Contents

List of Figures

Dedicated with much love to Clarissa Joyce Stevenson.

Preface

A history of this book

This book describes research work done at Westfield College, King's College and Imperial College, all in the University of London. The work was begun in about 1985, and its first written incarnation was as the author's Ph.D. thesis of the same title, examined at the end of 1987. This version is a complete rewrite. It covers the same research material, but is more introductory in nature. The literature review chapters which make Ph.D. theses so turgid has been shortened and relegated to "Pointers to the Literature" sections at the end of each major chapter.

The thesis was written under the guidance of two supervisors, both of whom I can count as good friends. Hugh Glaser was the ideal supervisor to the beginning research student, being a continuous source of ideas while always encouraging me to follow my own. In the diaspora that accompanied the destruction of the Department of Computer Science at Westfield College, Hugh left to manage the Flagship project at Imperial College, and Peter Osmon (now Professor of Computer Systems at City University) bravely stepped in as I started to write up. Peter's influence is present on almost every page—he has a special talent to help clarify one's ideas by demanding a special attention to explanations. Hugh has since moved on to Southampton University.

Chris Hankin, also originally at Westfield College but now at Imperial College, must take the credit for much of my more profound understanding of much of the material. He also played no small part in providing a stimulating environment at Imperial College where much of the thesis and book were written. Special thanks must go to Chris for being so understanding when the book got in the way of work I should have been doing for him.

Acknowledgements and thanks

More than most, this work is the fruit of many collaborations and conversations. Some quite brief discussions have had a profound influence on the nature and presentation of material presented herein. Individuals deserving particular recognition include

Paul Anderson (City University)

David Bolton (City University)

Simon Croft (King's College London)

John Darlington (Imperial College)

Tony Field (Imperial College)

Sean Hayes (Hewlett Packard, Bristol)

David Holburn (Cambridge University)

Sebastian Hunt (Imperial College)

Simon Hughes (Imperial College)

Lee McLoughlin (Imperial College)

Laurence Pearl (Cancer Research Institute, Royal Marsden Hospital)

Mike Reeve (Imperial College)

Malcolm Shute (Middlesex Polytechnic)

Andrew Smith (King's College London)

Gil Thornhill (Dept. of Civil Engineering, City University)

David Till (King's College London)

Paul Williams (AI Limited, Watford, UK)

Gil Thornhill showed astonishing tolerance during the less sociable periods of this book's creation, and made an excellent "typical reader" against whom to test my explanations. Sebastian Hunt was very helpful in giving Chapter 2 a theoretical once-over. Paul Williams contributed a great deal to the details of Chapter 5 in preparation for the work he did for his master's dissertation [Wil88]. Tony Field was responsible for checking the structure and progress of the book, and has made a fine job of proof-reading it. Finally, many thanks to Donald Knuth, Leslie Lamport, Richard Stallman and the Free Software Foundation for the use of their their software systems TEX, LATEX and GNU Emacs.

My apologies must go to all those—especially Judy, Rebecca and Wayne and my family—whom I have neglected in writing this book. I dread the thought that you might read it and decide whether it's all been worthwhile!

Funding

The beginnings of this work were funded by an SERC Research Studentship. It was completed with the help of the COBWEB project, funded by the UK Alvey Programme (with thanks to Chris Hankin), other Alvey Programme funding (with thanks to John Darlington), and the Department of Computing, Imperial College.

Jokes

There are no jokes in this book.

Chapter 1

Introduction

The two aims, on the one hand for highly-parallel hardware, and on the other for easy and speedy creation of high-quality software, are seen by many to be directly antithetic. J.P. Eckert wrote, when arguing for parallel data transfer and arithmetic in computers of EDVAC's generation, that

> The arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are controlled by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is far too complicated to use [Eck46].

The quest to overturn this wisdom, which had been learned "over and over again" in 1946, has occupied a large portion of the computer science community since then. Why is parallel programming difficult?

- **Performance:** The performance of a parallel program is difficult to optimise—counting the number of instructions is no longer good enough, because some of the instructions may be executed simultaneously.
- **Portability:** There are many more ways in which two parallel computers may differ, and these can mean that quite different algorithms are suitable for different target architectures.
- **Determinacy:** The order of events during parallel program execution is almost always indeterminate. The program's output is determinate only if it is written carefully.

All of these problems do arise to some extent when programming sequential computers, but in the general case of parallel computing they are epidemic.

1.1 Functional programming

The main subject of this book is the interesting and powerful class of functional programming languages. The reason for choosing such a language is the ease with which

such programs can be manipulated algebraically, and the bulk of the book is devoted to introducing and demonstrating this in action.

It is through algebraic manipulation of programs that the problems of parallel programming are addressed. We retreat from the hope that a single program will serve for all the different parallel computers we might wish to use, and instead begin with a single specifying program. Versions for different target architectures can then be derived by the application of a toolbox of mathematical transformations to the specification, leading to versions tuned to the various machine structures available. The transformation pathways can then be re-used when modifications to the specification are made.

1.2 Loosely-coupled multiprocessors

Parallel programming is much simplified if we can assume that interprocessor communication is very efficient, as in a shared memory multiprocessor. This book is about programming a much larger class of computers for which such simplifying assumptions do not hold. In general, there are two distinct problems in mapping a parallel program onto a computer: *partitioning* and *mapping*. The most important simplifying assumption often made is to avoid mapping, and assume that performance is independent of where processes are placed. The class of loosely-coupled multiprocessors is defined to characterise architectures where this assumption is not valid: a loosely-coupled multiprocessor is a collection of processing elements (PEs), linked by an interconnection network with the property that communication between "neighbouring" PEs is much more efficient than communication between non-neighbours. Depending on the interconnection network's topology, there are many varieties of such an interconnection network. The important feature is that not all PEs are local to one another, so that process placement is important to program performance.

The importance of this class of architectures is that they are easy and inexpensive to build on a large scale. It is not, therefore, surprising to find quite a number of loosely-coupled multiprocessors on the market and in use. Examples include Meiko's *Computing Surface*, Parsys's *Supernode* and Intel's *iPSC*.

In architectures of this kind the full generality of the software design problems for parallel computers become apparent. We find that data communication is often a primary computational resource, and that much of the algorithm design effort is aimed at reducing a program's communications demands. Several examples are given of how this can be done using program transformation. The techniques have application to other parallel architectures including more closely-coupled machines and SIMD computers.

1.3 Neighbour-coupled multiprocessors

A neighbour-coupled multiprocessor is a more idealised abstract computer architecture, and is introduced here as an experiment. A neighbour-coupled multiprocessor is a loosely-coupled multiprocessor, where each PE is very closely coupled to its neighbours, so closely that the programmer can assume that a PE can read and write its neighbour's memory as quickly as its own.

We shall return to this abstract architecture later in the book to examine whether it allows useful simplifications.

1.4 A reader's guide

The book consists of the following components:

- **Chapter 2. Functional Programming:** This chapter introduces functional programming from first principles. The programming language is presented by means of examples. Simple techniques are given for manipulating programs to modify their structure while retaining the same input/output mapping. These are augmented by a handful of induction rules for proving generic properties about programs.

  The language is based on Miranda[1] and Haskell (a public-domain language design for which a specification is in preparation [HWA+88]).

- **Chapter 3. Sequential and Parallel Implementation Techniques:** The aim of this chapter to sketch how our functional language might be compiled to run efficiently on a conventional computer, and to examine how this scheme (graph reduction) might be extended for a tightly-coupled multiprocessor.

- **Chapter 4. Specifying and Deriving Parallel Algorithms:** This chapter examines how parallelism and inter-process communication are manifest in a functional program script. Horizontal and vertical parallelism are identified and examples are given in the form of divide-and-conquer and pipeline algorithms respectively. The main emphasis in this chapter is the development of program transformation techniques. Examples are given of introducing pipeline parallelism, and of transforming a divide-and-conquer algorithm into a cyclic "process network" program. This is illustrated by application to a simple ray tracing program.

- **Chapter 5. Distributed Parallel Functional Programming:** We can write programs for which a good placement onto a loosely-coupled multiprocessor can be made. This chapter applies a declarative programming language approach to actually specifying this placement. It incorporates abstraction mechanisms to give concise mappings for regular architectures and algorithms. The notation is illustrated with several examples.

- **Appendix A. Proofs and Derivations:** This appendix gives proofs and derivations which would have cluttered the presentation given in chapter 4. Although quite dense later on, the earlier material in this chapter is quite tutorial in nature and might be read concurrently with Chapter 4 by those more interested in program derivation and verification than in parallel programming.

- **Appendix B. Common Definitions:** This appendix lists widely-used function definitions for easy reference.

[1] Miranda is a trademark of Research Software Ltd.

- **Appendix C. Programming in a real functional language:** The functional language used in this book is not quite compatible with any commonly-available language implementation. This appendix lists the small (and quite innocuous) differences from Miranda in order to aid a reader who wishes to experiment.

Each chapter ends with some pointers for the interested reader towards other books, articles and research papers which might be of interest.

Chapter 2

Functional Programming

This chapter gives a tutorial introduction to functional programming as employed in this book. It deals with the programming language, functional programming techniques, and the mathematical transformation and verification of functional programs. Finally, a review of the success of the functional approach is given.

2.1 The programming language

The functional language used in this book is representative of a class of programming languages, rather than being any one in particular. We will, however, stick as closely as possible to the notation used in [BW88]. Their excellent book is recommended to the reader needing a more detailed and introductory guide to functional programming. Apart from some minor typographical details, which are summarised in Appendix C, the language employed is a simple subset of Miranda.

To summarise its main features, the language is:

- **Functional**: a program comprises an expression, and a set of equations defining functions, values and types required to give the expression meaning.
- **Higher-order**: a function can appear anywhere where a value can appear, notably as a parameter to a function, or as its result.
- **Curried**: a function expecting two parameters is normally defined so that the parameters may be provided one-by-one, so that it may be specialised to its first parameter by simple application.
- **Lazy**: a function's parameter is evaluated only when its value is needed for the program to produce its next item of output (and then it is evaluated only once).
- **Typed**: a program can never fail at run-time due to a type error. Types are inferred automatically, at compile-time, and are checked against optional type declarations when present.

2.2 Equations

A program, called a *script*, defines types, functions and values by means of equations. New types can be defined in terms of old ones by *type equations*. For example,

```
Date == (Day, Month, Year)
```

defines a new type, Date whose elements are tuples of length three, comprising elements of the types Day, Month and Year. An example of an element of the type might be

```
birthday = (18, 8, 61)
```

This mechanism is often used just to give a synonym for a built-in type such as the numbers Num, as in these definitions:indexdefinitions!of type synonyms

```
Day == Num
Month == Num
Year == Num
Price == Num
```

(To simplify matters we will not distinguish different kinds of numbers here). Types whose values may take one of several forms can be defined using a simple notation derived from the BNF language for defining grammars[1]. For example:

```
Class ::= FIRST | SECOND
Ticket ::= PLATFORM |
           SINGLE Class Date Price Destination |
           RETURN Class Date Price Destination Period
```

As with a grammar, such types can be recursively defined:indexdefinitions!of data types

```
ListOfNum ::= NIL | CONS Num ListOfNum
```

An element of the type ListOfNum is either the empty list, denoted by NIL, or is built from a number and another element of the ListOfNum type. CONS and NIL serve to distinguish the two cases, and are called *constructors*. Throughout this book, constructors will be written in upper case to distinguish them from other variables.

Data types may have type variables, given names α, β, γ etc., strong polymorphic For example we can define a list of objects of arbitrary type by writing

```
List α ::= NIL | CONS α (List α)
```

The type for a list of numbers can now be referred to simply as List Num. A list of characters would have the type List Char. We can define the type variable Destination by writing

[1] A glossary of symbols is collected in Appendix B.

```
Destination == List Char
```

We can define a variable with this type by writing, for example,

```
home = CONS 'A' (CONS 't' (CONS 'h' (CONS 'e' (CONS 'n' (CONS 's')))))
```

However, because they are so useful, we use a special notation for lists, in which [ ] denotes the empty list NIL, and where : is an infix version of CONS. Thus the definition of home is:

```
home = 'A' : 't' : 'h' : 'e' : 'n' : 's' : [ ]
```

We can take this further and write the elements of a list inside square brackets. Hence,

```
home = ['A', 't', 'h', 'e', 'n', 's']
```

A list of characters, has, of course, the obvious shorthand:

```
home = "Athens"
```

Lists can be defined recursively. For example,

```
sawtooth = 1 : 2 : 3 : 4 : 5 : sawtooth
```

defines the infinitely-long list

```
sawtooth = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, ...]
```

Notice how this mirrors the idea of a communications channel, or a wire in a digital system. The list models a sequence of values *in time*, and : can be read as "followed by". Such lists—often called "streams"—give us the power to express the behaviour of a process as a function mapping a stream to a stream.

Definitions can be parameterised. For example, the equation

```
from n = n : ( from (n + 1) )
```

defines from n to be the list of integers starting from n. More generally, functions (like from) are defined by more than one equation:

```
exp 0 x = 1
exp (n+1) x = x × (exp n x)
```

In this book we will be careful to write such definitions so that at most one left-hand side can possibly match any particular expression. For this reason, unless the first parameter of exp is restricted to the natural numbers (i.e. the integers ≥ 0), a better way to write the definition above is:

```
exp n x = 1,                    if n = 0
exp n x = x × (exp (n−1) x),    if n > 0
```

The Boolean expressions are called *guards*, which must be satisfied before the corresponding equation can be applied. They must be mutually-exclusive. The keyword otherwise is a shorthand for the guard which succeeds when all others fail[2].

Functions over data types can be defined using pattern-matching on the LHS. For example, the function map is defined below by two equations, one for the two possible forms its list parameter may take:

```
map f [ ] = [ ]
map f (x:xs) = (f x) : (map f xs)
```

map applies its function parameter f elementwise to its list parameter. We can summarise its behaviour informally by writing

```
map f [a1, a2, ...an] = [f a1, f a2, ...f an]
```

or even,

```
map f [ ...ai ...] = [ ...f ai ...]
```

To illustrate map in use, we must supply it with a single-parameter function. We can use + as a prefix operator by enclosing it in parentheses, so that

```
(+) 3 5 = 3 + 5
```

By writing (+) 3 we denote the function which adds 3 to its parameter (a technique called *partial application*, or sometimes *currying*, after the logician H.B. Curry). Thus,

```
map ((+) 3) [1, 2, 5, 10, 20, 50, 100] = [4, 5, 8, 13, 23, 53, 103]
```

and similarly,

```
map ((×) 3) sawtooth = [3, 6, 9, 12, 15, 3, 6, 9, 12, 15, 3, 6, 9, 12, 15, 3, 6 ...]
```

Constructors can be curried in just the same way as can ordinary functions. For example,

```
map (CONS 'f') ["lame", "lies", "airy"] = ["flame", "flies", "fairy"]
```

(remembering that "···" is shorthand for a list of characters.)

[2] A complete programming language would include a shorthand allowing equations to be prioritised, to allow overlapping equations and non-exclusive guards to be used. See Appendix C, section C.2

2.2.1 Types and Type Checking

We employ a strongly-typed language, that is, type errors cannot occur at run-time. Type specifications are optional, and we will usually give them. If not provided by the programmer, a variable's type is inferred automatically from its definition and use.

A compiler can infer automatically, for example, that sawtooth is a list of numbers. We can provide its type specification explicitly as follows:

sawtooth :: list Num

For "::" read "has the type". The list type is expressed in a natural shorthand: [Num] denotes the type list Num.

It is easy to see that the partial application (+) 3 is a function from numbers to numbers. We can assert this by writing a type specification:

(+) 3 :: Num → Num

Because it is applied to 3, we can infer the type for (+):

(+) :: Num → (Num → Num)

As another example, take the append function (infix version "++"), which joins two lists together:

append [a1, a2, ...an] [b1, b2, ...bn] = [a1, a2, ...an, b1, b2, ...bn]

Now

append [a1, a2, ...an] [b1, b2, ...bn] :: [α]

therefore the partial application of append to its first parameter only must have the type

append [a1, a2, ...an] :: [α] → [α]

and therefore append itself should have the type specification

append :: [α] → ([α] → [α])

In general, if f is a function which takes n parameters, with types α, β ... ψ, and returns a result of type ω, its type is:

f :: $\alpha \to \beta \to \gamma \to \cdots \to \psi \to \omega$

For convenience, we assume that → associates to the right, so to understand this type specification, re-insert the missing brackets:

f :: $\alpha \to (\beta \to (\gamma \to \cdots \to (\psi \to \omega)) \cdots))$

This notation may seem slightly counter-intuitive, but arises quite naturally from the need to assign a type to a partial application.

Because the $\rightarrow$ operator is not associative, brackets are necessary when parameters are themselves functions. For example, from the definition of map the compiler infers the type specification

$$\mathsf{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$

As with the definition of the list type, type variables α and β show where a consistent substitution with actual types can be made. The function map takes two parameters, a function (type $\alpha \rightarrow \beta$) and a list (type $[\alpha]$). It returns a list, of type $[\beta]$.

While type specifications are not strictly necessary, and even in strongly-typed languages can still be optional, they will always be given from here onwards.

2.2.2 Block structure: where clauses

It is often useful to abstract a subexpression to avoid writing it twice. For example:

```
f :: Num → Num → Num

f a x = b + x × (b + x × (b + x × b))
        where
        b = a × a + 1
```

Note that

- A **where** clause can be associated with the right-hand side of an equation.
- The scope of a **where** clause (i.e. the extent of the script over which its definitions are to be applied) is defined to be the right-hand side of the equation to which it is attached.
- A **where** clause may comprise several definitions of functions and values, but may not introduce new types.

2.2.3 The layout rule

We employ a simple rule to avoid ambiguity in **where** clauses: the right-hand side of an equation must remain strictly to the right of the equation's "=" sign—even if it spills over onto several lines. This rule applies to equations nested inside **where** clauses as well as at the top-level. For example:

```
ticket :: Ticket
ReturnTo :: Destination → Date → Class → Ticket
AwayDay :: Ticket → Ticket
```

```
ticket = AwayDay (ReturnTo "Buxton" today SECOND)
         where
         ReturnTo dest date class = RETURN class date ReturnPrice dest period
                                    where
                                    ReturnPrice = (PriceOf dest) × 2
                                    period = 90

         AwayDay (RETURN class date price dest period)
                 = RETURN class date (price × reduction) dest 1
                   where
                   reduction = 2/3
```

It is very rare indeed that deeply-nested **where** clauses are desirable or necessary.

Ordinary systems of definitions are valid **where** clauses, and may be recursive. Note that as well as functions, it is often also useful to define values recursively, an example being sawtooth as given above.

2.2.4 Reduction

If there is any doubt about the value an expression should have, one can always calculate it. We apply the equations which make up the script to simplify the expression, successively replacing an instance of an equation's LHS by the corresponding RHS. Let us take a simple example without the list shorthand:

```
map ((+) 3) (1 : (2 : (5 : 10 : [ ])))
```

Now the second of the two equations defining the map function can be applied, since its left-hand side matches the parameter value supplied. We bind f to ((+) 3), x to 1 and xs to (2 : (5 : 10 : [ ])), and substitute in the right-hand side to yield

```
(((+)3)1) : (map ((+) 3) (2 : (5 : 10 : [ ])))
```

We call an expression which matches some left-hand side a *reducible expression*, or *redex* for short. The resulting expression contains several redexes, of which the first (marked by the brace) is an application of the built-in addition operator:

```
(((+) 3) 1) = 4
```

so we have

```
4 : (map((+)3)(2 : (5 : 10 : [ ])))
```

(by convention, the brace marks the expression *next* to be rewritten). At this point, we know that the first element of the list is 4. To find the next element, re-apply the equation defining map:

$$4 : \underbrace{(((+)3)2)} : (\mathsf{map}\ ((+)\ 3)\ (5 : 10 : [\,]))$$

That is:

$$4 : 5 : \underbrace{(\mathsf{map}((+)3)(5 : 10 : [\,]))}$$

Now we know the second element is 5. Repeat to find the third and fourth:

$$\begin{aligned}
&4 : 5 : \underbrace{(((+)3)5)} : (\mathsf{map}\ ((+)\ 3)\ (10 : [\,])) \\
&= 4 : 5 : 8 : \underbrace{(\mathsf{map}((+)3)(10 : [\,]))} \\
&= 4 : 5 : 8 : \underbrace{(((+)3)10)} : (\mathsf{map}\ ((+)\ 3)\ [\,]) \\
&= 4 : 5 : 8 : 13 : \underbrace{(\mathsf{map}((+)3)[\,])} \\
&= 4 : 5 : 8 : 13 : [\,]
\end{aligned}$$

The final reduction made use of the first equation defining map. There are no more redexes: the expression is in *normal form*.

Notice that during reduction the equations are not treated symmetrically: an instance of an RHS is not rewritten to the corresponding LHS. A reduction process which includes such steps may fail to terminate when it should, although it cannot derive incorrect results.

Reduction forms the basis for most implementations of functional programming languages, and after extensive optimisation it can be done very efficiently indeed.

2.2.5 Pattern matching and reduction order

At each stage during reduction, several equations may apply to the expression at the same time. If we are to use reduction to define the meaning of an expression, there are some important questions to answer. Does it matter in what order the reductions are performed? How do we make sure the reductions we do contribute to the result, rather than to an expression which is ultimately discarded? Fortunately, the theory of such systems gives us some very strong properties. We must, however, obey the mutual exclusion rule for writing equations: of all the equations defining a variable, at most one may ever apply. Thus, we cannot write

```
either x y = x
either x y = y
```

nor

```
SpecialCase 818 = TRUE
SpecialCase 242 = TRUE
SpecialCase n = FALSE
```

With guards the responsibility rests with the programmer to ensure mutual exclusion; in general, the compiler cannot verify that a definition like

```
f x = TRUE,     if g1 x
f x = FALSE,    if g2 x
```

is allowable.

Provided mutual exclusion is satisfied, the following properties hold:

- **Confluence:** No matter what order we apply applicable equations to an expression, it is always possible to reach the expression's normal form, if it has one. It is impossible to reduce an expression to two different normal forms. This property is sometimes called the Church-Rosser property.

- **A general normalisation strategy:** If an expression has a normal form, it can be found by the following strategy:

  1. Identify the outermost reducible function application. This consists of a known function identifier, say f, and zero or more parameters p_1, p_2 ... p_n. The parameters need not be known.
  2. Test the application against each of f's defining equations. Each of the tests is performed in parallel, evaluating parameters as necessary. At most one will terminate signaling success. The other tests may terminate signaling failure, or may fail to terminate. Once a winning test has been identified, the other test processes can be abandoned. Thanks to the mutual exclusion rule, we know that their can be only one successful test.

  (A reducible function application is a function identifier applied to as many parameters as appear in the function's defining equations).

A good compiler can analyse the patterns concerned and avoid having to race parallel processes. A sequential scan can be used instead. Most practical functional languages only allow patterns which can be sequentialised in this way.

2.2.6 More definitions to think about

Before moving on to program transformation, here are some simple definitions to illustrate the language in use. The definitions of these and other handy "building blocks" are collected in Appendix B.

