. The exceptions raised are neither typed nor parametrised.

Ada's exception mechanism (see for example [8]) is based on the termination model, and here too exceptions are non-typed and non-parametrised. Exceptions are not declared in procedure headings, and so compilers are unable to do much checking. Ada's mechanism is fairly complex and therefore difficult to formalise.

Yemini and Berry [57] have proposed an interesting scheme which is more ambitious than most of the others, and claims to cover all of the possible responses to the raising of an exception. They base their ideas on the so-called 'replacement model': by viewing a program as an expression, with side-effects allowed in expression evaluation, they see the raising of an exception in an expression as corresponding to a sub-expression which could not be fully evaluated. The handler of an exception produces a result which can be used in one of two ways: it can either replace the result of the sub-expression (thus giving the effect of a resumption), or it can replace the result of the whole expression which raised the exception (a termination) — hence the name 'replacement model'. The authors' concerns with modularity and orthogonality lead to a very powerful and flexible mechanism.

Formalising exception mechanisms

There have been several attempts at formalising some of these exception mechanisms. In many cases, this has meant imposing restrictions on the mechanism and providing formal semantics for only a part of it, or making significant proposals for change. For instance, Luckham and Polak [33] have attempted to give

axiomatic semantics to the exception mechanisms in Ada, and have only succeeded by making major changes: banning automatic propagation of exceptions and insisting that they are propagated explicitly to invokers, for example.

Yemini [56] has given an axiomatisation of the exception mechanism based on the replacement model, which involves only two new proof rules in addition to those of the block-structured language in which the mechanism is used. The simplicity of the axiomatisation is probably due to the use of procedures and the concern with orthogonality of program constructs.

Turning now to predicate transformer semantics, Cristian showed [15, 16] how a semantics could be given to a deterministic programming language with exceptions. His technique involved viewing programs as multi-exit structures, and thereby giving their meaning with sets of predicate transformers. Writing wp_c for Cristian's wp, we have in his notation,

- $wp_c(P, ;, \alpha)$ denotes the weakest precondition under which program P is guaranteed to terminate normally, satisfying the predicate α. This is simply the usual Dijkstra predicate transformer $wp(P, \alpha)$; and
- $wp_c(P, e, \alpha)$ similarly denotes the weakest precondition under which P is guaranteed to terminate at exit point e, satisfying α.

According to Cristian, the meaning of a program P was given by the predicate transformers $wp_c(P, ;, \alpha)$ and $wp_c(P, e, \alpha)$ for all possible exit points e.

However, if we try to use Cristian's technique on our own language which includes non-determinacy, exception blocks and an **exit** command, we soon run into problems. Following Cristian's ideas, we find that the meaning of a program P is given by the two predicate transformers

$$wp_c(P, ;, \alpha) \text{ and } wp_c(P, \mathbf{exit}, \alpha).$$

However, the separation of the meaning of P into two separate predicate transformers is the root cause of the problem. Consider the program

$$Q \mathrel{\hat{=}} \mathbf{skip} \parallel \mathbf{exit}$$

in which $\parallel$ denotes nondeterministic choice.[1] Since we cannot guarantee that Q will terminate successfully, $wp_c(Q, ;, \alpha) = \mathit{false}$. Similarly, since we cannot guarantee that the **exit** will be taken, $wp_c(Q, \mathbf{exit}, \alpha) = \mathit{false}$. So Cristian's semantics for Q is given by these two (constant) predicate transformers.

Now consider instead the program $[\![Q]\!]$. We know that, for successful termination,

$$wp_c([\![Q]\!], ;, \alpha) = \alpha$$

since the **exit** in Q will be caught by the exception block. This exposes the problem with Cristian's approach: we cannot give the meaning of $[\![Q]\!]$ in terms

[1] The language for which Cristian gave a semantics in [16] did not include this operator.

(only) of the meaning of Q, since α does not appear in the latter. The problem is that, with the two separate predicate transformers, we cannot express that Q is guaranteed to terminate, either normally or exceptionally: we can say only that it cannot be guaranteed to terminate in either state in particular.

Apart from Cristian's work, there have also been one or two other attempts to generalise predicate transformers to deal with exceptions. At the start of this work, we were aware of Cristian's approach; but since then we have encountered three others, two dating from the 1980s and the other contemporaneous with our own work.

In an unpublished report [6], Back and Karttunen discussed how Dijkstra's weakest precondition predicate transformer [17] could be generalised: instead of giving the semantics of a language by a function

$$w : Stat \to (Cond \to Cond)$$

(where $Stat$ is the set of all statements of the language, and $Cond$ denotes the set of all possible pre- and postconditions), they introduced the idea of a multiple-argument predicate transformer:

$$w : Stat \to (Cond^m \to Cond) \quad .$$

They used this notion to give semantics for statements with multiple exit points:

> $w(S)(Q_1, \ldots, Q_m)$ is the weakest precondition which guarantees that execution of S will terminate at one of the exit points of S, such that, if exit h_i is reached, then condition Q_i will hold, for $i = 1, \ldots, m$.

After showing how Dijkstra's so-called 'healthiness conditions' may be generalised to multiple-argument predicate transformers, they defined a simple and elegant language for multi-exit statements (see also [2]), and used this to give semantics to a language with goto statements, by transforming it to the language with multi-exit statements.

Their work is slightly more general than our work of Chapter 3, in that it allows for statements with any number of exit points rather than just the two, normal and exceptional, that we deal with. We prefer however to deal with multiple exits using a procedure mechanism, thus keeping the extra semantic structure to a minimum. They also do not treat recursion. More significant is the fact that they deal only with a programming language, rather than a refinement calculus, and so there is no notion of refinement of programs containing multi-exit statements.

In another report, published slightly later, [37], Manasse and Nelson give a similar definition of a weakest precondition of two arguments, though their work is not primarily concerned with exception handling, but the transformation of high-level control structures into low-level instruction sequences.

Another related piece of work [29] has been carried out at approximately the same time as our own. Following the work of Lukkien [34], which gave an

operational (trace-based) semantics for the guarded command language, Leino and van de Snepscheut give a similar semantics for a language with exceptions, by adding a state variable to indicate whether a statement has terminated normally or exceptionally, a construction isomorphic to our use of a pair of postconditions. From this trace semantics, they derive a weakest precondition semantics, with a definition, very similar to ours, of $wp(S)$ as a function of two arguments, for normal and exceptioual termination.

They explore the algebraic properties of *wp* in terms of arbitrary functions of two arguments, but are not concerned, as we are, with the refinement of programs containing **exits**. Indeed, the use of explicit specifications is a significant advantage when considering rules for rigorous program development that must refer to "assertions established *within* a program fragment". With specifications available, those assertions are explicit parts of the program; without them, rules for reasoning about a complete block must refer to the *reasoning* employed with respect to its constituents, rather than simply to the constituents themselves.

We could use the procedure-based exception mechanism of Chapter 4 to describe the construct $aaa \lhd bbb$ introduced in [29], which either executes *aaa* successfully, or leaves *aaa* via an **exit** and continues with *bbb*. This structure is easily modelled as

$$
\begin{array}{l}
[\![\ \textbf{handler}\ H \mathrel{\widehat{=}} bbb \bullet \\
\quad aaa' \\
]\!]
\end{array}
$$

where, rather than "**exit**", the body aaa' uses "**raise** H", but is otherwise identical to *aaa*.

Of course, the benefits of a "$\lhd$-calculus", so nicely explored in [29], are not so accessible when the relatively heavy mechauism of procedures is used. But the procedure-based mechanism is perhaps easier to adapt to the sometimes perverse demands of existing practice, and thus might be necessary anyway.

10.2 Related work on iterators

Several programmiug languages include some sort of iterator mechanism, and we review a selection of them in this section. There has been much less work on formalising iterators — that is, providing a formal semantics and a mechanism by which an iterator construct can be proved correct. Where such work has been carried out for a particular language, it is mentioned below.

Alphard

The Alphard language [52], developed at CMU in the late 1970s, has two iterator constructs:

- a **for** construct which is used for iteration over a complete data structure; and
- a **first** construct which is used (primarily) for search loops.

We will concentrate on the **for** construct, remarking only that execution of the **first** construct involves traversing a data structure until an element is found which meets some condition, and then performing some action. (If no element satisfying the condition is found, an alternative action can be taken.) The **for** construct takes the following form:

for x : $gen(y)$ **while** $\beta(x)$ **do** $ST(x, y, z)$.

A local variable x is declared, which will take, in turn, the values specified by the generator $gen(y)$. For each value of x which satisfies the constraint β, the loop body ST is executed. Clearly, the heart of the **for** construct lies in the idea of the generator $gen(y)$, so we look a little more closely at this. A generator is a 'form' (the Alphard term for abstract data type) obeying certain conditions: it must provide two Boolean-valued functions *&init* and *&next*, which have the side-effect that their invocation will produce a sequence of values to be bound to the loop variable. For both functions, the Boolean value returned indicates whether there are elements remaining which have yet to be iterated over.

Thus the meaning of the **for** construct can be given:

```
begin local x : gen(y);
    π ← x.&init;
    while π cand β(x) do
        (ST(x, y, z); π ← x.&next)
end
```

where π is a compiler-generated Boolean variable, $\leftarrow$ denotes assignment, and *cand* denotes the 'conditional conjunction' operator.

A simple example of an Alphard generator is the *upto* generator, which produces the sequence of numbers between a lower bound lb and an upper bound ub – that is, $\langle lb, lb + 1, ..., ub \rangle$, or the empty sequence if $ub < lb$.

```
form upto(lb, ub : integer) extends k : integer =
    beginform
    specifications
        inherits (allbut ←);
        function
            &init(u : upto) returns (b : boolean),
            &next(u : upto) returns (b : boolean);
    implementation
        body &init = (u.k ← u.lb; b ← u.lb ≤ u.ub);
        body &next = (u.k ← u.k + 1; b ← u.k ≤ u.ub);
    endform
```

Tbis generator can be used in a **for** statement. For instance, summing the first n integers is achieved by:

$$sum \leftarrow 0; \textbf{for } j : upto(1, n) \textbf{ do } sum \leftarrow sum + j \ .$$

It is possible to add verification information to an Alphard form, using an invariant clause, an initially clause, and pre- and postconditions for each functiou. It is also possible to give a concrete version of the state, and to give a representation function relating the concrete to the abstract view. Proof obligations can then be given to ensure the correctness of the form: this means ensuring that invariants are maintained by abstract and concrete operations, and that initial states correspond. Since a generator is just a special sort of form, it is possible to apply these proof rules to a generator. Using the expansion of the **for** construct given above, it is also possible to obtain a proof rule for the construct. However it is rather unwieldy. Because of this, there are various simpler proof rules for the **for** construct, which can be used when the generator satisfies certain conditions. In practice, many generators do satisfy these conditions, so the full form of the proof rule is seldom needed. Thus the effort of the proof is transferred from the verification of the **for** construct, to verification that the generator obeys the necessary conditions.

When comparing the Alphard generator with our own **it..ti** construct, we can see two immediate differences from our own work. Firstly, we have hidden all the details of how to 'move on' to the next element of the collection, by using a recursive procedure. Thus tbe user of the iterator has no need to know anythiug about the internal details of the object over which he is iterating, beyond its definition. Secondly, we have considerable flexibility with the **it..ti** construct: as noted in Chapter 9, we have the freedom to choose whatever 'view' is most convenient of the type of the variable to he iterated over. However, in Alphard, once the *&next* function is defined, the iteration order is fixed. On the other hand, it should be noted that the Alphard mechanism has the advautage that it is easy to describe a generator that produces only part of some structure to be iterated over — perhaps every other element of a list. While possible, this would be more convoluted with **it..ti**.

CLU

At around the same time that Alphard was being developed at CMU, Barbara Liskov's team at MIT was developing a language called CLU [30]. One of the guiding principles behind CLU was that it should support abstraction in program construction [32]. The language contains mechanisms to support three forms of abstraction:

- procedural abstraction — supported by procedures;
- data abstraction — supported by the use of clusters, the CLU term for abstract data type; and

- control abstraction — as well as the usnal **if** and **while** constructs, iterators can be defined.

An iterator is therefore a procedure-like construct, at the same level as procedures and clnsters. Like Alphard, iterators are used in conjunction with **for** statements: as the iterator produces elements of a data structure one at a time, so the **for** statement consumes them.

The syutax of the definition of an iterator in CLU is:

id = **iter** [*parms*] *args* [*yields*] [*signals*] [*where*]
routine_body
end *id* .

The *yields* clause specifies the number, order and types of the objects which will be delivered at each stage of the iteration. Within the *routine_body*, a *yield* statement is used to present the caller (a **for** statement) with the next element. (The *signals* clanse specifies which exceptions may be raised, and the *where* clause specifies **own** variables.)

The CLU **for** statement takes the following form:

for [$id_1, ...$] **in** *invocation* **do** *body* **end** ,

where *invocation* is the invocation of an iterator. Unlike Alphard, where the looping mechanism is found in the **for** statement, in CLU the looping must be explicitly programmed in the body of the iterator.[2] Each time a **yield** statement is executed in the iterator's body, the objects yielded are assigned to the variables declared in the **for** statement, and the body of the **for** statement is executed. Then the iterator body is resumed at the point iumediately following the **yield** statement. The **for** statement terminates on termination of the iterator.

For instance, we can define an iterator to yield the characters of a string, one at a time:

```
string_chars = iter (s : string) yields (char);
    index : int := 1;
    limit : int := string$size(s);
    while index ≤ limit do
        yield(string$fetch(s, index));
        index := index + 1;
        end;
    end string_chars;
```

[2] In fact, it is usual to use another, more primitive, for statement in the body of an iterator.