List projectors

Because they model a sequence in time, lists are a fundamental concept in functional programming, especially in this book. Extensive use will be made of these two functions to decompose them:

$\mathsf{hd} :: [\alpha] \to \alpha$
$\mathsf{tl} :: [\alpha] \to [\alpha]$

$\mathsf{hd}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x}$
$\mathsf{tl}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{xs}$

The functions hd and tl are called the *projectors* of the list data type, and satisfy the equation

$\mathsf{for\ all\ as} \in [\alpha], \mathsf{as} \neq [\ ]:$
$\qquad \mathsf{as} = (\mathsf{hd\ as}) : (\mathsf{tl\ as})$

We can prove this (informally) using the equations defining hd and tl by making the substitution (which makes the assumption that as does evaluate to some known list),

$\mathsf{b} : \mathsf{bs} = \mathsf{as}$

giving us

$\mathsf{b} : \mathsf{bs} = (\mathsf{hd}\ (\mathsf{b} : \mathsf{bs})) : (\mathsf{tl}\ (\mathsf{b} : \mathsf{bs}))$

Using the equations defining hd and tl this follows immediately.

Generalising "+" over lists

It is natural to generalise an operator like + to add corresponding elements of a pair of lists of numbers. If we have, for example,

$(+) :: \mathsf{Num} \to \mathsf{Num} \to \mathsf{Num}$

and two lists of numbers,

$\mathsf{as} = [\mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \ldots \mathsf{an}]$
$\mathsf{bs} = [\mathsf{b1}, \mathsf{b2}, \mathsf{b3}, \ldots \mathsf{bn}]$

we would want the result of generalising + to lists to be

$\mathsf{map2}\ (+)\ \mathsf{as}\ \mathsf{bs} = [(+)\ \mathsf{a1}\ \mathsf{b1}, (+)\ \mathsf{a2}\ \mathsf{b2}, (+)\ \mathsf{a3}\ \mathsf{b3}, \ldots (+)\ \mathsf{an}\ \mathsf{bn}]$

That is,

$\mathsf{map2}\ (+)\ \mathsf{as}\ \mathsf{bs} = [\mathsf{a1}+\mathsf{b1}, \mathsf{a2}+\mathsf{b2}, \mathsf{a3}+\mathsf{b3}, \ldots \mathsf{an}+\mathsf{bn}]$

This function map2 is defined by the equations

```
map2 :: (α → β → γ) → [α] → [β] → [γ]

map2 op (a : as) (b : bs) = (op a b) : (map2 op as bs)
map2 op [ ] [ ] = [ ]
```

It is called map2 because it is a natural extension of map to functions of two parameters.

Our next definition is a generalisation of function application to lists of functions and lists of parameters:

```
ply :: [(α → β)] → [α] → [β]

ply [ ] [ ] = [ ]
ply (f : fs)(x : xs) = (f x) : (ply fs xs)
```

A very alert reader might realise that

```
ply = map2 apply
      where
      apply f x = f x
```

We can verify that this is so by substituting

```
op = apply
```

in the definition of map2. This yields a definition identical in structure to the explicit definition of ply, except that map2 apply now appears where ply did.

Insertion

An important family of operations (often called *folding* or, confusingly, *reduction*) concern inserting an operator between adjacent pairs of elements of a list. Suppose op is an infix operator akin to +. An intuitive definition of such an insertion function might be

```
insert (op) base [ ] = base
insert (op) base [a1, a2, a3, ··· aN] = a1 op a2 op a3 ··· op aN
```

This is ambiguous in general, since we have not specified the bracketing to be applied in the RHS—we have left some freedom in the reduction order. This makes no difference if op is associative:

```
for all a, b, c
    a op (b op c) = (a op b) op c
```

The list joining operator ++ is associative, for example, but subtraction is not. Strictly, addition of integers is associative only if it is implemented correctly for values of arbitrary size. If overflow can occur, the order in which a list of numbers is added can affect the result. Note that all associative functions have the type

$\alpha \to \alpha \to \alpha$

An associative function must take parameters of the same type as each other and as its result. Thus, the type of insert is

insert :: $(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$

When the function being inserted is not associative, we must choose an ordering for the brackets. There are two sensible options: we can associate the operator to the left or to the right. We define variants of insert for each option. For example,

```
insertleft (op) base [a1, a2, a3, a4, a5, a6]
                    = (((((base op a1) op a2) op a3) op a4) op a5) op a6
```

and

```
insertright (op) base [a1, a2, a3, a4, a5, a6]
                     = a1 op (a2 op (a3 op (a4 op (a5 op (a6 op base)))))
```

For non-associative operators, it is more common to use the usual prefix form of function application, so that

```
insertleft f base [a1, a2, a3, a4, a5, a6]
                  = f (f (f (f (f (f base a1) a2) a3) a4) a5) a6
```

and

```
insertright f base [a1, a2, a3, a4, a5, a6]
                   = f a1 (f a2 (f a3 (f a4 (f a5 (f a6 base)))))
```

Their definitions are

insertleft :: $(\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$

```
insertleft f base [ ] = base
insertleft f base (a : as) = insertleft f (f base a) as
```

and

insertright :: $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$

```
insertright f base [ ] = base
insertright f base (a : as) = f a (insertright f base as)
```

(In insertleft notice how the base parameter is used to accumulate the result so far). Because the function f need not be associative, the types of its parameters need not be the same. For insertleft it must be

f :: $\alpha \to \beta \to \alpha$

while for insertright it must be

f :: $\alpha \to \beta \to \beta$

In either case, base is needed to form a "seed" value from which to build a result of the right type.

Here are a handful of examples. With associative operators we can use insert and leave the choice of insertleft or insertright or whatever free. To sum the elements of a list, write

```
sum :: [Num] → Num

sum as = insert (+) 0 as = insertleft (+) 0 as = insertright (+) 0 as
```

To join up all the lists in a list of lists, write

```
join :: [[α]] → [α]

join as = insert (++) [ ] as
```

To reverse a list, try

```
reverse :: [α] → [α]

reverse as = insertright postpend [ ] as
             where
             postpend a as = as ++ [a]
```

The intuition behind insertleft's operation is that the result is formed by repeatedly building on the base using successive elements of the list, starting from the end. For example, suppose we want to count the frequency of occurrence of integers between 0 and range in a list, in order to build a histogram. Define

```
histogram :: Num → [Num] → [Num]

histogram range data = insertleft IncrementBucket EmptyBuckets data
                       where
                       EmptyBuckets = replicate range 0
                       IncrementBucket buckets n = MapElement ((+) 1) n buckets
```

EmptyBuckets is a list of range zeroes, constructed using replicate:

```
replicate :: Num → α → [α]

replicate 0 x = [ ]
replicate (n+1) x = x : (replicate n x)
```

IncrementBucket buckets n adds one to the n^{th} element of the list of frequencies buckets. It uses the more general-purpose function MapElement,

```
MapElement :: (α → α) → Num → [α] → [α]

MapElement f 0 (x : xs) = (f x) : xs
MapElement f (n+1) (x : xs) = x : (MapElement f n xs)
```

Bird and Wadler give an excellent coverage of the insertion functions, which they call *foldl* and *foldr*, in their textbook [BW88].

2.2.7 Recurrences

We have now seen a couple of ways of capturing common computational structures in our functional notation, but it is still not obvious how to express simple calculations such as those computed by loops in an imperative language. There are several ways of doing this, but here we are going to introduce an "idiom"—a clear and commonly-understood way to express iteration. The idiom is developed by means of two examples: the calculation of the Fibonacci numbers, and the application of the Newton-Raphson method to the calculation of square roots.

Example: the Fibonacci numbers

The n^{th} Fibonacci number is defined by a recurrence relation:

```
fib 0 = 1
fib 1 = 1
fib n = (fib (n−-1)) + (fib (n−2)),          if n ≥ 2
```

This mathematical definition serves as a computational definition, and when executed gives the desired result. It does take a very long time when given larger parameters because each recursive invocation of fib recalculates many values already computed elsewhere. To construct a more sensible program we need to make sure that these values are saved for re-use. Let's build them into a list, fibs, so that the n^{th} element of fibs contains fib n. The list is defined by the equations

```
fibs sub 0 = 1
fibs sub 1 = 1
fibs sub n = (fibs sub (n−1)) + (fibs sub (n−2)),          if n ≥ 2
```

where **sub** is the list indexing operator,

```
(a : as) sub 0 = a
(a : as) sub (n+1) = as sub n
```

This definition of fibs is not quite a valid definition in our language, because of the use of **sub** on the left hand side. One approach might be to extend the language to allow it, but a simple definition makes this an unnecessary luxury. Define

```
generate :: (Num → α) → [α]

generate f = map f (from 0)
```

(recall that from n computes the list of integers starting from n). Informally,

```
generate f = [f 0, f 1, f 2, ···]
```

Now, we can define the list of Fibonacci numbers by writing

```
fibs = generate NextFib
       where
       NextFib 0 = 1
       NextFib 1 = 1
       NextFib n = (fibs sub (n−1)) + (fibs sub (n−2)),        if n ≥ 2
```

All that remains is to pick out the Fibonacci number we wanted in the first place,

```
fib n = fibs sub n
```

Notice, of course, that only a limited number of elements of fibs need to be calculated before fib n is found.

Example: Newton-Raphson approximation

The Fibonacci example corresponds to a for loop in an imperative language, because the number of iterations (n) is fixed beforehand. The Newton Raphson example corresponds to a while loop in an imperative language, where the number of iterations is determined by testing some condition at each iteration.

To calculate the square root of a using the Newton-Raphson method, we solve the equation

$$x^2 - a = 0$$

by defining a function f x = x² − a, and its derivative, f' x = 2 × x, and forming the series defined by

$$\begin{aligned} x_0 &= x/2 \\ x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} = \frac{x_i + a/x_i}{2} \end{aligned}$$

($x/2$ is just an initial guess). Translating this into the programming notation, using generate, gives

```
xs = generate NextEstimate
     where
     NextEstimate 0 = x/2
     NextEstimate (i+1) = ((xs sub i) + a/(xs sub i))/2
```

The square root is the limit of the series, defined to be the value of x_i such that

$$|(x_i - x_{i-1})/x_i| \leq \epsilon$$

for some given value of ϵ. We can find this value by defining a function until,

```
until :: (Num → Bool) → [Num] → Num

until predicate xs = select (map predicate (from 0)) xs
                     where
                     select (FALSE : tests) (x : xs) = select tests xs
                     select (TRUE : tests) (x : xs) = x
```

This function finds the first element of xs which satisfies predicate. Now the square root function as a whole is given by

```
sqrt a = until converges xs
         where
         converges 0 = FALSE
         converges (i+1) = abs( ((xs sub (i+1)) − (xs sub i))/(xs sub(i+1)) ) ≤ ε
         xs = generate NextEstimate
              where
              NextEstimate 0 = a/2
              NextEstimate (i+1) = ((xs sub i) + a/(xs sub i))/2
```

This expresses the iteration rather neatly, keeping quite close to the original mathematics. A particularly pleasing feature is that if a more complex solution scheme were employed which involved references to x_{i-2} and x_{i-3}, for example, very little change is required.

There does in fact remain one inefficiency in this idiom: the list indexing operation **sub** must step through the list to find the n^{th} element, and as this is done at each iteration anew a great deal of unnecessary work is involved. For reasonable uses of **sub** in such recurrences, this can be removed by a straightforward program transformation which is shown in section 4.5.1 and in Appendix A, section A.3. We assume that this is done by the compiler.

Iteration

When each successive state depends only on the previous state, an especially simple form of recurrence applies. Define

```
iterate :: (α → α) → α → [α]

iterate f x = x : (iterate f (f x))
```

so that

```
iterate f x = [x, f x, f (f x), f (f (f x)), f (f (f (f x))), ...]
```

We will see later that it is sometimes useful, particularly when looking for parallelism, to transform iterate into a circular form:

```
iterate f x = output
              where
              output = x : (map f output)
```

The definition of a variable in terms of itself may seem surprising. Compare it with this (somewhat convoluted) definition of the factorial function:

```
fac :: Num → Num
f :: (Num → Num) → Num → Num

fac x = f fac x
        where
        f g x = 1,                    if x = 0
              = x × (g (x − 1)),      if x > 0
```

Just as a function can be defined recursively, so can any other value. One can think of this as an aspect of "equal rights" for all the programming language's objects.

2.2.8 Vectors and matrices

A vector is similar to a list, but is designed for efficient access and construction. A matrix is a two-dimensional array with similar properties. Higher-dimensional arrays can be built from vectors of vectors, matrices of matrices etc.

The size of a vector v is represented by a number n, and its elements are indexed v **sub** 0 ... v **sub** (n−1), just as with lists. A vector can be created by the function MakeVector:

```
MakeVector :: Num → (Num → α) → <α>

MakeVector bound f = <f 0, f 1, f 2, ... , f (bound−1)>
```

The bound of a vector can be found using the function VectorBound:

```
VectorBound <α> → Num
```

The size of a matrix m is represented by a pair of numbers, MatrixBounds m = (xBnd,

yBnd), and its elements are indexed in a similar way:

$$\begin{array}{lllll} \ll & \mathsf{m}\ \underline{\mathbf{sub}}\ (0,0), & \mathsf{m}\ \underline{\mathbf{sub}}\ (1,0), & \ldots & \mathsf{m}\ \underline{\mathbf{sub}}\ (\mathsf{xBnd}-1,0), \\ & \mathsf{m}\ \underline{\mathbf{sub}}\ (0,1), & \mathsf{m}\ \underline{\mathbf{sub}}\ (1,1), & \ldots & \mathsf{m}\ \underline{\mathbf{sub}}\ (\mathsf{xBnd}-1,1), \\ & & \vdots & & \\ & \mathsf{m}\ \underline{\mathbf{sub}}\ (0,\mathsf{yBnd}-1), & \mathsf{m}\ \underline{\mathbf{sub}}\ (1,\mathsf{yBnd}-1), & \ldots & \mathsf{m}\ \underline{\mathbf{sub}}\ (\mathsf{xBnd}-1,\mathsf{yBnd}-1) \quad \gg \end{array}$$

A matrix is created using the function MakeMatrix:

$$\mathsf{MakeMatrix} :: (\mathsf{Num}, \mathsf{Num}) \to ((\mathsf{Num}, \mathsf{Num}) \to \alpha) \to \ll\alpha\gg$$

and is defined so that

$$\begin{array}{llllll} \mathsf{MakeMatrix}\ (\mathsf{xBnd},\mathsf{yBnd})\ \mathsf{f} & & & & & \\ = & \ll & \mathsf{f}\ (0,0), & \mathsf{f}\ (1,0), & \ldots & \mathsf{f}\ (\mathsf{xBnd}-1,0), \\ & & \mathsf{f}\ (0,1), & \mathsf{f}\ (1,1), & \ldots & \mathsf{f}\ (\mathsf{xBnd}-1,1), \\ & & & \vdots & & \\ & & \mathsf{f}\ (0,\mathsf{yBnd}-1), & \mathsf{f}\ (1,\mathsf{yBnd}-1), & \ldots & \mathsf{f}\ (\mathsf{xBnd}-1,\mathsf{yBnd}-1) \quad \gg \end{array}$$

Example: integration by Simpson's rule

A vector or matrix can be defined using a recurrence in exactly the same way we used a list earlier. This function integrates $f(x)$ over the range $a \to b$ using Simpson's rule with a step length h:

$$\begin{array}{l} \mathsf{integral\ f\ a\ b\ h} = \mathsf{MakeVector}\ ((\mathsf{b}-\mathsf{a})/\mathsf{h})\ \mathsf{NextElement} \\ \qquad\qquad \mathbf{where} \\ \qquad\qquad \mathsf{NextElement\ 0} = \mathsf{0} \\ \qquad\qquad \mathsf{NextElement\ n} = (\mathsf{integral}\ \underline{\mathbf{sub}}\ (\mathsf{n}-1)) \\ \qquad\qquad\qquad\qquad + (\mathsf{h}/3)\times((\mathsf{f}\ (\mathsf{x}-\mathsf{h})) \\ \qquad\qquad\qquad\qquad\qquad + 4\times(\mathsf{f\ x}) \\ \qquad\qquad\qquad\qquad\qquad + \mathsf{f}\ (\mathsf{x}+\mathsf{h})), \qquad \mathbf{if}\ \mathsf{n} \geq 1 \\ \qquad\qquad\qquad\qquad \mathbf{where} \\ \qquad\qquad\qquad\qquad \mathsf{x} = \mathsf{n}\times\mathsf{h} + \mathsf{a} \end{array}$$

This definition recomputes f three times at each point, and we can use the same technique we used with fib to avoid it by introducing a list:

```
integral f a b h = MakeVector ((b−a)/h) NextElement
                   where
                   NextElement 0 = 0
                   NextElement n = (integral sub (n−1)) +
                                   (h/3)× ((fs sub (n−1))
                                           + 4×(fs sub n)
                                           + (fs sub (n+1))),        if n ≥ 1
                   fs = generate fn
                        where
                        fn n = f (n×h + a)
```

There is no problem mixing generate'd lists and MakeVector'ed vectors in the same recurrence. The difference between them is that the space occupied by a vector is used as long as any element is referred to, while early parts of a list which are no longer needed can be reclaimed. Generally, a list is a better choice if all that is required for output is the final state, but a vector is good for when the entire course of values is to be presented as output.

Matrices can also be defined by recurrences, in many interesting ways. For example, in applying the Gauss-Seidel method to the solution of linear simultaneous equations, a matrix is built whose "South" and "West" boundary is defined independently, but whose internal elements depend on their South and West neighbours. This comes out very easily:

```
GaussSeidel f a
 = MakeMatrix (Bound, Bound) NextElement
   where
   NextElement (0,0) = a
   NextElement (0,y) = a,        if y ≠ 0
   NextElement (x,0) = a,        if x ≠ 0
   NextElement (x,y) = f (NewMatrix sub (South (x,y)))
                         (NewMatrix sub (West (x,y))),    if x ≥ 1 ∧ y ≥ 1
```

where f depends on the equations being solved. South and West calculate neighbours' coordinates from the present coordinate:

```
South (x,y) = (x,y−1)
West (x,y) = (x−1,y)
```

This example is interesting because it is very close to the original mathematics, and is highly parallel. Computation can proceed in a "wavefront", which marches diagonally across the matrix. Coding the algorithm in an imperative language is rather awkward because the matrix must be scanned in the right order to ensure that values are defined before they are used. Writing an imperative parallel version is harder still. This and other examples are the subject of an excellent article on declarative scientific programming by Arvind and Ekanadham [AE88].

Function composition

Functions are obviously important in functional programming—but what can one do with a function? A fundamental operation on functions is to *compose* them, to form another function which applies first one function, and then the other. It should be possible to deduce precisely what compose must do from its type specification:

$$\mathsf{compose} :: (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

Its definition is

```
compose f g = h
              where
              h x = f (g x)
```

Though more explanatory, this is precisely equivalent to

```
compose f g x = f (g x)
```

The infix form of compose is written "∘":

```
(f ∘ g) x = f (g x)
```

Composition is clearly associative:

```
(f ∘ g) ∘ h = f ∘ (g ∘ h)
```

This is easily shown by providing the missing parameter x, and then reducing. The LHS is

$$\begin{aligned}\underbrace{((\mathsf{f} \circ \mathsf{g}) \circ \mathsf{h})\ \mathsf{x}} &= \underbrace{(\mathsf{f} \circ \mathsf{g})\ (\mathsf{h}\ \mathsf{x})} \\ &= \mathsf{f}\ (\mathsf{g}\ (\mathsf{h}\ \mathsf{x}))\end{aligned}$$

and the RHS is

$$\begin{aligned}\underbrace{(\mathsf{f} \circ (\mathsf{g} \circ \mathsf{h}))\ \mathsf{x}} &= \mathsf{f}\ \underbrace{((\mathsf{g} \circ \mathsf{h})\ \mathsf{x})} \\ &= \mathsf{f}\ (\mathsf{g}\ (\mathsf{h}\ \mathsf{x}))\end{aligned}$$

as expected.

The purpose of "∘" is to allow us to build functional objects without having to introduce parameters explicitly. This is taken one step further by the next example.

Combinators and combinator abstraction

The function "∘" passes only one parameter at a time. It is sometimes useful to pass more than one, and this requires a generalisation of function composition:

$$\circ\circ :: (\beta_1 \to \beta_2 \to \gamma) \to (\alpha \to \beta_1) \to (\alpha \to \beta_2) \to \alpha \to \gamma$$

```
f ∘∘ g1 g2 = h
             where
             h x = f (g1 x) (g2 x)
```

An interesting early result in the theory of functional programs is that these functions allow us to express arbitrary functions without referring to variables at all. For example, the function f defined by

```
f x = (log x) / ((sqrt x) − (2 × x))
```

is equivalent to

```
f = (/) ∘∘ log
          ((−) ∘∘ sqrt      )
                  ((×) 2)
```

(we assume for convenience here that "∘∘" binds more tightly than application). To see that this is so, let us provide the missing parameter and apply reduction:

```
f x = ( (/) ∘∘ log
            ((−) ∘∘ sqrt
                    ((×) 2) )) x

    = (/) (log x)
          ( ((−) ∘∘ sqrt
                    ((×) 2) x) )

    = (/) (log x)
          ((−) (sqrt x)
               (2 × x) )

    = (log x) / ((sqrt x) − (2 × x))
```

It is possible to find an algorithm, called a *combinator abstraction algorithm,* which systematically transforms any definition to remove variables, introducing operators like "∘∘" (called *combinators*) instead. This is frequently useful in program transformation, and, as we shall see in Chapter 3, it is a common compilation technique.

Classical works use a more fundamental set of combinators, called S K and I:

```
S a b c = a c (b c)
K a b = a
I a = a
```

It happens that

S a b c = apply $\circ\circ$ a b c
where
apply f x = f x

Field and Harrison [FH88] and Glaser, Hankin and Till [GHT84] both give good introductions.

2.3 Equational Reasoning

We have already seen some simple arguments about functions like hd, tl, map2 and ply. These employed the straightforward approach of using the equalities given in the program script to rewrite expressions. This *equational* way of reasoning about functional programs derives its basis from the reduction mechanism by which the meaning of an expression is calculated: it is really only controlled, symbolic evaluation of the program.