We can then use this iterator to discover how many numeric characters a string contains:

```
count_numeric = proc(s : string) returns(int);
    count : int := 0;
    for c : char in string_chars(s) do
        if char_is_numeric(c)
            then count := count + 1;
            end;
        end;
    return(count);
    end count_numeric
```

In her thesis [53], Wing gave a method for specifyiug iterators, which has recently been extended to iterators for concurrent and distributed systems [54]. Wing's technique for specifying CLU iterators involves adding assertions to an iterator, similar to those used for a procedure, to give pre- and postconditions for each invocation. However, unlike a procedure, an iterator's specification is concerned with more than just two states — as well as the overall first and last states, there are the intermediate states for each invocatiou. There also needs to be a distinction hetween two kinds of termination for iterators — the 'real' termination when all the elements have been yielded, and the suspension that occurs after each **yield**. The assertions refer to state variables which can be decorated with subscripts *pre* and *post*, as well as a special state object (ie an auxiliary variable) *first*, which flags when we are in the very first state, and history variables which 'remember' values between invocations.

Both of the remarks made above when comparing Alphard iterators to the **it..ti** construct still hold true for CLU iterators: the user has to program explicitly the method of progress through the collection, and the **it..ti** is more flexible. However there is more generality here, iu that it is possible for the user of the CLU iterator to write a **yield** statement which returns a more complex expression than simply the current object in the collection — for instance, each **yield** statement might return a pair of consecutive characters in the sequence, allowing a **for** statement to calculate the frequency of pairs of characters. This would be considerably more complex using an **it..ti**.

On the verification side, while Wing's assertions and associated proof rules do allow the verification of a CLU iterator, the proof is at a very low level, dealing with the intermediate states during the iteration as well as the overall pre- and postconditions. In contrast, the effort of verification for the **it..ti** construct is at a much higher level, involving the reformulation of the postcondition as a catamorphism. Once this has been done, implementation as an **it..ti** is immediate, by *assignment iterator* 5.2, or a similar law for other types.

Object-oriented languages

In recent years, many object-oriented environments have introduced libraries of abstract data types, often including container classes. These container classes, which describe such types as sets, bags, trees, etc., often contain some sort of iteration mechanism, whereby the user can traverse the data structure. However, these mechanisms are often based on the notion of a cursor, with the user having to supply cursor manipulation routines. As an example, we will describe the iteration mechanisms available in Eiffel [40].

The Eiffel library contains an Iteration Library which consists of classes which encapsulate various iteration mechanisms over arbitrary data structnres — linear iteration, two-way iteration, tree iteration (preorder, postorder or inorder). These iterations are defined in terms of two sorts of deferred routines - - these are routines which are called in the iterator, but not actually defined until the iterator is used: traversal rontines and operation routines. The traversal routines are concerned with cursors, and need only be defined once for each data type. The operation routines are concerned with the particular actions to be taken as part of each iteration, and so can be given different values to achieve different iterations.

For example, the *LINEAR_ITERATION* library class contains the following *do_until* routine:

```
do_until(s : TRAVERSABLE[T]) is
    - - Starting at the beginning of s, apply action to every item of s
    - - up to and including the first one satisfying test.
require
    traversable_exists : s ≠ Void;
    traversable_satisfied : invariant_value(s)
do
    from
     start(s); prepare(s)
    invariant
     invariant_value(s)
    until
     off(s) or else test(s)
    loop
     action(s);
     forth(s)
    end;
    if not off(s) then action(s) end;
    wrapup(s)
ensure
    not off(s) implies test(s)
end - -do_until
```

In this iteration, the traversal routines are *start*, *forth* and *off*, and the operation rontines are *prepare*, *action*, *test* and *wrapup*.

In order to iterate over an object s which is a *FIXED LIST*, we can use an integer index *position* to represent the cursor, and the traversal routines become:

- $start(s)$ – $position := 1$
- $forth(s)$ – $position := position + 1$
- $off(s)$ – $position > count$, where *count* gives the number of occupied places in s.

The Eiffel renaming mechanism is used to allow two different iterations over the same structure, by renaming the operation routines as they are imported.

There are difficulties with the cursor approach to iterators, not least the problems of nested iterations, when it is not easy to keep track of several cursors, and the problem of robust iterations, when elements may be added or removed during an iteration. There has been work reported to solve these problems: [23] proposes a CLU-like mechanism for Eiffel and [24] does the same for C++, while [26] is concerned with robust iterators in a C++ class library.

The chief disadvantage of the Alphard mechanism and the more recent proposals for object-oriented languages is that they require the user of the iterator to supply routines to control the iteration. This means that knowledge of the data structure's implementation is required, thus negating one of the primary advantages of using the iterator in the first place. The CLU mechanism is much cleaner in that respect, but the complications in the proof obligations caused by the suspend/resume semantics are non-trivial. Lamb has proposed [27] the use of trace specifications for the specification of Alphard-style iterators. He also mentions the use of procedure parameters, and shows how traces can be used to give 'partial specifications' of iterators which use procedure parameters.

However, the main difference between the **it..ti** construct and all of the related work mentioned in this section is in the level of abstraction and the level of mathematical maturity required for their use: the iteration schemes for Alphard, CLU and Eiffel all require the user of the iterator to supply routines which need knowledge of the internal structure of the type of the variable being iterated over. The user of the **it..ti** can work at a more abstract level, but needs to be more mathematically mature: while the average programmer can easily understand the ideas of initialising and advancing a cursor, he may have more difficulty with catamorphisms! However, he would probably benefit from thinking more deeply about the constructors of both the type of the variable to be iterated over and the target type of the iteration.

10.3 Further work

There are several directions in which it would be possible to extend the work described in this thesis, and we now investigate a few of them.

Exception handling and parameters

In Section 4.4, we mentioned the idea of passing parameters to exception handlers. This feature already exists in some programming languages such as CLU [31]. It would not be very difficult to extend our handler mechanism – based as it is on the idea of procedures — to cover this possibility, using the standard (by value) procedure-passing mechanism for procedures.

Exception handling in other programming languages

In Chapter 8, we showed how the exception-handling mechanism introduced in Chapter 4 could be mapped to the C++ language and the Collection Class Library. It would be interesting also to investigate how to develop programs in other languages with different exception-handling mechanisms. For instance, Burns and Wellings [13, Chapter 6] describe the exception-handling mechanisms in the Ada95 and C++ languages and a proposed mechanism for C [28]. These mechanisms are classified, together with those for CHILL, CLU and Mesa, in terms of the scope of a handler, whether exception propagation occurs, whether the resumption or termination model is used, and whether parameters can be passed to a handler. An attempt to formalise these mechanisms using our framework could make an interesting comparison.

Iterators in other languages

In a similar vein, it would be possible to examine the iterator mechanisms of various programming languages to see how easy it would be to map our **it..ti** construct onto different target languages.

Iterators and optimisation

The functional programming literature (e.g. [11, Chapters 7–10]) contains detailed studies of the application of catamorphisms to so-called 'optimisation problems'. Since the **it..ti** construct is based on catamorphisms, it should be possible to re-cast these problems in the refinement calculus framework. This alternative approach might yield further insights into this important application area.