We examine the technique more closely by means of an example, which we draw from the rich algebra of equalities between functions like map, map2 and ply. We will prove that

for all op $\in \alpha \to \beta \to \gamma$,
as $\in [\alpha]$,
bs $\in [\beta]$:

map2 op as bs = ply (map op as) bs

We begin by defining a new function map2', a name for the form on the RHS:

map2' op as bs = ply (map op as) bs

Now *instantiate* this equation for the cases of the parameters as and bs, that is specialise the equation for particular forms of parameters. Begin with when both as and bs are empty:

map2' op [ ] [ ] = ply (map op [ ]) [ ]

Now apply reduction (sometimes called *unfolding* when used in program transformation) to the RHS. Use the definitions of map and ply:

$$\begin{aligned} \text{map2' op [ ] [ ]} &= \text{ply } \underbrace{\text{(map op [ ])}}\ \text{[ ]} \\ &= \underbrace{\text{ply [ ] [ ]}} \\ &= \text{[ ]} \end{aligned}$$

Next we take the case where both are lists of one or more elements:

```
map2' op (a : as) (b : bs) = ply (map op (a : as)) (b : bs)
```

Now we apply some reduction to the RHS, using the equations for map and ply:

```
map2' op (a : as) (b : bs) = ply (map op (a : as)) (b : bs)
                           = ply ((op a) : (map op as)) (b : bs)
                           = (op a b) : (ply (map op as) bs)
```

In the RHS of this equation is an instance of the RHS of the equation we used to define map2'. We can use that equation to rewrite the equation above to

```
map2' op (a : as) (b : bs) = (op a b) : (map2' op as bs)
```

This step, called *folding*, used an equation backwards, from RHS to LHS. As was noted earlier, computations which use such steps may not always terminate when they should. Thus, this transformation is not guaranteed to preserve termination properties correctly—an independent proof is needed, which would normally use the technique of induction, which is introduced in section 2.5. We do, of course, retain the guarantee that when the program does terminate it yields the expected answer.

The result of these instantiations and simplifications is a new pair of equations concerning map2':

```
map2' op [ ] [ ] = [ ]
map2' op (a : as) (b : bs) = (op a b) : (map2' op as bs)
```

This definition of map2' is identical in structure to the definition of map2, and we can therefore conclude that, indeed,

```
map2' = map2
```

that is,

```
map2 op as bs = ply (map op as) bs
```

Note that we used four kinds of step in this argument: *definition* (of function map2'), *instantiation* (of map2' for empty and non-empty lists parameters), *folding* and *unfolding*. Similar steps which will be used later include include *abstraction* (introduction of a **where** clause), *laws*, meaning the application of ready-proven equalities, and *cancellation*. Cancellation is simply the rule that if, for all parameters x,

```
f x = g x
```

then we can infer that f = g —in fact this is the definition of equality for functions. We will often use it in defining functions. For example, one might write

```
sum as = insert (+) 0 as
```

By cancellation this is equivalent to

```
sum = insert (+) 0
```

Together, these rules constitute a very powerful transformation technique and we shall use it extensively. It was pioneered in Darlington and Burstall [Dar82], where it is called the *fold/unfold* system. A great deal of work has been done on providing automated support to check and manage such derivations. Very powerful automatic techniques exist which can derive many useful results without human intervention. Furthermore, while the need to assure the preservation of correct termination behaviour remains in general, it is possible to show that many forms of derivation are completely valid despite the use of folding.

Because of the problem of assuring termination correctness, equational reasoning must be supplemented by induction techniques. In fact, it often turns out to be easier to perform a complete verification by induction rather than find a forward derivation. The next few sections develop a very simple basis for using inductive arguments of various kinds.

2.4 Partial functions and partial data structures

Several of the definitions given so far have been recursive: an object is defined in terms of itself. Whenever this occurs, the possibility exists that the object's value has not been properly defined. When one applies reduction to find the value of such an object, we may never reach the normal form. This section introduces the ideas necessary to frame questions about the termination of functional programs, which can become quite subtle when, for example, infinitely-long lists are considered.

A particularly useful way to address the problem is to introduce a special symbol, $\bot$ (called "bottom"), the archetypical non-terminating computation, which can be expressed in the functional language simply by the equation

$$\bot = \bot$$

We consider all non-terminating computations, and all computations with an undefined result, to be equal to $\bot$. This presumption is valid as long as we consider only the result of the computation (its *extensional* properties), and not the manner of its execution (its *intensional* properties).

Of course we cannot always tell whether a particular expression is equal to $\bot$. As an example, we might try to write a program to find whether my telephone number appears in the decimal expansion of e:

```
FindSubList MyPhoneNumber DigitsOfe
where
DigitsOfe = [2, 7, 1, 8, 2, 8, ...]
```

The rôle of $\perp$ is to provide the algebraic language to ask such questions.

2.4.1 Strictness

For example, one interesting question to ask of an N-parameter function f is whether, when we make the i'th parameter $\perp$, the result has to be $\perp$ too. More formally the question is whether

$$\mathsf{f}\ \mathsf{x}_1\ \ldots \mathsf{x}_{i-1}\ \perp\ \mathsf{x}_{i+1}\ \ldots \mathsf{x}_N = \perp$$

for all x_j, $j \neq i$. If so, f is said to be *strict* in its i'th parameter: *either*

- $\mathsf{f}\ \mathsf{x}_1\ \ldots \mathsf{x}_{i-1}\ \mathsf{x}_i\ \mathsf{x}_{i+1}\ \ldots \mathsf{x}_N = \perp$ always, *or*
- f must use its i'th parameter to produce its result.

If f is not strict in its i'th parameter, it cannot make use of its i'th parameter in forming its result.

The practical import of this is that if f is strict in parameter i, then when reducing an application of f to actual parameters e_1 to e_N,

$$\mathsf{f}\ \mathsf{e}_1\ \ldots \mathsf{e}_{i-1}\ \mathsf{e}_i\ \mathsf{e}_{i+1}\ \ldots \mathsf{e}_N$$

parameter e_i can be reduced before the application of f, or in parallel with it, while still retaining the guarantee that the normal form will be found if it exists.

Powerful techniques exist for strictness analysis based on the technique of abstract interpretation [HBJ88], and this offers the prospect of highly parallel reduction. See section 3.1.5 for more details.

2.4.2 Recursion

Another use of $\perp$ is to lend a mathematical meaning to recursive definitions, and to form a basis for the induction techniques we will introduce later in the chapter.

It is quite straightforward to give a mathematical semantics to non-recursive definitions, but a recursive definition has to be unravelled into an infinitely large expression before it loses its recursive nature. Using $\perp$, however, we can approximate to the semantics of a recursive definition as closely as necessary.

Firstly, let us define our notion of approximation (consider functions over numbers only): the function f approximates the function g (written $\mathsf{f} \sqsubseteq \mathsf{g}$) if and only if

$$\begin{array}{l}\mathsf{for\ all\ x}\\ \qquad \mathsf{f\ x} = \mathsf{g\ x} \vee \mathsf{f\ x} = \perp\end{array}$$

(where $\vee$ denotes logical "or"). For example, if we have the definitions

```
f x = ⊥        g 1 = a        h 1 = a
               g 3 = c        h 2 = b
                              h 3 = c
```

Functions g and h yield $\bot$ except where defined otherwise. Now

```
f ⊑ g ⊑ h
```

since h is consistent with g, but is defined for more parameter values. All functions are more defined than f, which is undefined for all parameters.

Now suppose we have a recursively-defined function r,

```
r x = ...r ...r ...
```

in which r appears one or more times on the RHS. Let us abstract r from the RHS:

```
r x = body r x
      where
      body r' x' = ...r' ...r' ...
```

The function body captures the contents of the "...", but allows us to manipulate the recursive call explicitly. The primed symbols r' and x' are new variables. Define

```
r₀ x = ⊥
```

and

```
r₁ x = body r₀ x
```

that is,

```
r₁ = body r₀
```

Clearly, $r_0 \sqsubseteq r_1$. The function r_1 is not much use: it is undefined on all but the simplest input values. However, we can extend it by iterating again:

```
r₂ = body r₁
```

and again,

```
r₃ = body r₂
```

Thus,

$$\mathsf{r}_i = \mathsf{body}^i$$
where
$$\mathsf{body}^i\ \mathsf{x} = \underbrace{\mathsf{body}\ (\ \mathsf{body}\ \cdots\ \mathsf{body}}_{i\ \mathit{times}}\ (\ \bot\ )\ \cdots\ )$$

We can generate a list of all these iterates:

$$[\mathsf{r}_0,\ \mathsf{r}_1,\ \mathsf{r}_2,\ \mathsf{r}_3\ \cdots\ ] = \mathsf{iterate\ body}\ \bot$$

For any given input x there is some integer n such that

$$\mathsf{r}_n\ \mathsf{x} = \mathsf{r}\ \mathsf{x}$$

Notice that the iterations form an increasing chain, called the *Kleene chain*:

$$\mathsf{r}_0 \sqsubseteq \mathsf{r}_1 \sqsubseteq \mathsf{r}_2 \sqsubseteq \cdots \sqsubseteq \mathsf{r}_i \cdots$$

The meaning of r itself is just the limit of this chain as i tends to infinity:

$$\mathsf{r} = \lim_{i\to\infty} \mathsf{r}_i$$

because then the equation

$$\mathsf{r}_i = \mathsf{body}\ \mathsf{r}_{i-1}$$

will actually hold. This limit is, therefore, the solution of the equation we used to define r in the first place.

Technically, r is called the *least fixed point* of body, because r is the least-defined function which is unchanged by the transformation body. For a more formal treatment of this material the reader is referred to [Sch86].

2.4.3 Partial data structures

A partial list is a finitely-long list which ends with $\bot$ instead of [ ]. For example,

$$1 : \bot$$
$$1 : 2 : \bot$$
$$1 : 2 : 3 : \bot$$
$$1 : 2 : 3 : 4 : \bot$$

are partial lists of numbers. When infinite lists were first introduced earlier in this chapter, attention was drawn to the analogy with a communications channel, on which values are transmitted periodically—or even sporadically—but indefinitely. A partial list represents a channel on which a few elements are sent, but then is silent forever. It is impossible to distinguish a partial list from a longer one simply by computing its value, because one cannot tell when to give up waiting for the next value to appear.

Just as with functions, there is a useful notion of approximation for partial lists: $\mathsf{l}_1 \sqsubseteq$

l_2 if and only if

$$\mathsf{l}_1 = \bot \vee (\mathsf{hd}\ \mathsf{l}_1 \sqsubseteq \mathsf{hd}\ \mathsf{l}_2 \wedge \mathsf{tl}\ \mathsf{l}_1 \sqsubseteq \mathsf{tl}\ \mathsf{l}_2)$$

This is sometimes called the prefix ordering because $\mathsf{l}_1 \sqsubseteq \mathsf{l}_2$ if and only if l_1 is an initial prefix of l_2 and ends in $\bot$ (or is actually equal to l_2). Under this ordering, it should be clear that

$$\bot \quad \sqsubseteq \quad 1 : \bot \quad \sqsubseteq \quad 1 : 2 : \bot \quad \sqsubseteq \quad 1 : 2 : 3 : \bot$$

Just as with the recursive function r, we can give a meaning to a recursively-defined list l,

$$\mathsf{l} = \ldots \mathsf{l} \ldots \mathsf{l} \ldots$$

by considering successive iterates starting from $\bot$. This time, let us take a concrete example:

$$\mathsf{fibs} = 1 : 1 : (\mathsf{map2}\ (+)\ \mathsf{fibs}\ (\mathsf{tl}\ \mathsf{fibs}))$$

We abstract out the recursive reference to fibs:

$$\begin{array}{l}\mathsf{fibs} = \mathsf{body}\ \mathsf{fibs} \\ \quad\ \textbf{where} \\ \quad\ \mathsf{body}\ \mathsf{fibs'} = 1 : 1 : (\mathsf{map2}\ (+)\ \mathsf{fibs'}\ (\mathsf{tl}\ \mathsf{fibs'}))\end{array}$$

Now we can enumerate the first few iterates, and use reduction to find their values:

$$\mathsf{fibs}_0 = \bot$$

$$\mathsf{fibs}_1 = \mathsf{body}\ \mathsf{fibs}_0 = 1 : 1 : \underbrace{(\mathsf{map2}\ (+)\ \mathsf{fibs}_0\ (\mathsf{tl}\ \mathsf{fibs}_0))} = 1 : 1 : \bot$$

$$\mathsf{fibs}_2 = \mathsf{body}\ \mathsf{fibs}_1 = 1 : 1 : \underbrace{(\mathsf{map2}\ (+)\ \mathsf{fibs}_1\ (\mathsf{tl}\ \mathsf{fibs}_1))} = 1 : 1 : 2 : \bot$$

$$\mathsf{fibs}_3 = \mathsf{body}\ \mathsf{fibs}_2 = 1 : 1 : \underbrace{(\mathsf{map2}\ (+)\ \mathsf{fibs}_2\ (\mathsf{tl}\ \mathsf{fibs}_2))} = 1 : 1 : 2 : 3 : \bot$$

$$\mathsf{fibs}_4 = \mathsf{body}\ \mathsf{fibs}_3 = 1 : 1 : \underbrace{(\mathsf{map2}\ (+)\ \mathsf{fibs}_3\ (\mathsf{tl}\ \mathsf{fibs}_3))} = 1 : 1 : 2 : 3 : 5 : \bot$$

$$\mathsf{fibs}_5 = \mathsf{body}\ \mathsf{fibs}_4 = 1 : 1 : \underbrace{(\mathsf{map2}\ (+)\ \mathsf{fibs}_4\ (\mathsf{tl}\ \mathsf{fibs}_4))} = 1 : 1 : 2 : 3 : 5 : 8 : \bot$$

and so on. As before, we can easily generate a list of all these iterates:

$$[\mathsf{fibs}_0, \mathsf{fibs}_1, \mathsf{fibs}_2, \ldots] = \mathsf{iterate}\ \mathsf{body}\ \bot$$

and the limit of this series,

$$\mathsf{fibs} = \lim_{i \to \infty} \mathsf{fibs}_i$$

satisfies the original recursive equation used to define fibs.

The value of an element of this series, fibs_i, say, denotes the result of an unfinished computation. Thus we can think of the $\bot$ which appears in, for example,

$$\mathsf{fibs}_5 = 1 : 1 : 2 : 3 : 5 : 8 : \bot$$

as meaning "not yet" instead of "never". This should provide further support for the analogy between lists and communications channels.

2.5 Induction

In this section the most powerful technique for reasoning about functional programs is presented. Proof by induction over the natural numbers should be familiar from school mathematics. We will introduce some slight variations, all ultimately reducible via *computational induction* (below) to induction over natural numbers.

2.5.1 Computational induction

The most fundamental form of inductive argument about a functional program's behaviour is based on the number of iterations in the Kleene chain of approximations to a recursively-defined value. Suppose we have some recursive definition:

$$\mathsf{x} = \ldots \mathsf{x} \ldots \mathsf{x} \ldots$$

Using the ideas from the previous section, we have

$$\mathsf{x} = \lim_{i \to \infty} \{ \mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \ldots \}$$

and we need to show that some property P holds for x. It is easy to show $\mathsf{P}\ \mathsf{x}_i$ for all i:

Base case: show that $\mathsf{P}\ \bot$.

Inductive step: show that, given $\mathsf{P}\ \mathsf{x}_i$, $\mathsf{P}\ \mathsf{x}_{i+1}$ holds.

This establishes $\mathsf{P}\ \mathsf{x}_i$ for all i, but does not automatically imply that P holds for the limit, x, which is what actually interests us. Fortunately it is valid for a very large class of "admissible" predicates.

2.5.2 Admissible predicates

A predicate P is admissible if it is *chain complete*: P is defined to be chain complete if when P holds for every element of a Kleene chain it holds for its limit. That is,

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$$

and

$$P\ x_0 \wedge P\ x_1 \wedge P\ x_2 \wedge \cdots$$

implies

$$P\ (\ \lim_{i\to\infty}\ \{\ x_0, x_1, x_2, \ldots\}\ )$$

We will find that all the program properties we are interested in are admissible, because the assertion that any two expressions are equal is chain complete. An example of a non-chain-complete predicate is a test whether a list is partial.

A more mathematical treatment of this material including the characterisation of a class of admissible predicates is to be found in [MNV73].

2.5.3 Partial structural induction

This is by far the most commonly-used induction method. Just as computational induction applies when recursion is used in a function or value's definition, structural induction deals with recursion in data types, as, for example is found in the definition of lists given earlier:

$$\mathsf{List}\ \alpha ::= \mathsf{NIL} \mid \mathsf{CONS}\ \alpha\ (\mathsf{List}\ \alpha)$$

More generally, such definitions define tree-like structures, for example:

$$\mathsf{Tree}\ \alpha ::= \mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)$$

Just as with data value recursion, we can unravel this data type definition into its Kleene chain, starting with the type containing only undefined elements, which we will call $\{\bot\}$:

$$\mathsf{Tree}_0\ \alpha ::= \{\bot\}$$

$$\begin{aligned}\mathsf{Tree}_1\ \alpha &::= \mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ (\mathsf{Tree}_1\ \alpha)\ (\mathsf{Tree}_0\ \alpha)\\ &::= \mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ \{\bot\}\{\bot\}\end{aligned}$$

$$\begin{aligned}\mathsf{Tree}_2\ \alpha &::= \mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ (\mathsf{Tree}_1\ \alpha)\ (\mathsf{Tree}_1\ \alpha)\\ &::= \mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ (\mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ \{\bot\}\{\bot\})\\ &\qquad\qquad\qquad\quad (\mathsf{LEAF}\ \alpha \mid \mathsf{NODE}\ \{\bot\}\{\bot\})\\ &\quad\ \vdots\end{aligned}$$

This suggests an induction schema for showing that $P\ x$ for all $x \in \mathsf{Tree}\ \alpha$:

Base case: show that $P\ \bot$.

Inductive step: Given that

for all x $\in$ Tree$_i$ α:
    P x

show that

for all x $\in$ Tree$_{i+1}$ α:
    P x

Provided P is admissible, this schema proves P for any choice of α in Tree α: it subsumes the proofs for Tree [Char], Tree (Tree Num) etc.

We can improve the schema above substantially by observing that the inductive step is always proved for each case of the data type separately, and that the non-recursive cases (such as NIL and LEAF α) are more properly moved into the base-case since they do not require the inductive assertion. The simplified schema for partial structural induction on Trees is as follows:

Base cases: 1. Show that P $\perp$.

2. Show that

    for all x $\in$ α
        P (LEAF x)

Inductive step: Given that P t_1 and P t_2, show that

P (NODE t_1 t_2)

The partial induction schema for lists is

Base cases: 1. Show that P $\perp$.

2. Show that P [ ].

Inductive step: Given that P xs, show that

for all x $\in$ α:
    P (x : xs)

Notice that just as with computational induction, we require that P be admissible to infer from such a proof that P holds for an infinitely-large structure.

2.5.4 Total structural induction

There are many useful properties which hold for all finite, total (i.e. not partial) elements of a data type. An example is

for all as, bs $\in [\alpha]$, FiniteAndTotal as:
    reverse (as ++ bs) = (reverse bs) ++ (reverse as)
where
reverse :: $[\alpha] \to [\alpha]$

reverse [ ] = [ ]
reverse (x : xs) = (reverse xs) ++ [x]

Recall that "++" is the infix form of append, the function which joins lists. We assume the archetypal non-admissible predicate FiniteAndTotal, which holds for only those finite lists ending in [ ]. The property obviously fails for infinite and partial lists:

reverse ([1,2,3, $\cdots$] ++ bs) = $\bot \neq$ (reverse bs) ++ (reverse [1,2,3, $\cdots$])

A partial structural induction proof of this property fails in its base case, quite reasonably, because

reverse ($\bot$ ++ bs) $\neq$ (reverse bs) ++ (reverse $\bot$)

For such proofs, the total structural induction schema is useful. Here is the version for $[\alpha]$:

Base case: Show that P [ ].

Inductive step: Given that P xs, show that

for all x $\in \alpha$:
    P (x : xs)

This establishes P for all finite and total elements of $[\alpha]$. For total structural induction, we drop the $\bot$ part from the base case, and no longer require that P be admissible.

A particularly common use of total structural induction is over the natural numbers. These can be defined as a data type:

Nat ::= ZERO | SUCC Nat

For example the natural number 3 would be written

SUCC (SUCC (SUCC ZERO))

However, this data type includes such elements as

$$(\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}(\ \cdots\ ))\ \cdots\ )) = \infty$$

and

$$\mathsf{SUCC}\ (\mathsf{SUCC}\ (\mathsf{SUCC}\ \bot))$$

which might be thought of as "at least 3". Clearly, for most purposes we mean to deal with the finite, total natural numbers. Their total induction schema is

Base case: Show $\mathsf{P}\ \mathsf{ZERO}$.

Inductive step: Given $\mathsf{P}\ \mathsf{n}$, show that $\mathsf{P}\ (\mathsf{SUCC}\ \mathsf{n})$.

It is often fruitful to think of the finite and total elements of a data type as a distinct subtype, and we might use some notation to that effect in type specifications, as in

$$\mathsf{reverse} :: [\alpha]_! \to [\alpha]_!$$

where the "!" suffix indicates that the function requires a finite, total list to produce a result, and that the result it produces is finite and total.

In general, it is not strictly necessary to base a total structural induction argument on a data type. Any structure will suffice, provided a *well-founded ordering* can be imposed on it. A well-founded ordering is an ordering relation, say $\preceq$, on a set, say A, such that the set contains no infinitely long decreasing chain of elements

$$\cdots \mathsf{a}_4 \prec \mathsf{a}_3 \prec \mathsf{a}_2 \prec \mathsf{a}_1 \prec \mathsf{a}_0$$

The ordering may be a partial one: for some elements a and b, it may be that neither $a \preceq b$ or $b \preceq a$ holds. Using it we can state the general version of the structural induction principle:

To prove that: $\mathsf{P}\ \mathsf{x}$ for all $\mathsf{x} \in A$ where $\preceq$ is a well-founded ordering on A,

Base case: Show that $\mathsf{P}\ \mathsf{x}$ for all *minimal* elements x of A, that is for all $\mathsf{x} \in A$ such that there is no $\mathsf{x}' \prec \mathsf{x}$.

Inductive step: Given that $\mathsf{P}\ \mathsf{x}'$ for all $\mathsf{x}' \preceq \mathsf{x}$, show that $\mathsf{P}\ \mathsf{x}$.

Note that we can in fact assume P for *all* values smaller than x; this extends the induction schemata given so far.

2.5.5 Recursion induction

This final technique is arguably not an induction argument at all. In using the limit of the Kleene chain to give a meaning to a recursively-defined variable, we argued that if r is defined by the recursive equation

```
r = body r
    where
    body r' = ...r'...r'...
```

and

$$[r_0, r_1, r_2 \cdots] = \text{iterate body } \bot$$

then

$$r = \lim_{i \to \infty} r_i$$

is a solution to the recursive definition of r because at the limit,

$$r_i = r_{i-1}$$

However, we may be able to find a solution by other means. For example, let us define the function triangle over the natural numbers:

```
triangle :: Num → Num

triangle n = 1,                          if n = 1
triangle n = n + (triangle (n−1)),       if n > 1
```

We can use the Kleene chain to find the result of applying triangle to any particular parameter by unravelling far enough. However, there is a non-recursive solution to these equations:

```
triangle2 n = n×(n+1)/2
```

To verify that this is so, we can see whether the equations defining triangle are indeed satisfied when we substitute triangle2 for triangle. Is it true that

```
triangle2 n = 1,                          if n = 1
triangle2 n = n + (triangle2 (n−1)),      if n > 1   ?
```

Unfolding triangle2 throughout gives:

```
n×(n+1)/2 = 1,                            if n = 1
```

which is trivially satisfied, and

```
n×(n+1)/2 = n + ((n−1)×((n−1)+1))/2,                    if n > 1
```

We use arithmetic laws to simplify this:

$$
\begin{aligned}
n\times(n+1)/2 &= n + ((n-1)\times n)/2 \\
&= (2\times n + ((n-1)\times n))/2 \\
&= (2\times n + n^2 - n)/2 \\
&= (n + n^2)/2 \\
&= n\times(n+1)/2
\end{aligned}
$$

This completes the proof that triangle2 satisfies the equations defining triangle.