The Collection Class Library

Turning to the Collection Class Library, there are at least two areas that require further work before the aim of using the library in a formal development process can be realised. The first problem is the use of cursors, since our specification

of the sequence class in Chapter 8 did uot include any mention at all of cursors. Although several attempts were made to describe the cursor mechanism formally, it was not possible to find a well-structured description, such that the specification of any other library module also involving cursors would be able to re-use the cursor description in the sequence class. The second problem was referred to in the Introduction — pointers. The difficulty of formally developing programs which use the pointer mechanism of a programming language remains an open problem, aud an important one. For efficiency reasons, programmers are always going to want to program with pointers, and it is the task of the formal methods community to provide them with a formal basis for doing so.

Automation

All of the developments in this thesis have been carried out by hand, without the assistance of any automation. However, it is clear that refinement calculus development methods are likely to be adopted on a large scale by industry only when there is significant tool support. Thus the addition of our new constructs to existing refinement calculus tools [14] would be a great aid in promoting their use.

Case studies

Finally, more experience is needed in the use of all the language coustructs introduced in this thesis: exceptions, exception handlers, iterators, procedure variables, and so on. It is by working on case studies that we will be able to propose (and later prove correct) laws which will simplify the developments that use these constructs. As Naumann put it [49], 'it remains to gain more experience with development of higher order programs in refinement calculi, in order to find more convenient notations and derived laws'.

10.4 Conclusions

The work described in this thesis has extended the refinement calculus to cover two new areas — exceptious and iterators. The motivation for the work came from IBM's Collection Class Library, and it is against this that we can measure its success.

For exceptions, we built up a formalised exception mechanism, in stages, from a fairly abstract start where we simply made a distinction between normal and exceptional termination, and gave a semantic framework in which refinement laws could be proved correct (Chapter 3). We then extended this by allowing the user to differentiate between different exceptional terminations in a single program, associating appropriate actions with each exception. To achieve this, Chapter 4 introduced a mechanism for handling exceptions, based on the

use of procedures. The connection back to the Collection Class Library was made in Chapter 8 where a specification of a sample class — sequences — was given, and the Collection Class Library exception mechanism was related to the procedure-based mechanism previously described. Several programs were developed formally, making use of the refinement laws which had been proposed and proved correct. We can therefore regard the work on exceptions as reasonably successful, in that it achieved its aim of formalising the Collection Class Library's exception mechanism. Moreover, the ideas introduced should be easy to transfer to different languages, allowing the formalisation of other exception mechanisms.

For iterators, we took our inspiration from the field of functional programming. In particular, we based the **it..ti** construct in Chapter 5 on catamorphisms, although it was formally defined as a recursive procedure. Having worked initially with sequences, we also showed how iterators could be defined over more general types, and gave examples showing the **it..ti** construct in use for developing programs. The second of these examples also showed how a result from functional programming about catamorphisms — the banana-split law — could be used to assist in an iterator-based development. Since our goal was to describe iterators for the Collection Class Library, we then needed a way to encapsulate the **it..ti** construct so that it could be defined in a library class, to be used as required. This meant that it would be necessary to pass procedure values to the library procedure, corresponding to the branches of the iterator. Although the usual parameter mechanisms for the refinement calculus do not cover procedure parameters, we were able to use the work of Nanmann as a basis for a description of procedures as values. Once again, weakest precondition semantics allowed us to propose and prove correct various laws. In Chapter 7, we used this work to develop a theory of procedures as parameters and to provide an encapsulation of the iterator construct so that it could be placed in a library. Finally, Chapter 9 brought us back to the Collection Class Library, as we showed how an iterator for the sequence class described earlier could be formalised with an **it..ti** construct. This involved taking a slightly unusual definition of the sequence type, so that the recursion of the **it..ti** could be mapped to a *foldr*. The success in formalising the Collection Class Library's iterator mechanism is perhaps not so striking as that for exceptions, as it is dependent on finding a definition for the type which corresponds to a *foldr*. However, for types which do have such a definition, the formalisation works well. Moreover, the **it..ti** construct remains well-defined for types which do not have such a definition, and the results and laws on procedures variables have much wider application.

Bibliography

[1] A. Albano and R. Morrison, editors. *Persistent Object Systems*, Workshops in Computing. Springer-Verlag, 1992. Proceedings of the 5th International Workshop on Persistent Object Systems, San Miniato, Italy.

[2] R.-J.R. Back. Exception handling with multi-exit statements. Technical Report IW 125/79, Mathematisch Centrum, Amsterdam, November 1979.

[3] R.-J.R. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[4] R.-J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[5] R.-J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In Möller [41].

[6] R.-J.R. Back and M. Karttunen. A predicate transformer semantics for statements with multiple exits. University of Helsinki, unpublished MS, 1983.

[7] R. Banach and M. Poppleton. Retrenchment: an engineering variation on refinement. In Didier Bert, editor, *B'98: Recent advances in the development and use of the B method*, number 1393 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[8] J. Barnes. *Programming in Ada95*. Addison-Wesley, 1996.

[9] R.S. Bird. Functional algorithm design. In Möller [41].

[10] R.S. Bird. The algebra of programming principles. In M. Broy, editor, *Deductive program design*, volume 152 of *NATO ASI Series F: Computer and Systems Sciences*. Springer-Verlag, 1996. Proceedings of 1994 Marktoberdorf Summer School.

[11] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1997.

[12] H.J. Boom. A weaker precondition for loops. *Trans. Prog. Lang. Sys.*, 4(4):668-677, October 1982.

[13] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Langyages.* Addison-Wesley, 2nd edition, 1997.

[14] D. Carrington, I.J. Hayes, R. Nickson, Watson G., and Welsh J. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, University of Queensland, Anstralia, June 1994.

[15] F. Cristian. Robust data types. *Acta Informatica*, 17:365-397, 1982.

[16] F. Cristian. Correct and robust programs. *IEEE Trans. Soft. Eng.*, SE-10(2):163–174, March 1984.

[17] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, August 1975.

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

[19] J.D. Eckart. Iteration and abstract data types. *SIGPLAN Notices*, 22(4), April 1987.

[20] J.B. Goodenough. Exception handling: Issues and a proposed notation. *Comm ACM*, 18(12):683-696, December 1975.

[21] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

[22] IBM Corporation. *IBM C/C++ FirstStep Tools· Collection Class Library Reference.* S82G-3757.

[23] M. Katrib and I. Martinez. Collections and iterators in Eiffel. *Journal of Object-Oriented Programming*, pages 45–51, November-December 1993.

[24] M.H. Kim. A new iterator mechanism for the C++ programming language. *ACM SIGPLAN Notices*, 30(1), January 1995.

[25] S. King and C.C. Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7(1):54-76, 1995.

[26] T. Kofler. Robust iterators in ET++. *Structured Programming*, 14:62–85, 1993.

[27] D.A. Lamb. Specification of iterators. *IEEE Trans. Soft. Eng.*, 16(12):1352-1360, December 1990.