This means that when triangle x is defined, triangle2 x must also be defined, and must give the same result. This is not quite the same as proving that for all x, triangle x = triangle2 x, because triangle2 x may be defined when triangle x is not. This happens in our example when $x < 1$.

To be more precise, what recursion induction verifies is that

for all $x \in$ Num
    triangle x $\sqsubseteq$ triangle2 x

This is just the definition of $\sqsubseteq$ on functions:

triangle $\sqsubseteq$ triangle2

If triangle had been defined for all of its domain type, we could immediately infer that triangle2 = triangle. Indeed, if we modify triangle's type specification to

triangle :: Nat $\rightarrow$ Nat

so that triangle is a total function, then, over this restricted domain type, triangle2 must be equal to it.

The recursion induction principle

From this example we derive the following proof schema: given a system of recursive equations defining f, and a value f' which satisfies these equations, infer that

f $\sqsubseteq$ f'

Commonly, the proof of actual equality is unnecessary.

Note that a simple form of recursion induction was used earlier (page 27) to assert that because map2' and map2 are defined by equations of the same structure, they are the same.

2.6 Why Functional Languages?

Having introduced functional programming and given a very brief guide to transformation and verification within the functional style, we conclude this chapter with a review of the success and generality of the approach.

Functional languages are very often defined by default: they *lack* assignment. They are of interest here for positive, rather than for negative reasons. The prohibition of assignment does not prevent the construction of evolving data structures, but the interdependencies between operations, which arise when changes to data must be propagated between operations, must be made explicit. It is not proven that prohibiting assignment always simplifies programming, although the examples where the functional approach fails lead one to conclude that a neater solution than simply re-introducing assignment is possible (see [ASS85] pg. 290 for a discussion).

Foregoing explicit assignment removes a major source of programming errors by letting the machine arrange re-use of memory cells. Assignment is a means of informing the computer that a value is no longer required, and that the cell holding it is to be re-used for holding the new value given. In a functional language, the programmer is relieved of any concern for the lifetime of values, and the timing of destructive overwriting of the cells containing them. This is an important abstraction from the housekeeping needed with von Neumann programming. It is also a big step towards avoiding unnecessary concealment of a program's parallelism, since programmed memory re-use introduces spurious points of synchronisation between computations which might otherwise proceed in parallel.

2.6.1 Referential Transparency

The first positive reason for considering functional languages is that functional programs are easier to reason about than imperative ones. This should have become clear from the examples in this chapter.

In a functional program, any pair of expressions which are syntactically the same are semantically the same, scope rules allowing. This property, "referential transparency", is a corollary of the prohibition of assignment—without assignment, expression evaluation can have no side-effects, so different evaluations of the same expression must yield the same result. Referential transparency means that a program's script can be treated as a system of equations, and the equational form of reasoning familiar from algebra is applicable.

2.6.2 Higher-Order Functions

Another benefit of the functional approach is the ease and cleanliness with which higher order functions can be defined. Generally, no special syntax is needed to define or manipulate functions whose parameters or values are also functions. Moreover, algebraic properties such as referential transparency still hold; there is no context sensitivity problem for higher-order functions, as there is with dynamically-scoped languages such as (some dialects of) Lisp.

2.6.3 Polymorphic Type Checking

Strong typing in traditional languages destroys the usefulness of higher-order functions by insisting that a separate definition be written for each different parameter type combination, even if the function need not know its parameters' types. For example, the context of "map" above implies that its type is

$$\mathsf{map} :: (\mathsf{line} \to \mathsf{line}) \to ([\mathsf{line}] \to [\mathsf{line}])$$

This type would conflict with many other likely uses of "map", resulting in a type error under a strong type discipline like Pascal's [JW75].

Polymorphic type checking enables a single, generic function to be written instead. The generic function is assigned a type expression detailing the minimum structure required for type consistency:

$$\mathsf{map} :: (\alpha \to \beta) \to ([\alpha] \to [\beta])$$

If the usage of the function is then consistent with the function's type expression, run-time type errors can be guaranteed never to occur.

An object's contextually-implied type is consistent with its generic type expression if the implied type can be obtained from the generic type expression by a consistent substitution of type variables by sub-expressions of the contextual type.

Polymorphic type inference and checking are not confined to functional languages, but reflect the importance of higher-order functions to the expressive power of functional languages.

2.6.4 Declarative Completeness

Viewing a functional program as a system of recursive equations leads to a declarative reading, where the meaning of the program is taken to be the mathematical solution of the equation system. Our language has a very valuable *completeness* property: the reduction of an expression by rewriting terms according to the program's equations, using the general normalisation strategy, is guaranteed to yield a result if a result is mathematically deduceable. This leads to a view, taken in [HOS85], where a suitable functional language is regarded as a logic programming language based on equations (in contrast to relations, as in Prolog). The equational nature, combined with the completeness property, suggests that functional languages have a fundamental importance.

2.7 Why Not Functional Languages?

There are problems with functional languages. A selection are listed here.

2.7.1 Lack of Expressive Power.

Serious problems have been encountered with extensions of functional languages to express interactive resource management. Finding language constructs to express the kind of non-determinism needed is quite easy, and several examples have been implemented and used successfully, such as the "merge" operator of [AS85] and [Jon84], and the resource managers of [AB84]. The difficulty is rather with maintaining the language's desirable algebraic properties.

In the context of parallel programming, the effect of this restriction is that a functional program's result cannot depend on the order or speed of evaluation of its constituent

expressions. This is a very attractive safety feature for parallel programming, but does limit the application to some extent.

2.7.2 Lack of Abstractive Power.

Allowing program objects to have evolving local state can substantially simplify the expression of certain algorithms—the functional style prohibits an "object-oriented" programming structure. [ASS85] argues that, although assignment can always be avoided (e.g. using lazy streams), their constraint-propagation circuit simulator, for example, would be inordinately complicated if no assignment were allowed at all. An obscure program in a mathematically-simple language is, surely, at least as difficult to reason about as a clear program in a language with a more complex logic.

An alternative to simply re-introducing assignment is to notice that the language features proposed to help with the problem of interactive resource management mentioned above, such as Abramsky and Sykes' non-deterministic merge, or Arvind and Brock's resource managers, can also be used to help simplify the expression of shared access to a state variable. This may be a more structured and uniform approach than allowing assignment, but the semantic problems remain.

2.7.3 Performance

Until recently, the usefulness of functional languages has been hampered by a lack of fast implementations on conventional computers. Their use as a basis for research into programming for high-performance, parallel computers therefore needed some justification.

Functional programs tend to run slowly because the language provides useful services to the programmer. These generally include

1. Dynamic store allocation.
2. Lazy semantics.
3. Higher-order functions.

It has been shown (see [Jon87], [Aug84], [BGS82]) that these features can often be removed from the compiled code, after careful program analysis. We review such techniques in Chapter 3. When such modern compiler technology is applied, performance on conventional machines can be very nearly comparable to standard imperative language implementations.[3]

2.7.4 The update problem

There remains a systematic performance problem which is harder in general to resolve. The histogram function given in section 2.2.6 was introduced as an illustration. At its heart lies the function MapElement f i xs, which was defined so that

[3]Languages having call-by-value semantics, such as Common Lisp [Ste84], and Hope [BMS80] certainly have implementations with performance competitive with the standard imperative language compilers (see [BGS82]). Compilers for lazy languages, such as Lazy ML [Aug84], are not far behind.

$$\mathsf{MapElement\ f\ i\ [x_1,\ x_2,\ \ldots x_i,\ \ldots x_n]} \\ = \mathsf{[x_1,\ x_2,\ \ldots f\ x_i,\ \ldots x_n]}$$

It is not possible to define this function within the language in a way which avoids some copying of the list. The reason is that references to the unchanged list must retain the same meaning. We cannot, in general, just replace the changed element *in situ*. Such an update might be called "destructive", since it overwrites a value which is already defined.

What this means is that we cannot simulate a conventional, imperative programming language (where assignment can be destructive) with the same efficiency. By using a tree representation, it can be shown that the loss need only be a factor proportional to the logarithm of the data structure's size, but the overheads of such schemes compared with the imperative approach are inevitably large.

Of course, in particular cases a compiler can locate where a destructive implementation of functions like MapElement can be used. This is rather complicated, and not very predictable; for example, it depends on evaluation order, which in turn depends on strictness analysis. The space usage characteristics of functional programs are notoriously difficult to predetermine.

2.8 Summary

This chapter has given a swift introduction to functional programming in the style used in the remainder of this book. A particular emphasis has been laid on techniques for manipulating and verifying functional programs, as a foundation for the more extensive derivations which follow. One aim of the approach to programming being advocated is that programmers will make use of simple identities when constructing and maintaining programs, and for this reason the presentation has not separated the activity of writing programs from that of reasoning about them.

2.9 Pointers into the literature

Standard texts on functional programming

Bird and Wadler's textbook [BW88] is the most appropriate source for material which expands on the content of this chapter at a similar level. Their notation is very similar. Field and Harrison [FH88] give a much deeper treatment of a wide range of subjects in the area, and is recommended for the reader wishing to go beyond the introductory level of this chapter.

Much of the material in Henderson's book [Hen80] is covered by these later books, but it is at least worth referring to for its investigations of stream programming and backtracking. Glaser, Hankin and Till wrote a fundamental textbook [GHT84] covering everything needed for a basic grounding in the area at the time. Their treatment of mathematical foundations such as combinators and the λ-calculus is particularly worth referring to. It lacks coverage on topics which have since gained importance, such as type systems. Another useful, but again somewhat dated, reference is the collection (commonly

called the "blue book") edited by Darlington, Henderson and Turner [DHT82]. This is interesting because of the breadth of the functional programming research community it spans.

Abelson and Sussman's textbook [ASS85] is outstanding in many respects. They are especially successful in placing the functional paradigm in context, carefully developing a discussion of whether a pure functional language is sufficiently expressive. Their book gives a taste for the work of the large Lisp-based community, ostensibly based on a functional view but extending far outside it. For an introduction closer to the mainstream of Lisp culture, one might look to Wilensky [Wil84].

Foundations for reasoning about functional programs

The λ-calculus, originated by Church [Chu41], was introduced as a notation for functions in general, and deals with higher-order functions particularly tidily. Its correspondence with most functional programming languages is close enough for us to think of their syntax as "sugar" for what is really just programming directly in the λ-calculus. Dana Scott founded a large body of theoretical computer science by constructing a model of the λ-calculus, using only fundamental mathematical notions such as set theory. This enables facts about the λ-calculus to be proven using classical mathematics, and forms the formal basis, called domain theory, for reasoning about functional programs as mathematical objects. Strachey applied this to the problem of giving mathematical meaning to other programming languages as a means of formal specification of programming language meaning. This area, denotational semantics, is well covered by [Sch86] and [Sto77]. The λ-calculus itself is rather thoroughly covered by Barendregt [Bar84]. Stoy gives a particularly accessible introduction to Scott's domain theory in [DHT82].

The principles of reasoning about programs using the techniques presented here were actually developed before that mathematics were formalised, in the first recorded instance by McCarthy [McC67]. The survey of induction techniques given here was based on a article by Manna, Ness and Vuillemin [MNV73], where a variety (more than were given here) are illustrated and verified with respect to computational induction, and ultimately, therefore, to Scott's domain theory.

Backus [Bac78] presents a rather different approach based on the combinator language FP. Whereas in our language higher-order functions can be constructed at will, FP is restricted to small set of well-understood higher-order combining forms, and the language is characterised by a quite small set of generally-applicable equivalences which constitute an "algebra of programs". Being a combinator language, there are no variables at all, only functions to pick out and manipulate parameters. This avoids consideration of parameter values in deciding the applicability of algebraic laws, and leads to a claim that reasoning occurs at a "function level". The reader is referred to Field and Harrison [FH88].

The presentation in this book avoids the λ-calculus, drawing instead from the theory of term rewriting systems. This allows a simpler explanation of equational reasoning and pattern matching. It is slightly more general, since it includes some functions (such as the non-strict or of section 3.1.3) which cannot be written in the λ-calculus (although they are represented in most models of the λ-calculus). The reader is referred to Klop [Klo90] or Huet and Oppen [HO80].

Assessing functional programming

It is for the reader to ponder the question of whether functional programming has anything to offer programming practitioners, and if so what. As already mentioned, this is one theme of Abelson and Sussman's book [ASS85]. They identify some serious problems in expressing certain program structures in the functional style, but they do not deal with the question of reasoning about programs. Hughes' article "Why functional programming matters" [Hug84] finds much expressive power in the functional style, and some of this presentation has been drawn from it. His emphasis is on using streams and function composition to separate programs into modules. Note that these conclusions are not contradictory, but rather indicate that the functional style does matter, but is not a universal panacea.

Arvind and Ekanadham advocate their language, *Id Nouveau*, for scientific programming in [AE88]. They find much to commend a purely functional approach. The vector and matrix recurrence notation used here is drawn from their "I-structures", and shares the interesting advantage seen in the Gauss-Seidel example (see section 2.2.8), that the matrix's scan order need not be specified. They do find cause to augment their language with features which are not purely functional, but are still relatively pure (see section 4.11). They have not yet found cause to deal especially with the "update problem" outlined in section 2.7.4.

Determinacy and operating systems

For parallel programming, functional languages have an outstanding advantage: regardless of the parallelism used in the evaluation, the result will be completely repeatable. It is not possible, either by accident or by design, to write a functional program whose result depends on who wins a "race" between two parallel processes.

Although very handy for many applications, there are situations where some kind of race is just what is wanted. For example, one might have a parallel algorithm in which any solution will do, but we do not know which "solver" process will find a solution first (McBurney and Sleep give an example of this, where a global bounding value is non-deterministically updated to control pruning in a parallel searching algorithm).

More common examples occur in operating systems and when dealing with input and output devices. It is in the nature of an operating system that its behaviour depends on the termination order of the processes for which it is responsible.

A substantial amount of work has been done on operating system design in the functional style. Abramsky and Sykes [AS85] and Simon Jones [Jon84] have built operating systems by introducing a special operator, merge. The stream returned by merge e_1 e_2 contains the elements of the streams e_1 and e_2 in the order in which they are computed. This "fair, bottom-avoiding" merge operator is sufficient to encode all the operating systems applications studied, and is not difficult to implement. Unfortunately, programs using merge are very difficult to reason about.

Input-output and the "plumbing problem"

In a functional language, all the objects upon which an expression's value might depend must be manifest in the expression itself. When programming input and output this raises the "plumbing problem": every expression, function or module which might perform input or output must be "plumbed in" to the input and output controllers, at the very top level of the program. Thus an apparently quite small modification, making a function print some status information for example, can involve substantial changes at all levels of abstraction. The only published attempt known by this author to deal with this problem appears in the FL language design, and is studied by Williams and Wimmers in [WW88].

Extensions to our functional language's type system

The language presented here uses the Hindley-Milner system [DM82], with no frills. It lacks overloading, for example of integers and reals, and could be extended relatively easily to include subtypes and inheritance, giving an immense boost in its power to describe complicated logical structures clearly and concisely.

A simple example of a subtype structure occurs with records, where a record is a collection of objects structured into named fields. A subtype of a record is a larger record, containing all of the first record's fields, and more. A function f defined to take an object of type A as a parameter is automatically defined on objects of subtypes of A. We say that f is inherited by the subtypes of A.

Cardelli and Wegner review of types in programming languages [CW85] is required reading. Kaes has proposed an attractive approach to introducing overloading to a language like ours [Kae88], and this is likely to be incorporated in the *Haskell* language design [HWA+88]. Fuh and Mishra [FM88] present the basis for a type scheme which retains the polymorphism and type inference properties of the Hindley-Milner system, but incorporates subtypes and inheritance.

Our language has no modules, as would be required for writing large programs. Modules can be parameterised by types, and can package structures to reflect their mathematical structure. Standard ML [Mil84] is an example, while Cardelli and Wegner, and Burstall [Bur84a] develop the theory. See also the work of Goguen and the OBJ group [Gog88]. Goguen argues that higher-order functions are not needed for typical higher-order programming examples: parameterised modules do the job more simply, and facilitate the imposition of semantic constraints on parameters (e.g. that the operator be associative for insert).

Specification languages

The declarative completeness property of functional programs implies that an object cannot be specified in the language without a giving a program to construct it. It is often useful to be able to write down the behaviour expected of a program in a formal manner, before going into the detail needed to implement it. Much of this book is devoted to generating improved implementations from specifications given as simple implementations, but non-executable specification techniques are occasionally used informally. See, for example,

the breadth-first list-tree interconversion functions in Appendix A, section A.6.2, and the definition of the vector and matrix operations in Appendix B.

Several languages have been designed specifically for giving formal specifications of software systems. Examples include *Z* (a good, short introduction is [Suf82]), VDM [BJ82], OBJ [GT79] and LARCH [GH86c]. All these languages have an executable subset which is functional.

Another approach to extending the power of a functional language to specify is by augmenting it with the mechanisms of logic languages. Degroot and Lindstrom [DL86] give a comprehensive survey. An interesting attempt to capture an object-oriented style in a specification language is described by Goguen and Meseguer [GM86].

Chapter 3

Sequential and Parallel Implementation Techniques

Hopefully Chapter 2 gave the reader a feel for the power and simplicity of the functional approach to programming. This chapter deals with implementation techniques. The aims of this chapter are

- To demonstrate that functional programs can be compiled to achieve performance competitive with other programming paradigms on conventional, sequential computers.
- To explain how these fast sequential implementation techniques can be extended to tightly-coupled multiprocessors.
- To develop an understanding of how parallelism arises in the functional program's source code.
- To provide a framework for assessing the costs involved in attempting to exploit parallelism, so that they may be weighed against the possible benefits.

The treatment of implementation technology will not be very profound or detailed: the intention is to give just enough detail to understand the problems of writing good parallel functional programs.

The chapter deals with the graph reduction approach to functional language implementation, and for a deeper description see Peyton Jones' textbook [Jon87]. There are other approaches but the influence on the programmer's view of program behaviour is the same.

3.1 An Overview of Compilation

Compiling a functional language is not fundamentally different from compilation of conventional languages, but the opportunities for analysis and optimisation are more abundant. Moreover, the analysis and simplification often have quite tidy justifications with reference to the language's underlying theory.

The phases one would expect in a high-quality compiler for our language will include

1. Type checking,
2. Simplification,
3. Removal of pattern matching,
4. Variable abstraction—removal of parameterised **where** clauses.
5. Strictness analysis,
6. Boxing analysis,
7. Code generation

Sophisticated compilers will include other phases, such as storage class analysis to classify values according to whether register, global, stack or heap (in that order of preference) storage can be used. The next few sections give some explanation of each phase.

3.1.1 Type checking

This has two purposes: to detect and report programmer's mistakes as clearly as possible, and to annotate the program with information needed later. The language used in this book uses a straightforward polymorphic type scheme, essentially the Hindley-Milner type system [DM82]. This system allows an object to be assigned the most general type expression possible, under the condition that its code act identically on all instances of that type expression. It also has the advantage that type declarations can be inferred if they are not given, and that the type checking algorithm will fail if a run-time error could occur due to a type mismatch.

More realistic programming languages require a slightly richer type system than this, if only to handle coercion of integers to reals properly. As is shown by Fuh and Mishra, [FM88] and Kaes [Kae88] this need not be a substantial complication.

3.1.2 Simplification

There are many opportunities to apply the algebra of programs, and in particular the properties of well-known operators, to simplify the program which the programmer originally wrote. Examples include

Common subexpression elimination: This standard compiler technique is applicable without restriction in the functional world, because no function can have a side-effect. Some care does have to be applied to avoid increasing the amount of working memory a program may require.

Partial evaluation: When all parameters to a function are provided at compile-time the compiler can simply calculate the value. When some but not all parameters are provided, massive simplifications can still occur. In particular, common higher-order functions like map and insertleft can be specialised to their function parameter. This can improve strictness information, reduce function call overheads and allow better storage class optimisation.

Unfolding simple functions: This is called 'inlining' in the standard compiler literature. It often leads to much more substantial simplifications.

Data type transformations: Library functions like ListToVector and VectorToList are known to satisfy handy properties such as

ListToVector (VectorToList as) = as

Thus, for example, if operations like map and insertleft are defined over vectors by first translating to lists, this simplification is applicable as soon as the definitions are unfolded.

This is an area of very active research, and several transformations appearing in this book are candidates. For example, see the ++ optimisation in Appendix A, section A.1.1.

It may seem unwise to apply so much unfolding to programs. There is a danger that the space occupied may be too large, but the principle at work is quite reasonable: operators like insertright, map and so on are shorthand for what in an imperative language would appear as an explicitly-coded loop. Thus the "macro-expansion" implementation should be no worse than the imperative case.

3.1.3 Removal of pattern matching

In Chapter 2 section 2.2.5, a strategy was given for selecting which equation to apply to a redex. As described there, the normalisation strategy is very inefficient: it may involve many tests being performed more than once, and it incurs the overhead of spawning and then tidying up a number of parallel testing processes. We simplify the discussion here by insisting (along with almost all existing compilers and language designs), that the patterns be restricted in form so that the parallel, "racing" implementation is not necessary. Instead, a sequential pattern testing implementation can be used.

Normal-order reduction

For programs which use no pattern matching, a trivial sequential strategy is guaranteed to find the normal form if it exists:

- **The normal-order normalisation strategy:** The left-most, outermost reducible expression is reduced at each step.

Without pattern matching, each variable is defined by a single equation of the form

f x_1 x_2 ... x_N = RHS

for $N \geq 0$. There is no need for any pattern-testing processes as there is never more than a single candidate equation. Under the normal reduction order, none of the parameters are evaluated at all before the function is invoked.

It is helpful (although not formally necessary) to allow just one function to be defined by pattern-matching, a conditional with the rôle of the if...then...else construct of conventional languages:

```
Bool ::= TRUE | FALSE

cond :: Bool → α → α → α

cond TRUE a b = a
cond FALSE a b = b
```

Normal order is still a normalisation strategy even with the addition of cond. We can think of cond as being a built-in primitive.

Compiling pattern-matching

The purpose of the pattern-matching removal phase of the compiler is to translate a program with patterns into a program without, so that the normal reduction strategy above can be used instead of the general (parallel) normalisation strategy. To do this, the sequence of parameter testing must be explicitly coded using the cond operator.

As a very simple example, consider a variation of cond defined by the equations

```
guard :: α → α → Bool → α

guard a b TRUE = a
guard a b FALSE = b
```

This function is just the same as cond except that the condition is the leftmost parameter instead of the leftmost one. It is easily translated into a pattern-free definition using cond:

```
guard a b test = cond test a b
```

Now when an application of guard is applied to some parameters, this equation is used straight away, before evaluating any parameters. The next redex will be an application of cond, and this will require test to be evaluated. Thus, although the guard function does not work properly under the normal reduction order, we can code it in terms of cond to get the desired effect.