[28] P.A. Lee. Exception handling in C programs. *Software — Practice and Experience*, 13(5):389–406, 1983.

[29] K.R.M. Leino and J.L.A. van de Snepscheut. Semantics of exceptions. In Ernst-Rüdiger Olderog, editor, *Programming concepts, methods and calculi*, volume 56 of *IFIP Transactions A: Computer Science and Technology*. Elsevier, 1994.

[30] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Number 114 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

[31] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Trans. Soft. Eng.*, SE-5(6):239–251, November 1979.

[32] B. Liskov, A. Suyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20(8):564–576, August 1977.

[33] D.C. Luckham and W. Polak. Ada exception handling: an axiomatic approach. *Trans. Prog. Lang. Sys.*, 2(2), April 1980.

[34] J.J. Lukkien. An operational semantics for the guarded command language. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[35] D.M. MacLaren. Exception handling in PL/I. *SIGPLAN Notices*, 12(3), 1977.

[36] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[37] M.S. Manasse and C.G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories. September 1984.

[38] E.G. Manes and M.A. Arbib. *Algebraic approaches to program semantics*. Texts and monographs in computer science. Springer-Verlag, 1986.

[39] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[40] B. Meyer. *Eiffel: the Language*. Prentice-Hall, 1992.

[41] B. Möller, editor. *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[42] C.C. Morgan. The specification statement. *Trans. Prog. Lang. Sys.*, 10(3), July 1988. Reprinted in [45].

[43] C.C. Morgan. *Programming from Specifications*. Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1990.

[44] C.C. Morgan. *Programming from Specifications*. Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 2nd edition, 1994.

[45] C.C. Morgan and T.N. Vickers, editors. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer-Verlag, 1993.

[46] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[47] D.A. Naumann. *Two-categories and program structure: data types, refinement calculi, and predicate transformers*. PhD thesis, University of Texas at Austin, USA, 1992.

[48] D.A. Naumann. Predicate transformer semantics of an Oberon-like language. In Ernst-Rüdiger Olderog, editor. *Programming concepts, methods and calculi*, volume 56 of *IFIP Transactions A: Computer Science and Technology*. Elsevier, 1994.

[49] D.A. Naumann. Predicate transformers and higher order programs. *Theoretical Computer Science*, 150(1):111–159, 1995.

[50] C.G. Nelson. A generalization of Dijkstra's calculus. *Trans. Prog. Lang. Sys*, 11(4):517–561, October 1989.

[51] D.L. Parnas. Response to detected errors in well-structured programs. Technical report, Dept. Computer Science, Carnegie-Mellon University, July 1972.

[52] M. Shaw, Wm.A. Wulf, and R.L. London. Abstraction and verification in Alphard: iteration and generators. In M. Shaw, editor. *Alphard: form and content*, pages 73–116. Springer-Verlag, 1981.

[53] J.M. Wing. *A two-tiered approach to specifying programs*. PhD thesis, MIT. Lab. for Comp.Sci., 1983.

[54] J.M. Wing and D.C. Steere. Specifying iterators for concurrent and distributed systems. Unpublished draft, 1994.

[55] N. Wirth. *Algorithms and Data Structures*. Prentice-Hall, 1986.

[56] S. Yemini. An axiomatic treatment of exception handling. In *9th Annual Symposium on Principles of Programming Languages*, 1982.

[57] S. Yemini and D.M. Berry. A modular verifiable exception-handling mechanism. *Trans. Prog. Lang. Sys.*, 7(2), April 1985.

Appendix A

Procedures and C++ functions

As was mentioned in Section 8.3, we encounter a problem when modelling the C++ collection classes in the refinement calculus, due to the fact that the C++ language does not distinguish between procedures and functions. Although the refinement calculus language considers only procedures, many of the collection classes contain operations which are enquiries on the state variables – in C++, these are functions which give return values and therefore can be used as expressions. For example, as well as the normal operations such as *add* and *delete* which alter state variables, the sequence class also contains such operations as *numberOfElements*, which returns the number of elements in a sequence. If we stay strictly within the refinement calculus notation, we must model the operation as a procedure with a result parameter:

$$\textbf{procedure}\ \mathit{numberOfElements}(\textbf{result}\ n : \mathbb{N}) \mathrel{\widehat{=}} n := \#s\ .$$

Now, how can we use a procedure such as this? As usual when we intend to use a procedure, we manipulate our program until it 'matches' the body of the procedure (with suitable substitutions for parameter passing). In the simplest possible case, suppose we needed to develop code to implement

$$x := \#s$$

where s is a sequence. This is easy: we use the result-assignment law to refine this to a call of the procedure *numberOfElements* with x as the result parameter:

$$\sqsubseteq\ s.\mathit{numberOfElements}(x)$$

When we come to transliterate to C++, the obvious way to code up the result parameter is as an assignment. So we finish up with

```
x = s.numberOfElements    .
```

However, things are not so simple if the expression on the right of the assignment is slightly more complex:

$$x := \#s + 1 \ .$$

Now we have to introduce a local variable, which we use to store the result of a call to *numberOfElements*:

$$\sqsubseteq \quad \textbf{var } l \bullet \\ \quad s.numberOfElements(l); \\ \quad x := l + 1$$

This is transliterated to C++ as

```
l = s.numberOfElements;
x = l+1
```

whereas the program which we would really like to develop is

```
x = s.numberOfElements + 1    ,
```

using the result of the function call as a sub-expression in the expression being assigned.

One possible solution to this problem would be to extend the refinement calculus notation with some form of 'calculus of functions'. Although this is an interesting idea, it is not very relevant to the main topic of our work, and we therefore reject it. Indeed, it is a non-trivial problem, and could potentially lead to non-deterministic expressions in the language. Instead, we adopt a pragmatic solution, which allows us to specify these operations as procedures, in the normal refinement calculus fashion, and then to develop programs which use them as though they were functions, in the natural C++ way.

The solution is based on an abbreviation, which we will explain by developing code for the specification mentioned above:

$$x := \#s + 1 \ .$$

We notice that the body of *numberOfElements* consists simply of an assignment to the result parameter n. Under these circumstances, if the expression being assigned in the procedure body (here '$\#s$') appears as a sub-expression on the right-hand side of an assignment during a development, or within the guard of an alternation, we allow ourselves to replace it by the procedure name decorated with an exclamation mark, *numberOfElements*! So we can write

$$x := \#s + 1 \\ \sqsubseteq \\ x := numberOfElements! + 1 \ .$$

This is transliterated to C++ in the obvious way.

This abbreviation cuts out some of the details of the development: essentially, we are saved from introducing a local variable to store the result of the 'enquiry procedure'. If we were being totally formal, this local variable would become a local variable in the C++ code, which we could then optimise away by noting that, since the enquiry operation has no effect on the state variables, we are justified in replacing any occurrence of the local variable in an immediately-following expression with an inline evaluation of the enquiry.

The full details of these abbreviations are given below. In all of them, P is taken to be a procedure without side-effects which has a single result parameter, with a specification of the following form, where E is an expression:

procedure P(**result** r) $\widehat{=}$
  r := E .