Non-sequential patterns

This compilation technique cannot work in general. Take for example

```
or :: Bool → Bool → Bool

or FALSE FALSE = FALSE
or TRUE  x     = TRUE
or x     TRUE  = TRUE
```

This is the well-known Boolean 'or' function, but unusually it is defined to yield TRUE if one of its inputs is TRUE, even if the other input is still undefined. Its full truth table is

```
or FALSE FALSE = FALSE
or FALSE TRUE  = TRUE
or TRUE  FALSE = TRUE
or TRUE  TRUE  = TRUE
or ⊥     TRUE  = TRUE
or TRUE  ⊥     = TRUE
```

The only correct implementation of this or spawns two parallel processes, one to test the second defining equation (by evaluating the first parameter), the other to test the third equation (by evaluating the second parameter).

3.1.4 Variable abstraction

The next phase is a form of combinator abstraction, as described in Chapter 2, section 2.2.8. The difference is that the combinators are not chosen from a fixed set, but are derived from the input program. The process is often called *λ-lifting*, or, if certain "laziness" constraints are satisfied, *supercombinator abstraction*.

The object of λ-lifting is to eliminate function definitions from **where** clauses. The problem definitions are those which introduce new parameters in the LHS. For example, g in

```
f x y = (some expression involving g)
        where
        g z = (some other expression)
```

The transformation is very straightforward if the RHS of g's equation makes no reference to variables local to f, such as x and y. We just move g out of the **where** clause, if necessary renaming g to g' to make sure that no name clash is introduced:

```
f x y = (some expression involving g)
g' z = (some other expression)
```

If g's RHS does refer to free variables such as x and/or y, the free variables must be passed explicitly as parameters to g' every time g' is called. This gives

```
f x y = (some expression involving g x y)
g' x y z = (some other expression)
```

Peyton Jones gives the full details, which are quite complicated because of recursive definitions and the need to ensure that the transformation introduces no recomputation. The supercombinator abstraction algorithm makes this guarantee for all expressions (called *full laziness*, while λ-lifting introduces no recomputation of named expressions, but may recompute some unnamed ones. Most compilers employ λ-lifting because it involves less run-time overhead.

The result of the variable abstraction phase is a set of *simple recursion equations*, with simple LHS's (thanks to the pattern matching removal), and *flat* RHS's: a flat RHS either

has no **where** clause, or has a **where** clause all of whose equations have variables as LHS's. For example,

```
f x = map ((+) 1) xs
      where
      xs = x : xs
```

is a simple recursion equation because although it has a **where** clause, the LHS of the equation defining xs consists only of a variable.

Alternative implementation techniques avoid this step, with the result that an environment data structure must be carried about at run-time, carrying the values of variables bound for the scope of a **where** clause. The λ-lifting process simply makes this environment explicit: it makes sure that each free variable is passed as a parameter to just those expressions which need it.

3.1.5 Strictness analysis

The definition of strictness was first given in section 2.4.1. In its simplest form it can be given in terms of a single function application with a single parameter: the function f is strict in its parameter if

$$\mathsf{f}\ \bot = \bot$$

Strictness analysis is a compiler algorithm which analyses the syntax of every function definition and every function application and detects in each case whether the equation above holds. If this is not easily deduced, perhaps because of run-time dependency or simply intractability, the analyser assumes that the function is not strict. The analysis algorithm employs a particularly elegant approach (called abstract interpretation) based on an abstraction of the language's standard semantics, in which each value is represented by an abstraction—either $\bot$ or "not-$\bot$". An introduction is given in [Jon87], [FH88] and in Hankin and Abramsky's introductory chapter in [AH87].

Once deduced, strictness information is manifest as *strictness annotations* appearing on strict applications. Thus, if the source program contains the application of two expressions e_1 and e_2:

$$\mathsf{e}_1\ \mathsf{e}_2$$

and the compiler deduces that e_1 is strict in its parameter, i.e. that

$$\mathsf{e}_1\ \bot = \bot$$

then the application is annotated:

$$\mathsf{e}_1\ \#\ \mathsf{e}_2$$

For complete strictness information it is also necessary to annotate the strict formal parameters in function definitions. If f is defined by the equation

$$\mathsf{f\ x\ y\ z} = \cdots$$

and f is found to be strict in its second parameter, its definition is annotated:

$$\mathsf{f\ x\ y}_{\#}\ \mathsf{z} = \cdots$$

This is necessary if strictness information is to be made available when a function is passed as a parameter and used in a context in which its identity cannot be known at compile-time (Peyton Jones covers this well [Jon87]).

Strictness and reduction order

The value of strictness information is in the freedom it confers on the order in which reductions are applied. With no strictness information, every application must be applied in normal order, leftmost first: where e_1 and e_2 are expressions, then in the application

$$\mathsf{e}_1\ \mathsf{e}_2$$

the evaluation of e_2 must be suspended until e_1 has been evaluated to a stage where it needs the value of e_2. This approach is often called the *call-by-need* parameter passing strategy. By contrast, with strictness information,

$$\mathsf{e}_1\ \#\ \mathsf{e}_2$$

we can evaluate e_2 much earlier—as soon as we know we need the result of the application as a whole. This is because we know that e_1 will eventually use e_2 (unless it is $\perp$ of its own accord!). For example, we can employ *call-by-value* parameter passing, where e_2 is evaluated completely before evaluation of e_1 begins. This is the choice taken by most older language designs for efficiency reasons, regardless of strictness.

Alternatively, *parallel* parameter passing can be used; see section 3.2.

3.1.6 Boxing analysis

The explanation given so far of program execution relies on a tree-structured representation of the expression being evaluated. The tree has a node wherever an application occurs, with the function being applied as the left sub-tree, and the parameter expression as the right sub-tree. The tree is represented by a linked structure in the computer's memory. When a parameter expression is passed into a function body, a pointer to the piece of graph representing the expression is transferred—rather than copying the expression. This is important not only to reduce copying and reduce space use: it is also necessary so that once a parameter does get evaluated, it can be overwritten with the value so that it need not be calculated again. Note that this means that after some rewriting the expression tree will have nodes with more than one parent; it becomes a general directed graph.

In this mechanism, the parameter is called *boxed*, and to find its value a pointer must be followed. This indirection is very expensive compared with passing the parameter "unboxed", in a register as might a compiler for a conventional language. Clearly a parameter can only be passed unboxed if it is passed by value.

This problem is slightly awkward because to take full advantage of unboxed parameter passing, the code generated for a function's body must be quite different, but the boxed interface must still be available for call-by-need invocations. In the presence of many parameters the number of versions needed of each function's code can be very large, so a compiler should emit code only for those variants actually used, and should impose a limit beyond which boxed parameter passing must be used.

Only with boxing analysis is the speed improvement due to strictness analysis realised (a space improvement almost always occurs with or without boxing: consider an accumulating parameter function like length).

3.1.7 Code generation

After simplification, strictness and boxing analysis, much of a typical program will be handled well by a conventional code generator. However, in the general case problems arise which are peculiar to the functional case: at the heart lies the problem that a function can be applied to a parameter in a context where the compiler cannot tell whether the resulting application is a redex. This can be resolved by carrying an parameter count with the function, but this overhead is undesirable, and can be avoided.

Several fast and successful implementations of code generators for lazy, higher-order functional languages have been in existence for some time, and are described, for example, by Peyton Jones [Jon87], Augustsson and Johnsson [Aug87, Joh87] and Fairbairn and Wray [FW87]. In order to give the interested reader a concrete understanding of implementation issues, a simple code generator is described here in some detail. It should be emphasised that the approach taken is representative of but different from the various existing compilers, and does not describe a particular completed compiler[1].

We begin the explanation with an outline of the scheme, together with some motivation. We will be more specific in the next section:

- We separate function application from object evaluation. The *apply* operation takes a function f and a parameter a, and builds a heap cell (called "an application box") containing the code

  ```
  f1: push a
      push f1
      jmp f
  ```

  The label f1 is the address of the heap cell.

- Thus, for example, when a three-parameter curried function is applied to three parameters a, b and c, three heap cells are occupied:

[1] The ideas presented in this section owe much to work originally done by Hugh Glaser of Southampton University

```
f1: push a      f2: push b      f3: push c
    push f1         push f2         push f3
    jmp f           jmp f1          jmp f2
```

The value f3 represents a fully-parameterised function application: it is a redex, although this may not be locally detectable at the point where the final parameter is provided.

- When the result of a function application is passed to a strict basic operator (take "+" as an example), the *evaluate* operator is applied to it first. The *evaluate* operator takes an application box as input, and returns an unboxed, evaluated object. The input must be of base (i.e. non-functional) type.
- When a function has computed its return result, it must update the application box so that it represents the evaluated object rather than the corresponding function application. For example, if f3 is a three-parameter function, the box f3 above is overwritten with

  ```
  f3: push x
      return
  ```

- The *evaluate* operator invokes the code to calculate its input (say f3 for example) by pushing some current state information, and then branching to f3. Each parameter and the pointer to each closure, is thereby stacked in turn, and finally the supercombinator f itself is executed.
- When f comes to return its result, it must ensure that a base-type normal form is ultimately returned. Thus, every supercombinator definition evaluates its result before returning. This happens naturally when the result returned by f is the output of a strict basic operator. However, not all functions do this—some just return (some component of) an input parameter, or an application box.

  In general it is possible that f is not a three-parameter function, but is instead a function of fewer parameters, which returns a function, and this returned function is then applied to the remaining parameters.

  In either case, the compiler can detect that the return result is not of base type. Instead or returning an unboxed, base-type result, such a function removes the parameters it has consumed from the top of the stack

The correctness of this scheme relies crucially on the *evaluate* operator being applied only to redexes. Thus, we cannot (in general) interpret strictness annotations on function-typed objects as call-by-value function application.

3.1.8 A simple code generator

For concreteness a simple code generator is sketched here. It does not deal with constructors, parameterless definitions or **where** clauses, and assumes that pattern matching

has been transformed away. In the first instance, it is fully lazy and takes no account of strictness or boxing analysis. The input is a list of equations:

```
SourceCode == [Equation]
Equation ::= EQUATION function [formalparameter] rhs
```

where function and formalparameter are identifiers,

```
function == identifier
formalparameter == identifier
```

The right hand side can be any expression. Expressions are either base-value constants, function constants, parameters, applications of primitive (and strict) operators such as "+", or applications of user-defined functions (think of the list of Expressions here as being a pair for the time being):

```
rhs == Expression

Expression ::= CONST Num |
               FUNCTION identifier |
               PARAM Num |
               ADD Expression Expression |
               APPLY [Expression]
```

It proves useful to separate right-hand sides into two classes, *value*-type or *graph*-type:

```
ValueType :: Expression → Bool
GraphType :: Expression → Bool

ValueType (CONST n) = TRUE
ValueType (FUNCTION f) = FALSE
ValueType (PARAM n) = FALSE
ValueType (ADD e1 e2) = TRUE
ValueType (APPLY [e1, e2]) = FALSE

GraphType exp = not (ValueType exp)
```

The code generator outputs a list of code blocks, one for each equation in the source code:

```
CodeGenerator :: SourceCode → [CodeBlock]
```

A code block is simply an entry point label and the associated instruction sequence:

```
CodeBlock ::= LABEL identifier [Instruction]
```

The abstract machine

The abstract machine's instruction set is defined by the data type Instruction:

```
Instruction ::= PUSHVALUE Num |
                PUSHGRAPH identifier |
                PUSHPARAM Num |
                ADD |
                APPLY |
                EVAL |
                UPDATEVALUE Num |
                UPDATEGRAPH identifier |
                BOX |
                RETJMP Num |
                RET
                JMP identifier
```

After optimisation, this representation is used to generate code for the target processor. This may involve register allocation and other issues which will not concern us here.

The abstract machine maintains a heap and a single stack, into which it has two pointer registers, the stack pointer and the frame pointer. The frame pointer points to the base in the stack of the current invocation frame, where the return address is kept, while the stack pointer points to the top of the stack. Thus, the stack just before a three-parameter function like f returns a result v will have the form

```
sp → v (return value)
     f1 (first application box)
     a (first parameter)
     f2 (second application box)
     b (second parameter)
     f3 (third application box)
     c (third parameter)
     ret (invocation return address, pushed by EVAL)
fp → ofp (pointer to base of previous frame, pushed by EVAL)
     (previous frame)
```

The PUSHVALUE n instruction places the unboxed constant n on the top of the stack. The PUSHGRAPH instruction takes a pointer to a function or heap cell and puts it on the top of the stack. PUSHPARAM n picks the n^{th} parameter out of the stack frame and pushes it onto the stack.

The APPLY instruction takes a function f and a parameter x from the top of the stack, and replaces them with a pointer f1 to a heap cell (an application box) containing the code

```
f1: PUSHGRAPH x
    PUSHGRAPH f1
    JMP f
```

The EVAL instruction takes a pointer to an application box from the top of the stack, pushes the current frame pointer and return address and jumps to the code in the box. On return, an unboxed base-type object will have been placed on the stack by the function.

The UPDATEVALUE n instruction is used when an n-parameter function has computed its return value, in the case when the value is unboxed. It takes the value v at the top of the stack and uses it to overwrite the n^{th} application box in its invocation chain. To do this it finds the application box pointer saved adjacent to the n^{th} parameter, and overwrites it with the code

```
fn: PUSH v
    RET
```

The UPDATEGRAPH n instruction is similarly used when an n-parameter function has computed its return value, but in the case when the value is boxed. It takes the pointer v (to the boxed result) at the top of the stack and uses it to overwrite the n^{th} application box in its invocation chain with the code

```
fn :: JMP v
```

The BOX instruction takes an unboxed value v from the top of the stack, and puts it in a box in the heap, just like the cell fn above. It leaves a pointer to the box on the top of the stack (this instruction will almost always be optimised out in code generators which take boxing analysis into account).

The RETJMP n instruction is used when a function has computed its return result and updated the corresponding application box using UPDATEGRAPH. The object being returned is still a pointer to a box. This may occur either because one of the parameters is being returned (and so might not yet have been evaluated), or because the result is of function type. It saves the value on the top of the stack in a temporary register (which is a pointer to the box being returned), removes the top n parameters from the top of the stack, and finally jumps to the saved pointer. The code thereby invoked will eventually compute the unboxed base-type object required, and return.

The RET instruction is used when a function has computed a base-type, unboxed result. It picks up the return address and old frame pointer from the base of the invocation frame, resets the stack pointer to the top of the old stack frame, and pushes the returned value.

The JMP instruction simply transfers control to the function or box named.

The translator

The key to understanding the code generator is to distinguish between two modes, *graph mode*, where code to build a heap-based graph is generated, and *value mode*, where code to calculate actual values is generated. The main optimisation task is to avoid graph mode. The code generator takes each equation and classifies its RHS as either value type or graph

type. It then calls the appropriate graph or value mode code generator:

```
CodeGenerator :: SourceCode → [CodeBlock]

CodeGenerator equations = map TranslateEquation eqns

TranslateEquation (EQUATION fname params rhs)
 = LABEL fname ((Ggen rhs)++[UPDATEGRAPH n, RETJMP n]),    if GraphType rhs
   where
   n = length params
 = LABEL fname ((Vgen rhs)++[UPDATEVALUE n, RET]),         otherwise
   where
   n = length params
```

The graph mode code generator Ggen generates code to build the function application tree of its result using APPLY and BOX:

```
Ggen :: Expression → [Instruction]

Ggen (CONST n) = [PUSHVALUE n, BOX]
Ggen (FUNCTION f) = [PUSHGRAPH f]
Ggen (PARAM n) = [PUSHPARAM n]
Ggen (ADD e1 e2) = [PUSHFUNCTION "addfunction"]
                       ++(Ggen e1)++[APPLY]++(Ggen e2)++[APPLY]
Ggen (APPLY [e1, e2]) = (Ggen e1)++(Ggen e2)++[APPLY]
```

The function identifier "addfunction" refers to a function which adds its two parameters. It can be compiled in value mode, but is needed here to suspend evaluation of the addition and its parameters during graph construction.

The value mode code generator is more straightforward, but must call Ggen to build functions and lazy parameters:

```
Vgen :: Expression → [Instruction]

Vgen (CONST n) = [PUSHVALUE n]
Vgen (FUNCTION f) = [PUSHGRAPH f, EVAL]
Vgen (PARAM n) = [PUSHPARAM n, EVAL]
Vgen (ADD e1 e2) = (Vgen e1)++(Vgen e2)++[ADD]
Vgen (APPLY [e1, e2]) = (Ggen e1)++(Ggen e2)++[APPLY]++[EVAL]
```

A simple example

As an example, consider the function definitions

```
f x = (ident x) + x
ident x = x
```

This program is represented in the SourceCode data type as

```
[ EQUATION "f" ["x"]
               (ADD (APPLY [FUNCTION "ident", PARAM 1]) (PARAM 1)),
  EQUATION "ident" ["x"]
               (PARAM 1)
]
```

Applying the code generator we find that "f" is compiled in value mode (Vgen), while "ident" is compiled in graph mode using Ggen:

```
[ LABEL "f"
  [ PUSHFUNCTION "f",
    PUSHPARAM 1,
    APPLY,
    EVAL,
    PUSHPARAM 1,
    EVAL,
    ADD,
    UPDATEVALUE 1,
    RET ]
  LABEL "ident"
  [ PUSHPARAM 1,
    UPDATEGRAPH 1,
    RETJMP 1 ] ]
```

Optimisations in the code generator

Much of the performance comes from optimisations making use of information about particular cases. The most basic make use of information available from the immediate context, from strictness information and from types. A small selection is given here:

- **Unshared applications:** When the APPLY instruction is used to build an application box, it is possible to detect from the context whether the resulting pointer might be copied. If not, the box need not be updated when the function is evaluated. Then the application box need not include code to push the box address, although it must still put something there to make sure the parameters are stacked in the frame correctly. We call this instruction PUSHDUMMY. It is unnecessary if a non-updating variant of the function being applied is used, but this is not likely to be worthwhile.

- **Combining EVAL and APPLY:** this optimisation is simply the observation that in a sequence of the form

[PUSH f, PUSH x, APPLY, EVAL]

the application box cannot be shared, and will be freed immediately. It therefore need not be UPDATEVALUEed, and can be replaced by the code

[PUSHSTATUS label, PUSH x, PUSHDUMMY, JMP f, DEFINELABEL label]

where label is an unused identifier, DEFINELABEL label associates label with the following instruction and PUSHSTATUS label pushes the current frame pointer and label onto the stack.

This optimisation derives from splitting EVAL into [PUSHSTATUS label, JMP *old top of stack*, DEFINELABEL label]. Then it is simply a storage class optimisation to move the PUSH from the application box into the instruction stream. The APPLY equation for Vgen becomes

```
Vgen (APPLY [e1, e2]) = [PUSHSTATUS label]
                        ++(Ggen e2)++[PUSHDUMMY]
                        ++(Ggen e1)++[PUSHDUMMY]
                        ++[JMP (top of stack),
                           DEFINELABEL label]
```

- **Multiple applications:** It is very common for a curried function to be applied to several parameters at once. The application boxes can be compressed into a single heap cell. Thus, as well as the binary application rules in Ggen and Vgen we have rules for 2, 3 or more parameters. For example,

  Ggen (APPLY [e1, e2, e3]) = (Ggen e1)++(Ggen e2)++[APPLY2]

  where the APPLY2 instruction builds an application box f containing the code

```
f: PUSHGRAPH (graph of e2)
   PUSHDUMMY
   PUSHGRAPH (graph of e3)
   PUSHGRAPH f
   JMP (graph of e1)
```

  This is simply an optimisation by branch elimination of the code

f_{n-1}: PUSHGRAPH (graph of e3)
PUSHGRAPH f
f: JMP (graph of e1)
PUSHGRAPH (graph of e2)
PUSHDUMMY
JMP f_{n-1}

- **Strict applications:** when a function f is known to be strict, and its value is known to be of base type, then the value-mode code generator can generate code to apply EVAL to a parameter before passing it:

  ```
  Vgen (STRICTAPPLY [e1, e2]) = (Ggen e1)
                                ++(Ggen e2)++[EVAL]
                                ++[APPLY]++[EVAL]
  ```

  Things are not quite so simple because now e2 is passed to e1 unboxed. It would be easy to arrange a pointer to its box to be passed instead, but more efficient would be to use the equation

  ```
  Vgen (STRICTAPPLY [e1, e2]) = (Ggen e1)
                                ++(Vgen e2)
                                ++[APPLY]++[EVAL]
  ```

  (as well as the other optimisations listed above). To do this requires a variant of e1 to be used which expects its parameter unboxed, and this cannot in general be managed.

  A similar problem arises when trying to avoid having to BOX constants before passing them as parameters.

- **Tail recursion:** a tail recursive function is one whose result is an application. As things stand, the code generated will build a heap-based application box representing the application being returned, update the corresponding application box, clear the parameters consumed from the stack, and then jump to the tail-recursive call. This avoids needlessly consuming stack space, but is inefficient because the update is unnecessary, and because the parameters could be updated in place rather than being built in the heap and then copied onto the stack.

  It is important to deal with tail recursion well, as this is how loops are manifest. It is quite complicated, and the reader is referred to the literature review (section 3.4) for details.

To conclude this rather complicated code generation scheme, we note that we have avoided any run-time testing of the graph to determine whether functions can be invoked, and we have avoided tagging objects with their type. The costly aspects of the language are

- non-strict functions, requiring parameters to be passed as graph,

- updating application boxes, requiring pointers to boxes to be passed with parameters, in conjunction with higher-order and polymorphic functions.

Much of the overhead can be reduced by generating multiple variants of each function's code, but this is not always acceptable.

3.1.9 Garbage collection

Functional language implementations are very reliant on high-performance garbage collection. Very careful design of run-time data structures is required to allow unused heap storage space to be detected and collected efficiently. Moreover, garbage must be made available for collection as soon as possible, requiring some potentially quite expensive accounting as pointers are destroyed. Compile-time optimisation of this garbage accounting activity is an active research area.

3.2 Parallel graph reduction

In the last section compilation techniques were discussed for execution of a functional programming language on a single, conventional processing element. However, it was very common in examples of reduction that several redexes could be reduced in parallel. This is actually done by the large family of *parallel graph reduction machines* being constructed, including ALICE [DCF+87] GRIP [JCH85], ALFALFA [GH86b], FLAGSHIP [WSWW87] and others.

In order to understand how parallelism is exploited in these architectures, we examine how and when potentially-concurrent tasks are created, and how they interact with one another.