Abbreviation A.1 *assignment abbreviation definition*

$P!$ may be used as an expression in the right-hand side of an assignment, standing for a declaration of a fresh local variable l, a call of P with l as the result parameter, followed by the assignment with l in place of $P!$:

$x := exp(P!) \mathrel{\widehat{=}}$ |[**var** l •
  $P(l)$;
  $x := exp(l)$
]|

The associated law which allows us to introduce this abbreviation into an assignment is:

Law A.2 *assignment abbreviation*

If F is an expression, and P is defined by
procedure P(**result** r) $\widehat{=}$
  r := E
then
$x := F$
$\sqsubseteq$
$x := F[E \setminus P!]$.

Abbreviation A.3 *alternation guard abbreviation definition*

$P!$ may be used as an expression in the guard of an alternation, standing for a declaration of a fresh local variable l, a call of P with l as the result parameter, followed by the alternation with l in place of $P!$:

if ([] i • $G_i(P!) \rightarrow B_i$) **fi**
$\widehat{=}$ |[**var** l •
  $P(l)$;
  if ([] i • $G_i(l) \rightarrow B_i$) **fi**
]|

The law which allows us to introduce this abbreviation into the guard of an alternation is:

Law A.4 *alternation abbreviation*

$$\begin{array}{l}\textbf{if } ([]\ i \bullet G_i \rightarrow B_i)\ \textbf{fi} \\ \sqsubseteq \\ \textbf{if } ([]\ i \bullet G_i[E \setminus P!] \rightarrow B_i)\ \textbf{fi}\end{array}$$

The form of the abbreviation when $P!$ is used in the guard of an iteration is slightly more complex and may not actnally be needed.

Abbreviation A.5 *iteration guard abbreviation definition*

$P!$ may be used as an expression in the guard of an iteration, standing for a declaration of a fresh local variable l, a call of P with l as the result parameter, followed by the iteration with l in place of $P!$, and a call of P after the iteration body:

$$\begin{array}{l}\textbf{do } ([]\ i \bullet G_i(P!) \rightarrow B_i)\ \textbf{od} \\ \ \widehat{=}\ |[\ \textbf{var } l \bullet \\ \qquad P(l); \\ \qquad \textbf{do } ([]\ i \bullet G_i(l) \rightarrow B_i;\ P(l))\ \textbf{od} \\ \quad]|\end{array}$$

So our example above now becomes

$$\begin{array}{l}x := \#s + 1 \\ \sqsubseteq \\ x := s.numberOfElements! + 1\ .\end{array}$$

which is transliterated to C++ as

```
x = s.numberOfElements + 1      .
```

This abbreviation is used most frequently with the enquiry procedures in the specification of sequences in Chapter 8.

Appendix B

Summary of laws

For easy reference, all of the laws, definitions and abbreviations from elsewhere in the thesis are gathered together in this appendix.

Chapter 3

Law 3.1 *program after exit*

A program following **exit** has no effect.

exit; aaa = **exit**
(Equality of programs means semantic equivalence, that is, mutual refinement.)

Law 3.2 *exit ending block*

An **exit** at the end of an exception block has no effect.

$[\![aaa; \textbf{exit}]\!] = [\![aaa]\!]$

Law 3.3 *exception-free block*

Exception blocks have no effect on exception-free programs.

$[\![aaa]\!] = aaa$ *provided* aaa *is exception-free*

Law 3.4 *exceptional specification*

An exceptional specification can be formed by duplicating the postcondition of a non-exceptional specification statement, and surrounding it with an exception block.

$w : [\alpha, \beta] = [\![w : [\alpha, \beta > \beta]]\!]$

Law 3.5 *take normal branch*

A specification statement can be implemented by taking the normal branch unconditionally.

$w : [\alpha, \beta > \gamma] \sqsubseteq w : [\alpha, \beta]$

Law 3.6 *take exceptional branch*

A specification statement can be implemented by achieving the exceptional postcondition, and then performing an **exit**.

$w : [\alpha, \beta > \gamma] \sqsubseteq w : [\alpha, \gamma]$; **exit**

Law 3.7 *else notation*

Specifications and the 'else' notation

$w : [\alpha, \beta > \gamma] = w : [\alpha, \beta] > w : [\alpha, \gamma]$

Law 3.8 *take normal branch*

An 'else' construct can he implemented by taking the first branch unconditionally.

$aaa > bbb \sqsubseteq aaa$

Law 3.9 *take exceptional branch*

An 'else' construct can be implemented by taking the second branch unconditionally, followed by an **exit**.

$aaa > bbb \sqsubseteq bbb$; **exit**

Law 3.10 *choice-else*

A nondeterministic choice between two programs which do not contain exceptions is equivalent to an exception block containing the programs as branches of an 'else' construct.

$aaa \;[]\; bbb = [\![aaa > bbb]\!]$ *provided aaa and bbb are exception-free*

Law 3.11 *introduce trivial else*

An **exit**-exception pair can be introduced by offering the trivial choice between equal alternatives (corollary to *choice-else* 3.10).

$aaa = [\![aaa > aaa]\!]$ *provided aaa is exception-free*

Law 3.12 *sequential composition*

Distribute sequential composition through 'else'.

$aaa > bbb$
$\sqsubseteq$
$(ccc > bbb)\ ;\ (ddd > eee)$ *provided* $aaa \sqsubseteq ccc; ddd$ *and* $bbb \sqsubseteq ccc; eee$

Law 3.13 *sequential composition*

Splitting a specification with sequential composition.

$w : [\alpha, \beta > \gamma]$
$\sqsubseteq$
$w : [\alpha, \delta > \gamma]\ ;$
$w : [\delta, \beta > \gamma]$

Law 3.14 *recursion*

Let e be an integer-valued expression, V a logical constant, aaa a program, and $\mathcal{P}$ a monotonic program-to-program function, and assume that neither aaa nor $\mathcal{P}$ contains V. Then if

$$\{e = V\}aaa \;\sqsubseteq\; \mathcal{P}(\{0 \le e < V\}aaa)$$

we can conclude

$$aaa \sqsubseteq \textbf{mu}\ D \bullet \mathcal{P}(D)\ \textbf{um}\ .$$

Law 3.15 *iteration*

$w : [\alpha, \alpha \wedge \neg G > \beta]$
$\sqsubseteq$
do $G \rightarrow$
  $w : [\alpha \wedge G\ , \alpha \wedge (0 \leq e < e_0) > \beta]$
od

Law 3.16 *loop introduction*

$w : [\alpha, \beta]$
$\sqsubseteq$
loop
  $w : [\alpha\ , \alpha \wedge (0 \leq e < e_0) > \beta]$
end .

Chapter 4

Law 4.1 *handler definition*

A declaration

handler $H \mathrel{\widehat{=}} hhh$

is an abbreviation for

procedure $H \mathrel{\widehat{=}} hhh$; **exit** .

Law 4.2 *raise definition*

Raising an exception

raise H

is an abbreviation for

H ,

a call of procedure H.