3.2.1 Processes

Under sequential evaluation, there is a single reduction process, which applies a normal-order reduction strategy modified by strictness annotations to include call-by-value parameter passing. Under parallel graph reduction, there may be many such processes, each evaluating a different sub-graph. There are several overheads paid by parallel graph reduction machines against which the potential speedup must be weighed:

Fork overhead: The cost of creating a new process arises in three ways:

1. Constructing the graph representing the expression. This is the same as the cost of call-by-need parameter passing. It is to be compared with the lower cost of call-by-value parameter passing.
2. Placing a new process descriptor in a process pool to await scheduling. The new process descriptor will contain a reference to the graph of the expression in question, and when necessary, the identifier of the process which spawns it. Other processors may take descriptors from this pool, thus *migrating* the work across the machine.

3. Constructing and entering a new process when the graph reference is scheduled for execution.

In addition, there may be a cost associated with distributing the graph reference to another processing element for execution. We can account for all these costs as an average *fork* overhead chargeable for every process creation. Note that much of this cost is incurred whether or not the process is actually distributed to another processing element.

Synchronisation control: Because of sharing in the graph, a process may attempt to reduce a node in the graph which is already being reduced by another process. If allowed to proceed, considerable chaos will result. To prevent this, a marker must be placed on a node—signifying that "work is in progress below"—to ensure mutual exclusion whenever a reduction process attempts to reduce a node. The marker can be removed when the node is rewritten to normal form.

A process which needs the value of a marked node before it can proceed must suspend itself, after arranging to be re-awoken when the mark is removed.

Join synchronisation: When a reduction process successfully terminates, having reduced its expression graph to normal form, it overwrites the root node of the graph it reduced with the result, and removes its mutual-exclusion marker.

By this time, several other processes may be suspended awaiting this result. The identifiers of each waiting process will be held in a *pending list* associated with the node. The last thing a process does is to awaken these processes by informing the scheduler that they can be resumed.

Memory access interference: The multiple reduction processors must have fast access to the shared graph data structure. This requires a complex communications and arbitration system which incurs a delay on accesses to the graph. In the first instance, when assessing the performance issues for parallel graph reduction machines, this delay is assumed relatively small. This can be achieved using sophisticated interconnection network technology (surveyed in [WF84]), at a considerable cost. In section 3.2.3 we will see the influence of a poorer interconnection network.

As well as these additions to the amount of *work* a parallel graph reduction machine does, there is a severe increase in the *space* occupied. We return to this question in section 3.4.

3.2.2 Partitioning

In principle, a new process can be created whenever an already-existing process discovers a strict application. However, some processes terminate after doing very little useful work—and this can be dwarfed by the fork and join overheads incurred by the attempt to employ parallel reduction. When this is the case, it is wiser to generate code for call-by-value parameter passing. In general, we would require a compiler to prove for each strict application that a decision to use parallel reduction rather than fast sequential reduction

will not incur a substantial cost. This approach has been taken by Hudak and Goldberg with their "serial combinator" compilation technique [HG85].

Their strategy can be used to guarantee some speed-up due to parallelism—and should certainly ensure that the attempt to exploit parallelism does not result in a slow-down. Just how much speed-up depends on how much parallelism is actually present in the source program *after the grain-size analysis*. Some highly-parallel programs may contain no expressions which the compiler can guarantee are worth distributing. Worse yet, some architectures may have fork-join overheads so high that distributable expressions are very rare in any program at all.

The problem of ensuring a non-negative speed-up is far easier than arranging for really good performance. Particular algorithm structures such as divide-and-conquer (see Chapter 4) are well-understood, but in general considerable understanding of the algorithm is required, in order to select just the right expressions for which to spawn processes. The target is to maximise the grain size while still providing sufficient parallelism to exploit the machine's resources. Of course, this all depends on the program itself having a good parallel structure. To get the best from such a machine, these issues must become the concern of the programmer, and the approach of Chapter 5 is applicable.

3.2.3 Loosely-coupled parallel graph reduction machines

Up to now, we have assumed a tightly-coupled underlying architecture, in which access to a non-local processing element's memory is not much slower than access to local memory. If this is not so, the performance issues become much more complicated.

The first aspect of the problem can be considered to be with the notion of "grain-size". This was defined for the tightly-coupled case to be the amount of work done by a process between being created (when the fork overhead is incurred) and terminating (when the join overhead is incurred). We can simply compare the process's (minimum or likely) execution time with the total overhead to decide whether organising a new process is worthwhile.

In a loosely-coupled machine, a non-trivial overhead is incurred every time a process makes a non-local memory reference. We are therefore forced to think of the grain size as the amount of work done between non-local memory accesses. This rather complicates the calculation, and certainly reduces the proportion of strict applications which can be implemented safely using parallelism.

When an expression is passed from one processing element (say A) to another processing element (say B) for parallel evaluation, its parameters have to be accessed non-locally, by B from A, and any structure returned will probably be constructed by B, in B's memory, and so will be accessed non-locally by A.

Note that once an object has been evaluated to normal form, it can be copied (without introducing recalculation). Thus, when B accesses a parameter structure, it need only access each non-local node once. Similarly, when A accesses the result of the parallel evaluation, it need examine each node of the returned structure at most once (FLAGSHIP [WSWW87] employs this technique). Thus the overhead due to parameter/result access incurred by distribution is bounded by the size of the parameter and result structures.

Eager evaluation of lists

Recursively-defined data types such as lists and trees can have unbounded size. However, they are evaluated piecemeal, just enough to expose the outermost constructor (this is technically called *weak head normal form*). Thus the overhead incurred by non-local access to such a structure is limited by the number of parameters taken by the data type's constructor. Access to a subtree, or to the tail of a list, would constitute a quite separate evaluation and the advantage of employing parallelism can be considered separately.

However, lists can be treated differently from other structures because of their sequential access mode. Remote access latency can be avoided by calculating several elements ahead, and sending to the consuming processor before they are demanded. This requires strictness analysis to be applied to ensure that unwanted computations are not spawned. Burn's work on evaluation transformers, reported in [Bur87a] forms a basis for this approach. See also the process network view presented in section 4.3.

3.2.4 Neighbour-coupled parallel graph reduction machines

The neighbour-coupled architecture, introduced in section 1.3, is an interesting intermediate architecture for parallel graph reduction. Recall that in these architectures, a general, random, non-local access is relatively slow, just as in a loosely-coupled machine. The difference is that each processing element has a few neighbours to which it is tightly-coupled.

Now much of the problem with compile-time performance analysis can be simplified provided that a process is not migrated to a non-neighbour of the processing element which spawned it. With care, we can ensure that all parameters are available in the spawning processing element's local memory (if necessary by judicious copying).

3.3 Conclusion

This chapter has given a very brief overview of the graph reduction implementation technique for functional programming languages. Substantial optimisations can be applied and very high sequential performance can be achieved this way. We went on to examine how parallelism can be applied to speed up graph reduction. A qualitative analysis of the costs of parallel graph reduction demonstrated that the approach is well-suited to tightly-coupled parallel computers, but that in a loosely-coupled machine the cost of remote memory accesses dominates, and that compile-time process distribution becomes intractable.

3.4 Pointers into the literature

Standard works on compilers

Despite the additional problems of functional languages, the standard texts on compiler design are indispensable. Examples might include Gries [Gri71], Aho, Sethi and Ullman [ASU86] and Wulf and his colleagues [WJW+75].

Approaches to compiling functional programs

There are compilation problems special to lazy and higher-order languages, and researchers studying the area have developed a number of different abstract machine designs. Like ours, these generally form a simple, well-understood instruction set for an imaginary computer, and can be translated into instructions for a real machine. Field and Harrison [FH88] cover several different approaches well.

Abstract machines can be divided into two categories: **environment-based** and **combinator-based**.

- **Combinator-based abstract machines:** this chapter has described a combinator-based approach, where the compiler simplifies the program so that references to non-local, non-global variables are transformed into parameter references. This avoids the need for environment links (or displays), simplifies function invocation and is claimed to reduce contention for the environment between parallel reductions.

  The first appearance of this idea is Turner's combinator reduction machine [Tur79]. Turner translated programs into a fixed set of simple combinators (based on S, K and I, introduced in section 2.2.8), which form the abstract machine. Although the set described in the paper is small, optimised implementations use a large combinator set incorporating many of the language's library functions. Clarke and his colleagues at Cambridge University built a prototype sequential architecture (called SKIM) microcoded to support such combinators as its instruction set [CGMN80]. Stoye (in [Sto85]) presents a deeper study, developing the instruction set towards more conventional machines.

  Apparently concurrent work by Hughes [Hug83], Augustsson and Johnsson [Joh84b] developed algorithms to construct a combinator set especially for each program. The body of each combinator can then be translated into code for a conventional computer. This is done via the G-machine abstract machine by Augustsson and Johnsson's Lazy ML compiler, and is described in detail in [Aug87] and [Joh87], where substantial optimisations are presented. This material is given in simplified form in Peyton Jones textbook [Jon87]. Hughes approach (called supercombinator abstraction) maintains non-recomputation of shared subexpressions which may be compromised by Augustsson and Johnsson's simpler λ-lifting algorithm. It is not clear whether the overheads introduced by Hughes' algorithm are justified, but Goldberg [Gol87] gives an analysis which determines when the recomputation might occur. The Ponder compiler, described by Fairbairn [Fai82], uses similar techniques.

  The code generator presented in section 3.1.8 is based on ideas from Glaser and Hayes [GH86a] and Fairbairn and Wray [FW87], with much help from Hugh Glaser, Sebastian Hunt and Tony Field which is gratefully acknowledged.

- **Environment-based abstract machines:** These extend and formalise the conventional approach to implementation of block-structured programming languages. The λ-lifting phase is omitted, so that references to non-local, non-global variables remain. Each function maintains not only its own local environment, but also a pointer to a linked list of environment records, each holding values of non-local non-global variables it might refer to. In a higher-order language, this chain may include

the local environments of functions which have already returned. They must, therefore, be kept in the heap rather than on the stack, as is possible in conventional block-structured languages.

The first example is the SECD machine, introduced by Landin [Lan64]. A thorough treatment is given, including a lazy variant and a correctness proof, by Field and Harrison [FH88]. For generation of high-performance code, they also describe an optimised variant called FPM. The categorical abstract machine, CAM, can also be thought of as an optimised SECD-style evaluator. It is interesting in that its instructions are just combinators, drawn from a fixed set (Categorical Combinatory Logic). See Field and Harrison and Curien [Cur86], although the latter is quite theoretically-oriented.

An interesting variation was proposed by Steele in his RABBIT compiler prototype [Ste78], a more accessible presentation being [Kra88]. These compilers begin with a transformation phase resulting in a "continuation-passing style" (CPS) formulation of the program. This makes a function's return address an explicit parameter (of function type), called a continuation. When a function returns a value, the CPS function passes the value as a parameter to the continuation. CPS style programs can be evaluated by a simplified interpreter which does not retain function return addresses. The aim of this transformation is shift the data structures needed to manage control-flow into the domain of values. This makes them available for conventional value-based optimisations. It also makes the treatment of tail recursion more straightforward.

Compiling pattern matching

Various approaches have been described by Wadler, in Peyton Jones textbook [Jon87], Field and Harrison [FH88] and Augustsson [Aug87]. More general work on pattern matching has been done by Hoffman, O'Donnell and Strandh [HOS85], among others. Interest in pattern matching extends to the theorem proving and computer algebra communities; Klop [Klo90] and Huet and Oppen [HO80] cover some of the area.

Strictness analysis

Strictness analysis can be approached using conventional data flow analysis, but has proven a very successful application of abstract interpretation. This has the advantage of handling inter-functional dependency, recursion, higher-order functions and data structures. An introduction to abstract interpretation is given by Abramsky and Hankin in their introduction to [AH87].

Strictness analysis of first-order programs (or first-order parts of higher-order programs) was first described by Mycroft [Myc81], and this has been implemented with very positive results in Augustsson and Johnsson's Lazy ML compiler [Aug87, Joh87]. This was extended by Burn, Hankin and Abramsky and Peyton Jones to higher-order programs (see [HBJ88]), although efficiency problems with implementations of this scheme have yet to be resolved. Backwards analysis, as proposed by Hughes [Hug87], may prove a more practical alternative.

Extensions to discover strictness information about lists have been made by Wadler [Wad87], Burn [Bur87a] and others. The practical application of strictness analysis on lists is still a research topic; different approximations seem appropriate for different purposes. See, for example, Chapter 5 section 5.4.5.

Compile-time simplification

Performing large-scale simplification of programs is still very much an experimental technique. For a general review of partial evaluation, see page 160. An example of a complete compiler based on simplification is described by Hudak and Kranz [HK84]. Particular techniques are described by Wadler [Wad88b, Wad88a].

Store management and garbage collection

The assignment operation "$\mathsf{x} := \mathsf{x} + 1$" can be interpreted as a hint to the compiler that the old contents of cell x are no longer required, and the space can be reused to accommodate the value $\mathsf{x} + 1$. As functional languages have no such construct, other means must be found to reclaim memory space when it is no longer needed. Some of this can be done at compile-time, but at present most is the responsibility of the run-time system.

Run-time storage reclamation can roughly be divided into two quite different approaches: copying and reference counting.

- **Copying schemes:** The starting point for these algorithms is to separate the working memory into two parts, the TOSPACE and the FROMSPACE. When garbage collection occurs, data objects in use are copied from the FROMSPACE to the TOSPACE. After garbage collection, free space and allocated space form two adjacent contiguous blocks. In its simplest form, TOSPACE and FROMSPACE are statically allocated and of equal size, so half of the available memory is wasted. After collection, the rôles of the two spaces are reversed.

  An important advantage of copying is that the memory is compacted, so improving the performance of a virtual memory system; this can be further improved by strategies like using depth-first copying to locate linked objects near to one another.

  Although schemes do exist which eliminate the waste of the two-space method, a more attractive approach is to split the memory into many spaces, only one of which need be empty at once. This is described by Lieberman and Hewitt [LH83]. The spaces are ordered by age—the more garbage collections an object survives, the deeper in the vector of spaces it resides. Thus, most collections need deal only with the youngest objects. Unfortunately, Lieberman and Hewitt's scheme assumes that most pointers point to older objects. In the presence of lazy evaluation, assignment or logic variables this can often be far from the case, and then a substantial overhead is incurred. Moon [Moo84] describes a similar but much more complicated scheme, using substantial hardware support, to resolve these problems with high performance.

  Copying garbage collection is not invoked until free space becomes short, and the larger the physical memory the less often this need occur. The cost of each collection depends only on the amount of space occupied by non-garbage. This leads

to a startling conclusion: with enough memory we can make the garbage collection overhead asymptotically approach zero. When memory is short, on the other hand, performance can be very poor.

- **Reference counting schemes:** An alternative to copying is simply to keep a count with each cell of the number of pointers to it. When a pointer is copied or destroyed, this count is adjusted, and when it reaches zero the cell is marked reallocable. The main advantage of reference counting is that the rate and response time of the processing is always constant. Its main problem is that it fails for cyclic structures. There is no opportunity for compaction, so great care must be taken to place cells to maximise locality when virtual memory is in use. Finally, the overhead of reference counting depends on the amount of copying and deletion of pointers. Nonetheless, a great deal of work has been done in the area, particularly in parallel systems where copying schemes become rather complicated. For parallel systems with packet-switched interconnection, a variation on the scheme is necessary to avoid race conditions [WW87, Bev87][2], where "weights" are carried with the pointers instead of counts with the cells. Various other variations have been described by Glaser and his colleagues (e.g. [GT84]).

In principle, garbage collection can be avoided by compile-time scheduling of memory use. This has proven difficult, although attempts have been made by Mycroft [Myc81] and Hudak and Bloss [Hud87, HB84] and others. More fruitful to date have been transformation tactics which eliminate intermediate data structures. This is very common during derivations given in this book. Wadler [Wad88b] attempts to formulate strategies suitable for inclusion in optimising compilers.

Space leaks

Because they lack explicit control over space re-use, functional programs have a tendency to consume large quantities of space as they run. In some cases, this can be quite disastrous, and quite unnecessary. The problems can arise in several ways:

1. The program may necessarily demand more space than a more reasonable implementation would require. This can happen quite accidentally. Take, for example, this function definition, which contains duplicated common subexpressions:

```
f xs ys = cond ((sum (map g xs)) > 1)
               (cond (h ys)
                     (cond ((sum (map g xs)) < 10)
                           a
                           b)
                     c)
               d
```

   (the **cond** is used to force the three conditions to be evaluated sequentially). If we

[2]The idea seems also to have been current in dataflow circles at MIT as early as 1979

abstract the expression map g xs using a **where** clause, we reduce the amount of work done:

```
f xs ys = cond ((sum gxs) > 1)
               (cond (h ys)
                     (cond ((sum gxs) < 10)
                           a
                           b)
                     c)
               d
          where
          gxs = map g xs
```

Unfortunately this means that the list gxs must be retained in memory during the evaluation of h ys. There may not be enough memory remaining for this computation.

2. The space occupied may depend on the evaluation order. An example might be a function mean, specified by the equation

   ```
   mean as = (sum as)/(length as)
   ```

   A conventional sequential evaluator would select either the sum or the length calculation to perform first, leaving the other to do second. Either way means the list as must be held in memory in its entirety. There does exist a reduction order which evaluates both expressions in step (a data-driven order, for example). This program can be rewritten to make the step-by-step calculation explicit, but much of the value of a functional formulation is lost. This problem is approached in more depth by Hughes [Hug83].

3. The space may be inadvertently retained by the implementation, even though it cannot be reached. A common way this can happen is when several variables are held in a function's activation record. The variables may become garbage before the activation record does, but many implementations will not free the variables until the activation record is freed. With reference counting a similar problem occurs if reference decrement code is migrated across function invocations. This is partially addressed by Wadler [Wad86].

Parallel Graph Reduction

The principles of parallel graph reduction are reviewed in Chapter 24 of Peyton Jones' textbook [Jon87]. This conceptual basis was generalised and first implemented in the prototype ALICE machine, by Darlington, Cripps and their colleagues [DCF+87]. The ALICE work was the foundation for the FLAGSHIP project [WSWW87], where attempts to generalise the graph-rewriting model of parallel computation have been crystallised in the DACTL language design. DACTL [GKS87], and the related language Lean [BvEG+87a], extend the term-rewriting basis of functional programming to the more general rewriting

of linked graph structures, and have much in common with the ALICE compiler target language CTL. An important result of this work has been the formal verification that graph reduction implements functional languages properly (the more general result is given by Barendregt et al. [BvEG+87b], but an interesting algebraic approach is presented by van der Broek and van der Hoeven [vdBvdH86]).

Many other research groups have implemented or studied parallel graph reduction, and a complete list is impossible. Most notable might be the GRIP machine being constructed by Peyton Jones and his colleagues [PCSH87] and the ALFALFA and BUCKWHEAT implementations by Goldberg and Hudak [Gol88], whose partitioning [HG85] and work diffusion [HG84] studies and especially interesting. Other design studies include Bevan, Burn and Karia's [BBK87] and Keller and Lin's REDIFLOW machine [KL84, KSL86].

Other approaches to parallel code generation

There is not room here to cover even a fraction of the general literature concerned with the problem of taking a program with little or no specific control over parallel execution, and generating parallel object code from it. A fundamental distinction can be drawn between run-time scheduled object code and compile-time scheduled object code. With run-time scheduling the compilation problem is mainly concerned with partitioning the problem into large-grain processes in order to overcome the overhead of run-time process management. With compile-time scheduling, a much finer "grain" of processing can be employed—typically at the level of instructions —because high locality can be arranged and synchronisation delays can be avoided.

Sarkar [Sar89] and Goldberg [Gol88] describe recent quite successful approaches to the partitioning problem. Sarkar also approaches the problem of compile-time scheduling of large-grain processes to gain yet higher performance.

The compile-time scheduling literature goes back much further, because of the early prevalence of vector pipeline processors (of which the CRAY-1 [Rus78] is the classical example). An example of this work might be Kuck et al. [KKLW81]. Long instruction word architectures have led to other interesting fine-grain compile-time scheduling compilers. See for example Ellis' BULLDOG compiler [Ell82] and Aiken and Nicolau [AN88]. Wolfe [Wol89] gives a more unifying view, employing vector operations where possible (affecting innermost loops), but introducing large-grain processes at the outermost level as well. This kind of compiler is finding some commercial success with recent parallel processor systems [TMS87].

Chapter 4

Specifying and Deriving Parallel Algorithms

This chapter has two aims:

- to investigate how parallelism can be expressed in the form of a functional program,
- to develop techniques for transforming programs from one formulation into another, in order to express parallelism in different ways,
- to illustrate some of the techniques with simple examples, culminating in a simple pipelined ray-tracing program.

Several of the more involved transformations and verifications have been collected separately, and appear as Appendix A; they would interfere with the development of the first aim, to understand how parallelism appears in the code. They are, however, quite important to the second aim, of building a toolbox of techniques for changing the parallelism in a program, and the reader is encouraged to follow the Appendix on the second reading.

4.1 Horizontal and vertical parallelism

We have discussed how the graph-rewriting view of expression evaluation can be used to exploit parallel hardware. But what can we say about the structure of parallel computations under this regime?

Goldberg [Gol88] distinguishes two sources of parallelism in parallel graph reduction:

- **Horizontal** parallelism occurs when two or more of a function's parameters are evaluated in parallel.
- **Vertical** parallelism occurs when a parameter is evaluated in parallel with the function application to which it is being passed.

A simple example of purely horizontal parallelism is when a strict, built-in operator such as "+" is applied. In an application like

$(+)\ e_1\ e_2$

the parameter expressions e_1 and e_2 can be evaluated in parallel, but both must finish before the addition can proceed.

Vertical parallelism can occur whenever a parameter is passed to a strict, user-defined function. The parameter is evaluated in the time "window" between function application and use of the parameter by a strict, built-in operator like addition. For example, define

```
f x y = y+1,              if x = 0
f x y = f (x−1) y         if x > 0
```

A good strictness analyser will infer that f is strict in both its parameters (parameter x is always used; parameter y is used whenever f terminates)[1]. Now suppose we have defined g so that

```
g y = f 10000 y
```

Now suppose we have an application of g to an expression e_1:

```
g # e1
```

Given two processing elements, it should be clear how one processor can be occupied counting down from 10000 while the other evaluates e_1.

Horizontal and vertical parallelism account for all the parallelism available in a parallel graph reduction machine. Each leads to a different algorithmic structure. We identify these as the *divide-and-conquer* structure, which exploits horizontal parallelism, and *pipelining*, which exploits vertical parallelism.

4.2 Divide-and-conquer parallelism

The colonial maxim "Divide-and-conquer" has broad application in computer science. We can characterise a divide-and-conquer algorithm by a functional program scheme. Solve solves some problem, described by its parameter problem, using the divide-and-conquer approach:

```
Solve :: α → β

Solve problem = SimplySolve problem,                                  if Trivial problem
Solve problem = CombineSolutions problem (map Solve SubProblems)      otherwise
                where
                SubProblems = Decompose problem
```

where SimplySolve, CombineSolutions, Decompose and Trivial depend on the particular divide-and-conquer algorithm. They have the types

[1] Of course, a good optimiser would remove the calculations involving x since their results are never used

SimplySolve :: $\alpha \to \beta$

CombineSolutions :: $\alpha \to [\beta] \to \beta$

Decompose :: $\alpha \to [\alpha]$

Trivial :: $\alpha \to$ Bool

If the problem to be solved is trivially simple, it is solved directly using SimplySolve. If not, the problem is broken down into a list of subproblems. These are each solved separately (using map Solve), and finally CombineSolutions uses the list of solutions to the subproblems to solve the original problem. Provided CombineSolutions is known to be strict in each element of its list parameter SubProblems, plentiful horizontal parallelism is available.