Law 4.3 *sequential composition and raise*

aaa; bbb
= $[\![$ **handler** $H \mathrel{\widehat{=}} bbb \bullet$
  aaa; **raise** H
$]\!]$ *provided* aaa *and* bbb *are* **exit**-*free*

Law 4.4 *introduce handler*

$$
\begin{array}{ll}
 & [\![\ \mathcal{P}(aaa;\ \mathbf{exit})\ ]\!] \\
= & \quad [\![\ \mathbf{handler}\ H \mathrel{\widehat{=}} aaa \bullet \\
 & \qquad \mathcal{P}(\mathbf{raise}\ H) \\
 & \quad]\!]
\end{array}
$$

Law 4.5 *introduce handler*

$$
\begin{array}{ll}
 & [\![\ \mathcal{P}(aaa > (bbb;\ ccc))\ ]\!] \\
= & \quad [\![\ \mathbf{handler}\ H \mathrel{\widehat{=}} ccc \bullet \\
 & \qquad \mathcal{P}(aaa \mathbin{[\!]} (bbb;\ \mathbf{raise}\ H)) \\
 & \quad]\!]
\end{array}
$$

Law 4.6 *introduce handler to choice*

$$
\begin{array}{ll}
 & aaa \mathbin{[\!]} bbb \\
= & \quad [\![\ \mathbf{handler}\ H \mathrel{\widehat{=}} bbb \bullet \\
 & \qquad aaa \\
 & \qquad \mathbin{[\!]}\ \mathbf{raise}\ H \\
 & \quad]\!]
\end{array}
\qquad \textit{provided } aaa \textit{ and } bbb \textit{ are } \mathbf{exit}\textit{-free}
$$

Law 4.7 *raise-sequential composition*

$$
\begin{array}{ll}
 & \mathbf{raise}\ H \\
= & \quad \mathbf{raise}\ H;\ aaa
\end{array}
$$

Law 4.8 *disjunction-else distribution*

$$
\begin{array}{ll}
 & w : [\alpha, \beta_1 \vee \ldots \vee \beta_n] \\
\sqsubseteq & \quad [\![\\
 & \qquad w : [\alpha, \beta_1] \\
 & \qquad > \\
 & \qquad w : [\alpha, \beta_2] \\
 & \qquad \ldots \\
 & \qquad > \\
 & \qquad w : [\alpha, \beta_n] \\
 & \quad]\!]
\end{array}
$$

Law 4.9 *else distribution*

$$(aaa;\ bbb) > ccc$$
$$= \quad (aaa > ccc);\ bbb$$

Law 4.10 *else distribution*

$$(aaa;\ bbb) > ccc$$
$$\sqsubseteq \quad (aaa > ccc) : (bbb > ccc) \qquad \textit{provided } ccc \sqsubseteq aaa;\ ccc$$

Chapter 5

Definition 5.1 *sequence iterator*

An iteration over a sequence s of the following form

it s **into** r **with**
$\quad \langle\,\rangle \quad \longrightarrow bbb$
$\quad [\!] \ a{:}as \ \longrightarrow ccc$
ti

is defined as

$I(s, r)$,

where

procedure I(**value** s, **result** r) $\widehat{=}$
$\quad$ ***if*** s **is**
$\quad\quad \langle\,\rangle \quad \longrightarrow bbb$
$\quad\quad [\!] \ a{:}as \ \longrightarrow\ |[\ \mathbf{var}\ l \bullet I(as, l);\ ccc[as\backslash l]\ ]|$
$\quad$ **fi** .

Law 5.2 *assignment iterator*

If the value to be assigned to a variable is formed by the application of a catamorphism to a sequence, then the whole assignment can be implemented with an **it...ti** construct.

$$r := (\!|f, g|\!)\ s$$
$$\sqsubseteq$$
it s **into** r **with**
$\quad \langle\,\rangle \quad \longrightarrow r := f$
$\quad [\!] \ a{:}as \ \longrightarrow r := g(a, as)$
ti

Definition 5.3 *general iterator*

If t is any element of the type T defined above, then

it t **into** r **with**
  a $\longrightarrow$ *aaa*
[] b x $\longrightarrow$ *bbb*
[] c y t' $\longrightarrow$ *ccc*
ti

is defined to mean the same as

$I(t, r)$,

where

procedure I(**value** t, **result** r) $\widehat{=}$
  if t **is**
    a $\longrightarrow$ *aaa*
  [] b x $\longrightarrow$ *bbb*
  [] c y t' $\longrightarrow$ |[**var** l • $I(t', l)$; $ccc[t'\backslash l]$]|
  fi

Law 5.4 *assignment iterator*

$r := (\!|P, Q, R|\!)\ t$
$\sqsubseteq$
it t **into** r **with**
  a $\longrightarrow$ $r := P$
[] b x $\longrightarrow$ $r := Q(x)$
[] c y t' $\longrightarrow$ $r := R(y, t')$
ti

Chapter 6

Definition 6.1 *procedure type equivalence*

We extend the normal rules about type equivalence by explaining when two procedure types are type equivalent: types **proc** (**value** v : V, **result** r : R) and **proc** (**value** v' : V', **result** r' : R') are equivalent (written $\equiv$) exactly when $V \equiv V'$ and $R \equiv R'$. In other words, the parameter names are not significant, and neither are the global variables.

Law 6.2 *introduce local variable*

$$wp(\mathbf{var}\ x \bullet aaa, \phi) \\ = \forall x \bullet wp(aaa, \phi \dagger x)$$
provided ϕ *contains no* x

Definition 6.3 *procedure constant call*

$$[\![\mathbf{call}\ P(e,w)]\!]_\eta.\phi \\ = \\ \exists f, v, r, g \bullet \\ \quad \eta.P = \langle f, v, r, g\rangle \wedge \\ \quad \Phi$$

Definition 6.4 *explicit procedure expression call*

$$[\![\mathbf{call}\ (\mathbf{value}\ v, \mathbf{result}\ r \bullet p)(e,w)]\!].\phi \\ = \\ \exists f, g \bullet \\ \quad [\![p]\!] = \langle f, v, r, g\rangle \wedge \\ \quad \Phi$$

Definition 6.5 *procedure variable call*

$$\sigma \in [\![\mathbf{call}\ pv(e,w)]\!].\phi \\ = \\ \exists f, v, r, g \bullet \\ \quad \sigma.pv = \langle f, v, r, g\rangle \wedge \\ \quad \sigma \in \Phi$$

Law 6.6 *introduce procedure variable execution*

$$w : \left[\begin{array}{c} w : [pre, post] \sqsubseteq pv \\ pre \end{array}, post\right] \sqsubseteq \mathbf{call}\ pv$$

Law 6.7 *procedure variable value assignment*

If the procedure variable pv has been declared as **procedure** (**value** v), then we have the following refinement:

$$w : \left[\begin{array}{c} pre \\ w := E \sqsubseteq pv \end{array}, post\right] \sqsubseteq \mathbf{call}\ pv(A)$$
provided $w : [pre, post] \sqsubseteq w := E[v \backslash A]$

where A contains no v

Law 6.8 *procedure variable result assignment*

If the procedure variable pv has been declared as **procedure** (**result** r), then we have the following refinement:

$$a : \left[\begin{array}{c} pre \\ r := E \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(a)$$

provided $a : [pre, post] \sqsubseteq a := E$

where r does not occur in E.