4.2.1 Divide-and-conquer examples

We complete the characterisation of divide-and-conquer by giving a function which applies the divide-and-conquer strategy given definitions of the component functions:

```
DivideAndConquer :: (α → β)
                    → (α → [β] → β)
                    → (α → [α])
                    → (α → Bool)
                    → α
                    → β


DivideAndConquer SimplySolve CombineSolutions Decompose Trivial problem
= Solve problem
  where
  Solve problem = SimplySolve problem,                          if Trivial problem
  Solve problem = CombineSolutions problem
                                    (map Solve SubProblems),otherwise
                  where
                  SubProblems = Decompose problem
```

There follow four examples of how DivideAndConquer can be used in practice: in the Fibonacci recurrence, in the Quicksort algorithm, as a parallel implementation of insert, and to reduce overheads in a parallel implementation of map.

The Fibonacci Function

This is naturally defined by the recurrence relation

```
fib n = 1,                          if n ≤ 2
fib n = fib (n−1) + fib (n−2),      otherwise
```

This is a working functional program (although far better ways of calculating the Fibonacci numbers exist).

We can see that it has the form of a divide-and-conquer algorithm by writing its definition in terms of DivideAndConquer:

```
fib = DivideAndConquer (const 1)
                       (const sum)
                       (construct [(subtract 1), (subtract 2)])
                       ((≤) 2)
```

Here const 1 returns 1 whatever its parameter. The function sum adds the elements of a list of numbers. subtract n decrements its parameter by n. The function construct is analogous to map, but takes a list of functions and applies each one to the same parameter:

```
construct :: [α → β] → α → β

construct [ ] x = [ ]
construct (f : fs) x = (f x) : (construct fs x)
```

The Quicksort Algorithm

There are many parallel sorting algorithms, including several divide-and-conquer ones. This one is particularly straightforward. We define SelectSmaller to select all those elements of an input list smaller than some "pivot" value:

```
SelectSmaller :: Num → [Num] → [Num]

SelectSmaller pivot as = filter ((>) pivot) as
```

SelectBigger is similar:

```
SelectBigger :: Num → [Num] → [Num]

SelectBigger pivot as = filter ((≤) pivot) as
```

The function filter eliminates elements from a list unless they satisfy the predicate:

```
filter :: (α → Bool) → [α] → [α]

filter predicate [ ] = [ ]
filter predicate (a : as) = a : (filter predicate as),        if predicate a
filter predicate (a : as) =     (filter predicate as),        otherwise
```

Now the sort function is easily defined:

```
QuickSort [ ] = [ ]
QuickSort as = (QuickSort SmallerOnes) ++ (Quicksort BiggerOnes)
               where
               SmallerOnes = SelectSmaller pivot as
               BiggerOnes = SelectBigger pivot as
               pivot = hd as
```

The choice of pivot element can have a drastic effect on the algorithm's performance unless the input is truly randomly ordered.

This algorithm can be represented using DivideAndConquer as

```
QuickSort = DivideAndConquer (const [ ]),
                             (const ListAppend),
                             PivotAndSplit,
                             ((=) [ ])
            where
            ListAppend [as, bs] = as ++ bs
            PivotAndSplit as = [ SelectSmaller pivot as,
                                 SelectBigger pivot as ]
                               where
                               pivot = hd as
```

(ListAppend is just a special case of join = insert (++) [ ]).

The Insert function

In Chapter 2 a function called insert was introduced, which takes an associative function, which we denoted by the infix operator op, and is defined informally by the equations

```
insert (op) base [ ] = base
insert (op) base [a1, a2, a3, ··· aN] = a1 op a2 op a3 ··· op aN
```

This is unambiguous provided (op) is associative, when we can place brackets wherever convenient on the RHS. We will also require that base have the property that for all a,

```
a op base = a = base op a
```

A simple example is summation,

```
sum as = insert (+) 0 as
```

Given these restrictions, we can employ a divide-and-conquer implementation:

```
insert (op) base as
 = DivideAndConquer (const base) (const ListOp) ListSplit
   where
   ListOp [a, b] = a op b
   ListSplit as = [ take m as,
                    drop m as ]
                  where
                  m = (length as)/2
```

A useful more general approach to this is to transform the data type being used to represent as, from a list to a tree. Let us employ the following binary tree data type:

```
BinaryTree α ::= EMPTY |
                 LEAF α |
                 NODE (BinaryTree α) (BinaryTree α)
```

We need a pair of functions to turn the list into a tree, and *vice versa*:

```
ListToTree :: [α] → BinaryTree α
TreeToList :: BinaryTree α → [α]
```

and we specify that for all finite and total lists as,

```
TreeToList (ListToTree as) = as
```

Probably the most natural definitions for this pair of functions are:

```
ListToTree1 [ ] = EMPTY
ListToTree1 [a] = LEAF a
ListToTree1 (a0:a1:as) = NODE (ListToTree1 (take m (a0:a1:as)))
                              (ListToTree1 (drop m (a0:a1:as))),   if length as > 1
                         where
                         m = (length (a0:a1:as))/2
```

(it is necessary to introduce (a0:a1:as) to avoid ambiguity).

```
TreeToList1 EMPTY = [ ]
TreeToList1 (LEAF a) = [a]
TreeToList1 (NODE subtree1 subtree2) = (TreeToList1 subtree1)
                                       ++ (TreeToList1 subtree2)
```

where

```
take :: Num → [α] → [α]

take n (a : as) = a : (take (n−1) as),          if n ≠ 0
take n [ ] = [ ],                               if n ≠ 0
take 0 as = [ ]
```

and

```
drop :: Num → [α] → [α]

drop n (a : as) = drop (n−1) as,          if n ≠ 0
drop n [ ] = [ ],                        if n ≠ 0
drop 0 as = as
```

In Appendix A (Theorem 1) a proof is given that these functions do satisfy the specification. More importantly, there are very serious inefficiencies in the definitions as given and in the appendix a much more efficient, though more complicated, definition is derived.

Now we can turn the list into its tree representation, we must arrange to exploit the divide-and-conquer structure available. We have

```
insert (op) base as = insert (op) base (TreeToList1 (ListToTree1 as))
```

We apply equational reasoning to improve on this. First, let us name our new version,

```
TreeInsert (op) base as = insert (op) base (TreeToList1 (ListToTree1 as))
```

Now, instantiate it for the case when as has more than one element, and apply the appropriate equation for ListToTree1:

```
TreeInsert (op) base as
 = insert (op) base
          (TreeToList1 (ListToTree1 as) ),    if length as > 1
 = insert (op) base
          (TreeToList1 (NODE (ListToTree1 (take m as))
                             (ListToTree1 (drop m as)))),    if length as > 1
    where
    m = (length as)/2
 = insert (op) base ( (TreeToList1 (ListToTree1 (take m as)))
                      ++
                      (TreeToList1 (ListToTree1 (drop m as))) ),    if length as > 1
    where
    m = (length as)/2
```

At this point we must use a straightforward extension of associativity:

```
insert (op) base (as++bs) = (insert as base) op (insert bs base)
```

The result is

```
TreeInsert (op) base as
 = (insert (op) base (TreeToList1 (ListToTree1 (take m as))))
   op
   (insert (op) base (TreeToList1 (ListToTree1 (drop m as)))), if length as > 1
   where
   m = (length as)/2
```

Finally, applying the original equation defining TreeInsert, in reverse, we get

```
TreeInsert (op) base as
 = (TreeInsert (op) base (take m as))
   op
   (TreeInsert (op) base (drop m as)),    if length as > 1
   where
   m = (length as)/2
```

The remaining equations required to define TreeInsert for empty and singleton lists are easily derived:

```
TreeInsert (op) base [ ] = base
TreeInsert (op) base [a] = a
```

Such a transformation is likely to work well if the time required to apply the function op is quite substantial. However, the tree-based version clearly does more work and once sufficient parallelism has been generated, execution could revert from the expensive, parallel tree-based definition of insert to the original list-based one. It is conceivable that such a decision could be taken at run-time.

This transformation example brings out a rather complicated and interesting problem for program transformation technology: we introduced simple definitions for ListToTree1 and TreeToList1, and then used them to derive a parallel version of insert. Meanwhile, in Appendix A, the very inefficient definitions of ListToTree1 and TreeToList1 are optimised substantially. The optimisations do not destroy the possibility of a divide-and-conquer version of insert based on the optimised definitions, but we need to go through the derivation again. Because the derivation shares the same structure as before, and uses the same properties, we can hope that a computer could help.

The map function

The function map, defined by the equations

```
map f [ ] = [ ]
map f (a : as) = (f a) : (map f as)
```

has a clear interpretation for parallel programming: spawn a process to evaluate f a_i for each a_i of the input list. Provided sufficient strictness information is available, this is just what happens. It is slightly unsatisfactory because the processes must be spawned sequentially, once for each time an application of map is rewritten.

Just as with insert, we can apply divide-and-conquer by repeatedly sub-dividing the input list as to form a tree, at whose leaves we can apply the function f in parallel. In this case we can use a much simpler and more efficient version of the list-tree representation, because we are free to choose the order in which elements of the list appear in the tree. Rather than dividing the list into two halves by cutting it in the middle, we divide it into odd- and even-indexed sublists:

```
ListToTree2 :: [α] → BinaryTree α

ListToTree2 [ ] = EMPTY
ListToTree2 [a] = LEAF a

ListToTree2 (a0:a1:as) = NODE (ListToTree2 (EvenOnes (a0:a1:as)))
                              (ListToTree2 (OddOnes (a0:a1:as))), if (a0:a1:as) ≠ [ ]

                         where
                         EvenOnes [ ] = [ ]
                         EvenOnes [a0] = [a0]
                         EvenOnes (a0 : a1 : as) = a0 : (EvenOnes as)

                         OddOnes [ ] = [ ]
                         OddOnes [a0] = [ ]
                         OddOnes (a0 : a1 : as) = a1 : (OddOnes as)
```

and

```
TreeToList2 :: BinaryTree α → [α]

TreeToList2 EMPTY = [ ]
TreeToList2 (LEAF a) = [a]

TreeToList2 (NODE evensubtree oddsubtree)
            = (merge (TreeToList2 evensubtree) (TreeToList2 oddsubtree))
              where
              merge (a0 : evens) (a1 : odds) = a0 : a1 : (merge evens odds)
              merge as [ ] = as
```

In Appendix A (Theorem 2), total structural induction is used to verify that for all finite

and total lists as,

```
TreeToList2 (ListToTree2 as) = as
```

(Notice the striking resemblance between the structure of this computation and the structure of QuickSort).

Now let us define a map operator for trees:

```
MapTree :: (α → β) → BinaryTree α → BinaryTree β

MapTree f EMPTY = EMPTY
MapTree f (LEAF a) = LEAF (f a)
MapTree f (NODE subtree1 subtree2) = NODE (MapTree subtree1)
                                          (MapTree f subtree2)
```

It is very easy to verify using equational reasoning that for all finite and total lists as,

```
map f as = TreeToList2 (MapTree f (ListToTree2 as))
```

We can simply substitute this implementation of map when required.

It is, however, far from clear that it will improve matters unless the process creation or migration overhead is very large. It does more overall work than the simpler definition, but the work is potentially more parallel and more distributed. If there is already more than enough parallelism on the machine, it will certainly slow the computation down.

4.3 Pipeline parallelism

When vertical parallelism is used and the parameter concerned is a list, pipeline parallelism can occur. For example, suppose we have the function definitions

```
from n = n : (from (n+1))
```

and

```
integrate as = 0 : (integrate' 0 as)
               where
               integrate' sum [ ] = [ ]
               integrate' sum (a : as) = newsum : (integrate' newsum as)
                                         where
                                         newsum = a + sum
```

Provided we have sufficient strictness information, vertical parallelism is available in the application

Figure 4.1: Pipelining and horizontal parallelism

```
integrate (from 1)
```

One processor can be responsible for executing from 1, while another is responsible for the application of integrate to the other's output.

We can easily extend the pipeline:

```
map ((×) 10) (integrate (from 1))
```

Pipelining combines naturally with horizontal parallelism:

```
map2 (+) (map ((×) 2) (from 1))
         (map ((×) 3) (from 1))
```

This demands a diagram, given in figure 4.1. This resembles the graph representation of an expression like $(2 \times x) + (3 \times x)$, but now the nodes represent processes which can exist for a substantial period of time, operating on successive input values. Such a diagram is often called a *data flow graph*, since one could imagine a real, parallel computer built from units (represented as nodes in the graph) wired together according to the arcs given. During a computation, data would flow along the arcs and no other communications would be necessary. This idea has prompted a large variety of computer architectures based on the data flow idea, including for example, the Manchester data flow machine [GKW85], the MIT Tagged-Token data flow architecture [AN87] and many others. It must be emphasised that these computers are not rewired for each dataflow program, but rather exploit a dataflow graph program representation at run-time. We call such diagrams *process networks* in this book to emphasise that special dataflow hardware need not be involved, and that, as we shall see in Chapter 5, there may indeed be a static allocation of processes to processing elements.

4.3.1 Cyclic process networks

The example of Figure 4.1 is acyclic, but there is no reason why a cycle should not be introduced. Cycles in process networks correspond to iteration, and we can derive a cyclic process network definition from the recurrence idiom introduced in Chapter 2. Two

examples will be demonstrated: the Fibonacci numbers and the Newton-Raphson method.

Recall the definition of the list of Fibonacci numbers:

```
fibs = generate NextFib
       where
       NextFib 0 = 1
       NextFib 1 = 1
       NextFib n = (fibs sub (n−1)) + (fibs sub (n−2)),          if n ≥ 2
```

Note that for $n \geq 2$,

```
NextFib n = prevfib + pprevfib,          if n ≥ 2
            where
            prevfib = fibs sub (n−1)
            pprevfib = fibs sub (n−2)
```

That is,

```
NextFib n = ((+) ∘∘ (((sub) fibs) ∘ (subtract 1))
                    (((sub) fibs) ∘ (subtract 2)) ) n,          if n ≥ 2
```

Now in the definition of fibs, unfold generate:

```
fibs = map NextFib (from 0)
     = map NextFib (0:1:(from 2))
     = (NextFib 0):(NextFib 1):(map NextFib (from 2))
     = 1:1:(map NextFib (from 2))
     = 1:1:(map ((+) ∘∘ (((sub) fibs) ∘ (subtract 1))
                        (((sub) fibs) ∘ (subtract 2)) ) (from 2))
```

Here we use the properties that map (f ∘ g) = (map f) ∘ (map g), and map (f ∘∘ g h) = (map2 f) ∘∘ (map g)(map h):

```
fibs = 1:1:( (map2 (+)) ∘∘ (map (((sub) fibs) ∘ (subtract 1)))
                           (map (((sub) fibs) ∘ (subtract 2))) ) (from 2)
     = 1:1:( (map2 (+)) ∘∘ (map (((sub) fibs) ∘ (subtract 1)))
                           ( (map ((sub) fibs)) ∘ (map (subtract 2)) ) ) (from 2)
     = 1:1:(map2 (+) (map ((sub) fibs) (map (subtract 1) (from 2)))
                     (map ((sub) fibs) (map (subtract 2) (from 2))) )
```

Clearly map (subtract n) (from m) = from (m−n), and that map ((sub) as) (from 0) = as, so we have

Figure 4.2: A cyclic process network to calculate the Fibonacci numbers

$$
\begin{aligned}
\mathsf{fibs} &= 1{:}1{:}(\mathsf{map2}\ (+)\ \underbrace{(\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{fibs})\ (\mathsf{from}\ 1))}\\
&\qquad\qquad\qquad\qquad\underbrace{(\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{fibs})\ (\mathsf{from}\ 0))}\\
&= 1{:}1{:}(\mathsf{map2}\ (+)\ (\mathsf{tl}\ \mathsf{fibs})\\
&\qquad\qquad\qquad\qquad \mathsf{fibs})
\end{aligned}
$$

This is the complete process network formulation, and was used as an example in section 2.4.3, where its operation is explained. Its process network is given in Figure 4.2.

For the second example let us take for an example the generalised Newton-Raphson method. We solve for $\mathsf{f}\ \mathsf{x} = 0$ with $\mathsf{f}'\ \mathsf{x} = \frac{d(\mathsf{f}\ \mathsf{x})}{dx}$, and using an initial estimate x_0:

$$
\begin{aligned}
&\mathsf{xs}\ \underline{\mathbf{sub}}\ 0 = \mathsf{x}_0\\
&\mathsf{xs}\ \underline{\mathbf{sub}}\ \mathsf{i} = (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1)) - (\ \mathsf{f}\ (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1))\\
&\qquad\qquad\qquad\qquad\qquad\qquad / \ \mathsf{f}'\ (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1))\ ), \quad \mathbf{if}\ \mathsf{n} \geq 1
\end{aligned}
$$

with the implementation using the recurrence idiom:

$$
\begin{aligned}
&\mathsf{solve}\ \mathsf{f}\ \mathsf{f}'\ \mathsf{x}_0\\
&= \mathsf{until}\ \mathsf{converges}\ \mathsf{xs}\\
&\quad \mathbf{where}\\
&\quad \mathsf{converges}\ 0 = \mathsf{FALSE}\\
&\quad \mathsf{converges}\ \mathsf{i} = \mathsf{abs}(((\mathsf{xs}\ \underline{\mathbf{sub}}\ \mathsf{i})\\
&\qquad\qquad - (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1)))/(\mathsf{xs}\ \underline{\mathbf{sub}}\ \mathsf{i})) \leq \epsilon, \quad \mathbf{if}\ \mathsf{i} \geq 1\\
&\quad \mathsf{xs} = \mathsf{generate}\ \mathsf{NextEstimate}\\
&\qquad\quad \mathbf{where}\\
&\qquad\quad \mathsf{NextEstimate}\ 0 = \mathsf{x}_0\\
&\qquad\quad \mathsf{NextEstimate}\ \mathsf{i} = (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1))\\
&\qquad\qquad\qquad\qquad - (\ \mathsf{f}\ (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1))\ /\ \mathsf{f}'\ (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i}{-}1))\ ), \quad \mathbf{if}\ \mathsf{n} \geq 1
\end{aligned}
$$

The derivation of the process network formulation of this definition is given in Appendix A,

section A.3. In simplified form it is:

```
solve f f' x0
= select (map2 Test (tl xs) xs) (tl xs)
  where
  xs = x0 : (map Transition xs)
  Test prevx thisx = abs( (thisx − prevx)/thisx ) ≤ ε
  Transition prevx = prevx − ((f prevx)/(f' prevx))
```

We can introduce parallelism into this definition by separating the arithmetic operations into processes. This is done by propagating map into the bodies of the arithmetic expressions. To do this, Map2Test is defined to be the transformed version of map2 Test, and MapTransition is defined to be the transformed version of Map Transition:

```
solve f f' x0
= select (Map2Test (tl xs) xs) (tl xs)
  where
  xs = x0 : (MapTransition xs)
  Map2Test thisxs nextxs = map ((≥) ε)
                               (map abs ( (map2 (/) (map2 (−) thisxs prevxs)
                                                    thisxs)))
  MapTransition prevxs = map2 (−) prevxs (map2 (/) (map f prevxs)
                                                   (map f' prevxs))
```

The graphical representation of this network is given in Figure 4.3. This example clearly has some potential for parallelism in the evaluation of f x_i and f' x_i.

When the value at each step (x_i here) is a vector or matrix rather than a scalar, additional parallelism is available by pipelining successive iterations.

4.4 The Kahn principle

The relationship between the diagrams and the programs they are supposed to represent is made precise by what is sometimes called the *Kahn Principle*. Gilles Kahn, in a classic paper [Kah74], showed how, if we are given a functional specification of the behaviour of each process in a network, we can write down the behaviour of the network as a whole. First, label every arc of the network with a separate variable name. Then write down an equation for each variable, defining its value in terms of constants and other variables.

For example, Figure 4.4 shows the Newton-Raphson process network with each arc labelled with a new variable. Each node is labelled with a functional description of its behaviour. The Kahn principle says we can determine the behaviour of the network as a whole by writing down the system of equations relating the variables:

Figure 4.3: A cyclic process network applying the Newton Raphson method

Figure 4.4: A cyclic network with labelled arcs

```
a = select bs cs
bs = map ((≥) ε) ds
cs = tl es
ds = map abs fs
es = x0 : gs
fs = map2 (/) hs cs
gs = map2 (−) es is
hs = map2 (−) cs es
is = map2 (/) js ks
js = map f es
ks = map f' es
```

It is not hard to verify that this definition is equivalent to the definition of solve f f' x_0 given earlier.

Using this relationship, it is possible to visualise a large and useful class of functional programs as process networks, and this has been the starting point for several dataflow programming languages, most notably Lucid [WA85].

4.5 Parameter-dependent process networks

The process networks we have seen so far have been *static*: their size and shape has been independent of the program's parameters. This need not always be so. In some cases the process network depends on something simple like the length of some parameter list, as in the next example, although in general the dependency can in principle be arbitrarily complicated.

For a simple example, suppose we build a pipeline by composing three functions, map f_1, map f_2 and map f_3:

```
pipeline [f1, f2, f3] xs = map f1 (map f2 (map f3 xs))
```

This can be rewritten using "∘":

```
pipeline [f1, f2, f3] xs = ((map f1) ∘ (map f2) ∘ (map f3)) xs
```

When we don't know how many f_i's there are, we can use the insert function with "∘", together with the identity function ident x = x:

```
pipeline [f1, f2, f3] xs = (insert (∘) ident [(map f1), (map f2), (map f3)]) xs
```

so that

```
pipeline fs xs = (insert (∘) ident (map map fs)) xs
```

This captures a pipeline of processes as long as the list fs.

Now suppose that the functions f_i are not chosen arbitrarily, but are instances of a general function f, specialised by partial, curried, application to successive elements of the

list, say $[a_1, a_2, a_3]$, i.e.

$$[f_1, f_2, f_3] = \text{map } f\ [a_1, a_2, a_3]$$

or

$$fs = \text{map } f\ as$$

Now the pipeline is

$$\text{pipeline (map fs as) xs} = \text{(insert } (\circ) \text{ ident (map map (map f as))) xs}$$

This definition is rather hair-raising, what with insert $(\circ)$ ident and map map (map f) as, but what these forms actually do is quite down-to-earth. They appear in concrete form in an example drawn from the ray-tracing algorithm for three-dimensional image rendering.