Law 6.9 *procedure variable value specification*

If the procednre variable pv has been declared as **procedure** (**value** f), then we have the following refinement:

$$w : \left[\begin{array}{c} pre \\ w : [pre_1, post_1] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(A)$$

provided $w : [pre, post] \sqsubseteq w : [pre_1[f \backslash A], post_1[f_0 \backslash A_0]]$

where A_0 is $A[w \backslash w_0]$ and $post_1$ contains no f

Law 6.10 *procedure variable result specification*

If the procedure variable pv has been declared as **procedure** (**result** f), then we have the following refinement:

$$a : \left[\begin{array}{c} pre \\ f : [pre_1, post_1[a \backslash f]] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(a)$$

provided $a : [pre, post] \sqsubseteq a : [pre_1, post_1]$

where f does not occur in pre_1, and neither f nor f_0 occur in $post_1$.

Chapter 7

Definition 7.1 *procedure value substitution*

$$\begin{array}{ll} & wp(P[\textbf{value } fp \backslash AP], \phi) \\ = & \forall X \bullet X \sqsupseteq AP \Rightarrow wp(P, \phi)[fp \backslash X] \end{array}$$

Definition 7.2 *procedure result substitution*

$$\begin{array}{ll} & wp(P[\textbf{result } rp \backslash ap], \phi) \\ = & \forall\, lp \bullet wp(P[rp \backslash ap], (\forall\, ap \bullet ap \sqsupseteq lp \Rightarrow \phi) \dagger ap) \end{array}$$

Law 7.3 *procedure variable value and result specification*

If the procedure variable pv has been declared as **procedure** (**value** v, **result** r), then we have the following refinement:

$$w, ar : \left[\begin{array}{c} pre \\ w, r : [pre_1, post_1[ar \backslash r]] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(A, ar)$$

$$\textit{provided } w, ar : [pre, post] \sqsubseteq w, ar : [pre_1[v \backslash A], post_1[v \backslash A]\ ]$$

where r does not occur in pre_1, and neither r nor r_0 occnrs in $post_1$.

Law 7.4 *assignment seqiter*

If the value to be assigned to a variable is formed by the application of a catamorphism to a sequence, then the whole assignment can be implemented by a call to *seqiter*.

$$\begin{array}{l} r := (\!|f, g|\!)\ s \\ \sqsubseteq \\ \quad seqiter(\ \ (\textbf{result } er \bullet er := f), \\ \qquad\qquad (\textbf{value } a, as; \textbf{result } cr \bullet cr := g(a, as)\,), \\ \qquad\qquad r \\ \qquad)\ . \end{array}$$

Law 7.5 *procedure variable value and result specification*

If the procednre variable pv has heen declared as **procedure** (**value** v, **result** r), then we have the following refinement:

$$w, ar : \left[\begin{array}{c} pre \\ w, v, r : [pre_1, post_1[ar \backslash r]] \sqsubseteq pv \end{array} , post \right] \sqsubseteq \textbf{call } pv(A, ar)$$

$$\textit{provided } w, ar : [pre, post] \sqsubseteq w, ar : [pre_1[v \backslash A], post_1[v_0 \backslash A_0]]$$

where r does not occur in pre_1, and neither v, r nor r_0 occur in $post_1$, and A_0 is $A[w, ar \backslash w_0, ar_0]$.

Chapter 8

Law 8.1 *else distribution*

$$\begin{array}{ll} (aaa;\ bbb) \rhd ccc & \\ \sqsubseteq \quad aaa;\ (bbb \rhd ccc) & \textit{provided } ccc \sqsubseteq aaa;\ ccc \end{array}$$

Law 8.2 *disjunction-else*

$$\begin{array}{l} w : [\alpha, \beta \vee \gamma] \\ = \quad [\![\\ \qquad\quad w : [\alpha, \beta \rhd \gamma] \\ \quad\ ]\!] \end{array}$$

Law 8.3 *superfluous choice*

$$\begin{array}{ll} w : [\alpha, \beta] & \\ \sqsubseteq \quad \gamma \ \rightarrow w : [\alpha, \beta] & \\ \quad [\!] \ \neg\gamma \ \rightarrow aaa & \\ & \textit{provided } \alpha \Rightarrow \gamma \end{array}$$

Law 8.4 *absorb guard*

$$\begin{array}{ll} \alpha \rightarrow w : [\beta, \gamma] & \\ = \quad w : [\alpha \Rightarrow \beta, \alpha[w \backslash w_0] \wedge \gamma] & \\ & \textit{provided } w_0 \textit{ is not free in } \alpha \end{array}$$

Appendix A

Abbreviation A.1 *assignment abbreviation definition*

$P!$ may be used as an expression in the right-hand side of an assignment, standing for a declaration of a fresh local variable l, a call of P with l as the result parameter, followed by the assignment with l in place of $P!$:

$$\begin{array}{l} x := exp(P!) \mathrel{\widehat{=}} \ ||[\ \textbf{var}\ l \bullet \\ \qquad\qquad\qquad P(l); \\ \qquad\qquad\qquad x := exp(l) \\ \qquad\qquad]|| \end{array}$$

Law A.2 *assignment abbreviation*

If F is an expression, and P is defined by
procedure P(**result** r) $\widehat{=}$
$\quad r := E$

then

$x := F$
$\sqsubseteq$
$x := F[E \backslash P!]$.

Abbreviation A.3 *alternation guard abbreviation definition*

$P!$ may be used as an expression in the guard of an alternation, standing for a declaration of a fresh local variable l, a call of P with l as the result parameter, followed by the alternation with l in place of $P!$:

if ($[]$ $i \bullet G_i(P!) \rightarrow B_i$) **fi**
$\widehat{=}$ |[**var** $l \bullet$
$\quad P(l);$
$\quad$ **if** ($[]$ $i \bullet G_i(l) \rightarrow B_i$) **fi**
]|

Law A.4 *alternation abbreviation*

if ($[]$ $i \bullet G_i \rightarrow B_i$) **fi**
$\sqsubseteq$
if ($[]$ $i \bullet G_i[E \backslash P!] \rightarrow B_i$) **fi**

Abbreviation A.5 *iteration guard abbreviation definition*

$P!$ may be used as an expression in the guard of an iteration, standing for a declaration of a fresh local variable l, a call of P with l as the result parameter, followed by the iteration with l in place of $P!$, and a call of P after the iteration body:

do ($[]$ $i \bullet G_i(P!) \rightarrow B_i$) **od**
$\widehat{=}$ |[**var** $l \bullet$
$\quad P(l);$
$\quad$ **do** ($[]$ $i \bullet G_i(l) \rightarrow B_i; P(l)$) **od**
]|