4.5.1 Example: ray intersection test

We have a list of rays and a list of objects, and we need to find which object each ray strikes first. To avoid technicalities, let us assume suitable definitions for a data type Ray to represent a ray, giving its direction and starting point, a data type Object, describing perhaps a sphere, plane or just a polygonal facet, and a function

```
TestForImpact :: Ray → Object → Impact
```

where Impact is a data type which describes the interaction between the ray and the object. This may be a miss NOIMPACT, or a hit IMPACT, with details of how far along the ray the impact occurs (needed to find the ray's first impact), and other information relating to the angle of impact, the surface texture, refraction etc. which need not be specified here:

```
Impact ::= NOIMPACT |
           IMPACT Num ImpactInformation
```

Now what we need to find is the first object struck by the ray in the list of all objects of interest:

```
FirstImpact :: [Object] → Ray → Impact

FirstImpact objects ray = earliest (map (TestForImpact ray) objects)
                          where
                          earliest impacts = insert earlier NOIMPACT impacts
```

The function earlier compares two impacts, and returns the one which occured earlier in the ray's travel—i.e. the one the ray actually hits. It must take account of NOIMPACT properly:

Figure 4.5: The untransformed parallel ray intersection test

```
earlier :: Impact → Impact → Impact

earlier NOIMPACT NOIMPACT = NOIMPACT
earlier (IMPACT distance1 info1) NOIMPACT = (IMPACT distance1 info1)
earlier NOIMPACT (IMPACT distance2 info2) = (IMPACT distance2 info2)
earlier (IMPACT distance1 info1)
        (IMPACT distance2 info2) = (IMPACT distance1 info1), if distance1 ≤ distance2
earlier (IMPACT distance1 info1)
        (IMPACT distance2 info2) = (IMPACT distance2 info2), if distance1 > distance2
```

The complete definition to find the impacts corresponding to a list of rays is now

```
FindImpacts :: [Ray] → [Object] → [Impacts]

FindImpacts rays objects = map (FirstImpact objects) rays
```

This definition has the potential for very highly-parallel evaluation, arising from horizontal parallel evaluation of each FirstImpact objects ray_i expression. Figure 4.5 shows the pattern of data dependency in this computation.

Introducing pipeline parallelism

It is possible to make this algorithm more suitable for loosely-coupled parallel processors by transforming it to increase its locality.

The pipelined implementation consists of a chain of pipeline stages, PipelineStage. Each looks after its own object. PipelineStage object takes as input and produces as output a stream of PipeItems:

```
PipeItem α β ::= PIPEITEM α β
```

A object of the type PipeItem ray impact contains a ray and its earliest impact so far. The pipeline stage function PipelineStage tests the ray against the stage's object, and then compares the resulting impact with impact. Its output is a copy of the input ray, together with the earlier impact:

```
PipelineStage :: Object → PipeItem Ray Impact → PipeItem Ray Impact

PipelineStage object (PIPEITEM ray impact)
                     = PIPEITEM ray impact'
                       where
                       impact' = earlier impact NewImpact
                       NewImpact = TestForImpact ray object
```

Now it should be clear that we can write the definition of FirstImpact as

```
FirstImpact [object1, object2, ... objectN] ray
= impact
  where
  PIPEITEM ray impact
   = PipelineStage object1
       (PipelineStage object2
         ··· (PipelineStage objectN (ray, NOIMPACT)) ··· )

   = ((PipelineStage object1) ∘
      (PipelineStage object2) ∘
      ··· ∘ (PipelineStage objectN)) PIPEITEM ray NOIMPACT
```

This definition can be tidied somewhat using functions to build the PipeItem structure at the input to the pipeline, and to select out the impact at the output:

```
MakePipeItem ray = PIPEITEM ray NOIMPACT
TakeImpact (PIPEITEM ray impact) = impact
```

giving us

```
FirstImpact [object1, object2, ... objectN] ray
= (TakeImpact ∘
  ((PipelineStage object1) ∘
   (PipelineStage object2) ∘
   ··· ∘ (PipelineStage objectN))
  ∘ MakePipeItem)
  ray
```

We can remove the "···" notation using insert (∘) ident and map:

Figure 4.6: The transformed, pipeline-parallel ray intersection test

```
FirstImpact objects ray
= (TakeImpact ∘
   (insert (∘) ident (map PipelineStage objects))
   ∘ MakePipeItem)
   ray
```

Finally, recall that

```
FindImpacts rays objects = map (FirstImpact objects) rays
                         = map ( (TakeImpact ∘
                                  (insert (∘) ident
                                          (map PipelineStage objects))
                                  ∘ MakePipeItem )
                           rays
```

Now propagate the map into the composition using map (f ∘ g) = (map f) ∘ (map g):

```
FindImpacts rays objects = ( (map TakeImpact) ∘
                             (insert (∘) ident
                                     (map map (map PipelineStage objects)))
                             ∘ (map MakePipeItem) )
                             rays
```

This transformation is justified more formally in Appendix A, section A.4. The transformed version's process network is shown in Figure 4.6. The important difference between the transformed and untransformed algorithms is that in figure 4.5 the graph's connectivity is very high, since every intersection test process requires access to the entire objects list.

By contrast, in the pipeline version, figure 4.6, the graph's connectivity is very low. It is a much more distributed algorithm, more suitable for a distributed memory, loosely-coupled multiprocessor. We return to this point in the next chapter, section 5.1.2.

4.6 Infinite process networks

We have seen process networks whose size depends on the size of some data structure. In our functional language data structures need not be finite in size—a list might grow indefinitely, and the same can happen with a process network, as happens in the next example.

4.6.1 Example: generating primes using Eratosthenes' sieve

The infinite list of prime numbers can be computed as follows:

```
primes = sieve (from 2)
         where
         sieve (a : as) = a : (sieve (FilterMultiples a as))
```

where

```
FilterMultiples p (a:as) = a : (FilterMultiples p as),   if not(divides p a)
                         =     FilterMultiples p as,     if divides p a
```

This neat (although hardly clear) example[2] computes the list of all the prime numbers, using Eratosthenes' famous sieve algorithm. An explanation and derivation of this formulation of the algorithm is given in Appendix A, section A.5.

Its process network is given in Figure 4.7. At each invocation of sieve, a new instance of FilterMultiples is generated. Thus the process network has the form of a chain, which is constantly being extended as more primes are found.

The primes sieve is not a good parallel algorithm, because most of the work is done by a small number of the FilterMultiples processes. In fact infinite process networks don't seem very useful for parallel programming in general. As well as this balance problem, the processes must be mapped to processors at run time. The example is given mainly to demonstrate the potential existence of such programs.

4.7 Process networks as hardware descriptions

When a process network is finite in size, it is very natural to interpret it as a description of a physical, parallel, computer. A restricted form of our functional language can be used as a hardware description language, and we could use it to specify the design of VLSI devices, as in the example given here.

[2] Attributed to P. Quarendon by Henderson and Morris [HM76].

Figure 4.7: Some steps in the evaluation of the primes sieve

4.7.1 Primitives for hardware description

In specifying a digital electronic circuit, the fundamental data type is the (approximate) voltage on a wire at a particular time. We consider here just three possibilities: a settled high or low logic level, or some temporary intermediate value "XX":

Sample ::= HI | LO | XX

We will model the behaviour of a wire by the sequence of voltage samples taken at regular intervals indefinitely:

Signal == [Sample]

We will bundle wires using the tuple "$(\cdots)$" notation, and form indexed aggregates (to represent numbers, for example), using vectors:

bus == <Signal>

The approach is to use the functional notation to connect simple combinational circuits in a restricted set of ways.

Specifying combinational logic using truth tables

To do this we need a primitive for implementing circuits specified using just truth tables:

```
SignalCase :: [Signal] → ([Sample],[Sample]) → [Signal]

SignalCase inputsignal cases
 = (transpose ∘
    ∘ (map (SelectMatch cases))
    ∘ transpose) inputsignal
```

where

```
SelectMatch :: ([Sample],[Sample]) → [Sample] → [Sample]

SelectMatch ((lhs, rhs) : cases) samples = rhs,                          if lhs = samples
SelectMatch ((lhs, rhs) : cases) samples = SelectMatch cases samples,    otherwise
```

and

```
transpose :: [[α]] → [[α]]

transpose rows = [ ],                                          if rows = [ ]
transpose rows = (map hd rows) : (transpose (map tl rows))     otherwise
```

Transposition is specified by the requirement that for all n and m,

```
(rows sub n) sub m = (cols sub m) sub n
                     where
                     cols = transpose rows
```

It is used here to transform an finite list of signals into an infinitely-long list of samples. For example, a 3-element list of signals is transformed into a stream of three-element lists of samples:

$$\begin{array}{l}\mathsf{transpose}\ [[a_1,\ a_2,\ \ldots]\ \ [b_1,\ b_2,\ \ldots]\ \ [c_1,\ c_2,\ \ldots]] = \\ \qquad\qquad [[a_1,\ b_1,\ c_1],\\ \qquad\qquad\ [a_2,\ b_2,\ c_2],\\ \qquad\qquad\ [a_3,\ b_3,\ c_3],\\ \qquad\qquad\ \vdots\end{array}$$

In this form map SelectMatch can be used to apply the transformation specified by the truth table, to produce a stream of lists of samples as output. This is turned back into a list of signals by applying transpose again.

Using SignalCase it is easy to define the building blocks for more complicated circuits. For example, the "or" operation is defined by

```
IdealOr :: (Signal, Signal) → Signal

IdealOr a b = hd (SignalCase [a,b]
                             [[[LO, LO], [LO]],
                              [[LO, HI], [HI]],
                              [[HI, LO], [HI]],
                              [[HI, HI], [HI]],
                              [[XX, HI], [XX]],
                              [[HI, XX], [XX]],
                              [[XX, LO], [XX]]
                              [[LO, XX], [XX]]] )
```

Many combinational circuits are undefined if any input is undefined, so it might prove helpful to build this into SignalCase. It is left out here for clarity.

Modeling propagation delay

Notice the use of "don't know" values in the truth table, so that the gate propagates undefined results properly (this could be done automatically by SignalCase. Undefined results come into the world when the machine is switched on, and filter through the circuit at a rate determined by propagation delays. Thus, a more realistic or-gate would be

```
Or (a, b) = Delay τ (IdealOr (a, b))
```

where τ is the number of samples-worth of delay incurred. This is implemented by simply prepending τ undefined samples to the gate's output:

```
Delay t signal = (UndefinedFor τ) ++ signal
                 where
                 UndefinedFor τ = replicate τ XX
```

Thus, the or-gate's first τ outputs are undefined even if its input is defined. Thereafter, changes to the input are delayed τ time units in the output. This is not the only physically sensible model of delay.

Latches and registers

A register is the primitive storage element in a digital circuit. The behaviour of a simple register is specified by the equations

```
Register :: Sample → Signal → Signal → Signal
```

```
Register initialstate (si : input)
                      (XX : strobe) = initialstate : (Register initialstate input strobe)
Register initialstate (si : input)
                      (LO : strobe) = initialstate : (Register initialstate input strobe)
Register initialstate (si : input)
                      (HI : strobe) = initialstate : (Register si input strobe)
```

The register maintains and outputs an internal state, which retains the last value of the input when the strobe is HI. It is not hard to verify (using the alternative definitions of iterate) that this recursive definition is equivalent to

```
Register initialstate input strobe
 = output
   where
   output = initialstate : FeedbackFunction [output, input, strobe]
```

where

```
FeedbackFunction [op, ip, st] = hd (SignalCase [op, ip, st]
                                                [[[HI, HI, HI], [HI]],
                                                 [[LO, HI, HI], [HI]],
                                                 [[HI, LO, HI], [LO]],
                                                 [[LO, LO, HI], [LO]],
                                                 [[HI, HI, LO], [HI]],
                                                 [[HI, LO, LO], [HI]],
                                                 [[LO, LO, LO], [LO]],
                                                 [[LO, HI, LO], [LO]]])
```

This definition of Register specifies a combinational circuit and a wiring diagram, and so doubles as both a behavioural and a structural description of a circuit. The feedback, in the form of a recursive stream definition, is a necessary feature of any history-sensitive circuit description.

4.7.2 Example: Adder

A binary adder circuit would normally be provided to the VLSI designer as a carefully hand-crafted library "cell". However, we can specify it quite naturally. Following the standard digital circuit design textbooks, we start by building a bit-wise adder, from two "half-adders":

```
HalfAdder :: (Signal, Signal) → (Signal, Signal)
```

```
HalfAdder (a, b) = SignalCase [Delay τ a, Delay τ b]
                              [[[LO, LO], (LO, LO)],
                               [[LO, HI], (HI, LO)],
                               [[HI, LO], (HI, LO)],
                               [[HI, HI], (HI, HI)],
                               [[XX, HI], (XX, XX)],
                               [[XX, LO], (XX, XX)],
                               [[HI, XX], (XX, XX)],
                               [[LO, XX], (XX, XX)]]
```

Now we can put together a full adder taking two numbers and a carry from the preceding digit, and producing this digit pair's sum, and a carry for the next digit:

```
FullAdder :: (Signal, Signal, Signal) → (Signal, Signal)

FullAdder (a, b, CarryIn) = (sum2, CarryOut)
                            where
                            (sum2, Carry2) = HalfAdder (sum1, CarryIn)
                            CarryOut = Or (Carry1, Carry2)
                            (sum1, Carry1) = HalfAdder (a, b)
```

For convenience we will define projectors to pick out the sum and the carry:

```
SumOf (sum, carry) = sum
CarryOf (sum, carry) = carry
```

We construct the complete, multi-digit adder by writing down the standard addition algorithm as a vector recurrence:

```
BitwiseAdder :: Num → (Signal, bus, bus) → (bus, Signal)

BitwiseAdder BusSize (CarryIn, aBus, bBus)
 = (ResultBus, CarryOut)

   where
```

```
ResultBus = MakeVector BusSize Sums
            where
            Sums n = SumOf (AdderOutputs sub n)

CarryOut = CarryBus sub BusSize

AdderOutputs = MakeVector BusSize FullAdders
               where
               FullAdders n = FullAdder ((aBus sub n),
                                         (bBus sub n),
                                         (CarryBus sub n))
CarryBus = MakeVector (BusSize+1) Carries
           where
           Carries 0 = CarryIn
           Carries (n+1) = CarryOf (AdderOutputs sub n)
```

(where BusSize is the length (VectorBound) of aBus and bBus. This algorithm is entirely sequential and so rather slow, although it can be implemented with very little hardware. More realistic adders use a "look-ahead" carry scheme which breaks the chain of dependency between successive digits. This is left as an exercise for the interested reader.

4.7.3 Functional hardware description languages

The ease with which digital hardware can be specified in the functional notation has led to several commercial silicon design systems based on the functional approach, notably ELLA [MPT85]. It has even been claimed (by Johnson [Joh84a]) that the way a system is described functionally coincides precisely with the abstraction imposed by the digital view of circuit design.

Notice the distinction between functions which correspond to active circuitry (ultimately using the SignalCase construct), and functions which arrange wiring only—such as MakeVector, SumOf, sub etc. It was clear from the way we used the notation that these "wiring" functions are scaffolding which is not supposed to be present in the resulting circuit. We can remove the scaffolding using reduction, as soon as we know the size of the input buses. For example, a 3-bit adder circuit is specified by

```
BitwiseAdder 3 (CarryIn, <a0, a1, a2>, <b0, b1, b2>)
 = (ResultBus, CarryOut)


   where
```

```
ResultBus = MakeVector 3 Sums
            where
            Sums n = SumOf (AdderOutputs sub n)

CarryOut = CarryBus sub 2

AdderOutputs = MakeVector 3 FullAdders
               where
               FullAdders n = FullAdder ((<a0, a1, a2> sub n),
                                         (<b0, b1, b2> sub n),
                                         (CarryBus sub n))
CarryBus = MakeVector 3 Carries
           where
           Carries 0 = CarryIn
           Carries (n+1) = CarryOf (AdderOutputs sub n)
```

This reduces to

```
BitwiseAdder 3 (CarryIn, <a0, a1, a2>, <b0, b1, b2>)
 = (ResultBus, CarryOut)

   where

   ResultBus = <SumOf( FullAdder (a0, b0, CarryBus sub 0) ),
                SumOf (FullAdder (a1, b1, CarryBus sub 1)),
                SumOf (FullAdder (a2, b2, CarryBus sub 2)) >

   CarryOut = CarryBus sub 2

   AdderOutputs = <FullAdder (a0, b0, CarryBus sub 0),
                   FullAdder (a1, b1, CarryBus sub 1),
                   FullAdder (a2, b2, CarryBus sub 2) >

   CarryBus = <CarryIn,
               CarryOf (FullAdder (a, b0, CarryBus sub 0)),
               CarryOf (FullAdder (a1, b1, CarryBus sub 1)) >
```

This circuit this describes is illustrated in figure 4.8.

This use of symbolic evaluation in order to produce a static process network from an abstract description was seen in the ray-tracer pipeline example. In the hardware description context, it is useful to be able to distinguish "static" components from the "dynamic" parts of the program which serve only to capture the wiring in an abstract way. In some functional hardware description languages (e.g. Johnson's DAISY language

Figure 4.8: A three-bit adder circuit

[Joh84a], the language is syntactically separated to make the distinction especially clear.

4.8 Divide-and-conquer using a process network

This chapter began by distinguishing two different forms in which parallelism can appear in functional programs, being in essence "divide-and-conquer" and the pipeline/parallel process network. To complete this chapter's brief overview of transformation techniques, we present a transformation which can turn the divide phase of a recursive divide-and-conquer parallel program into a cyclic process network. For motivation, it will be used in a very simple ray-tracer, where the process network formulation allows a highly parallel pipeline algorithm. The transformation is quite complex and detailed, and is not vital to the remainder of the book, so the reader is invited to skip to the end of this chapter, pausing only to look at the introductory material here and in the next subsection.

The untransformed version

Recall the generic divide-and-conquer formulation (we simplify it by representing the trivial case by a decomposition with no subproblems):

$$\mathsf{DivideAndConquer} :: (\alpha \to [\beta] \to \beta) \to (\alpha \to [\alpha]) \to \alpha \to \beta$$

```
DivideAndConquer CombineSolutions Decompose problem
 = Solve problem
   where
   Solve problem = CombineSolutions problem (map Solve SubProblems)
                   where
                   SubProblems = Decompose problem
```

The transformed version

The final, transformed version has the form

```
DivideAndConquer CombineSolutions Decompose problem
 = EvaluateTree (StreamToMTree (StreamOfSubProblemTree [problem]))

   where

   StreamOfSubProblemTree problems
    = output
      where
      (output, feedback)
       = SplitStream
          DividePhase (problems++feedback)

   DividePhase NewAndRecycledProblems
    = insert (++) [ ]
             (map LayerOf NewAndRecycledProblems)

   LayerOf problem
    = (OUTPUTTAG (MTREETOKEN problem CombineSolutions NoOfSubproblems))
       : (map FEEDBACKTAG Subproblems)
      where
      Subproblems = Decompose problem
      NoOfSubproblems = length Subproblems
```

The process network of the cyclic pipeline is given in Figure 4.9. This definition makes use of several functions whose definitions have will be given shortly. They are collected in Appendix A, section A.6, where a proof of correctness of the transformation is given.

4.8.1 Operation of the cyclic divide-and-conquer program

To understand figure 4.9, it is necessary to think of the tree of subproblem decomposition as an explicit data structure. We have a "divide" phase where the tree is constructed, and

Figure 4.9: Sketch of the cyclic pipeline formulation of divide-and-conquer

a "conquer" phase, where solutions to subproblems are propagated up the tree from its leaves, until a solution to the root problem can be found.

The cyclic part of the transformed program appears in the divide phase. The tree is constructed generation by generation in a breadth-first manner. This means problem decomposition can be performed on all the elements of a generation at once.

The first generation is the input problem by itself. When the divide phase is applied to this, one node of the decomposition tree is built and a number of subproblems are generated. The node is tagged using OUTPUTTAG and is passed through SplitStream to output where it is collected (StreamToMTree) to form the decomposition tree. The subproblems are tagged using FEEDBACKTAG, and are passed by SplitStream to feedback. They are finally fed back to succeed input in the input stream to DividePhase.

The process comes to an end when subproblems can be decomposed no further. When a whole generation of subproblem-less subproblems is reached, the complete decomposition tree can be completed, and the conquer phase can begin.

4.8.2 Derivation of the cyclic divide-and-conquer program

The derivation consists of the following steps:

- Separating the two phases, divide and conquer, linked by the decomposition tree,
- Transforming the tree into a stream, scanned in breadth-first order—and back again,
- Integrating the tree construction process with transformation of the tree into a breadth-first scan,
- Introduction of a cyclic stream version of the integrated tree construction and scan process.

Separating divide from conquer with a decomposition tree

The intermediate tree has the type

MultiTree α β ::= MNODE α ($\alpha \rightarrow [\alpha] \rightarrow \beta$) Num [MultiTree α β]

Although there is only one kind of element in this type, it is tagged with the constructor MNODE for clarity.

Each node of a MultiTree,

MNODE Problem CombiningFunction NoOfSubproblems Subproblems

consists of a list of subtrees Subproblems, a number NoOfSubproblems giving the number of subtrees in the list, CombiningFunction, a function to take solutions of the subtrees' problems, and produce a solution of the problem the node itself represents, and Problem, the original problem to be solved. At the leaves, the list of subtrees is empty. Using MultiTree, we have the new functions

```
BuildTree :: α → MultiTree α β
EvaluateTree :: (MultiTree α β) → β
```

which make the intermediate tree explicit when put together:

```
DivideAndConquer CombineSolutions Decompose problem
 = EvaluateTree (BuildTree problem)

   where

   BuildTree problem = MNODE problem
                               CombineSolutions
                               NoOfSubproblems
                               (map BuildTree Subproblems)
                       where
                       Subproblems = Decompose problem
                       NoOfSubproblems = length SubProblems
```

where

```
EvaluateTree (MNODE problem CombineSolutions n subtrees)
             = CombineSolutions problem (map EvaluateTree subtrees)
```

This is proven in Appendix A section A.6.1, but note that it holds only if CombineSolutions is strict in all the sub-solutions.

Separating the phases by a stream

The next stage of the derivation is to flatten the intermediate tree structure into a stream (i.e. a lazily-produced list). To do this a special tree–stream data type transformation is used. It is necessary to scan the tree *breadth first* in order to bring out its parallelism. We use a list of special tokens,

```
MultiTreeToken α β ::= MTREETOKEN α (α → [α] → β) Num
```

(the number carries the number of subtrees for this node, and is necessary to enable the tree to be reconstructed from the stream). To perform the transformation we need two functions,

```
MTreeToStream :: MultiTree α β → MultiTreeToken α β
StreamToMTree :: MultiTreeToken α β → MultiTree α β
```

For all (finitely-branching) MultiTrees mtree we require that

```
StreamToMTree (MTreeToStream mtree) = mtree
```

(The functions StreamToMTree and MTreeToStream are defined in Appendix A section A.6.2, where they are derived. Now we can rewrite the two-phase divide-and-conquer formulation as

```
DivideAndConquer CombineSolutions Decompose problem
 = EvaluateTree (StreamToMTree (MTreeToStream (BuildTree problem)))
   where
   BuildTree problem = ···
```

Simplifying tree construction

In the next step of the transformation, we have the composition

```
MTreeToStream (BuildTree problem)
where
BuildTree problem = MNODE problem
                          CombineSolutions
                          NoOfSubproblems
                          (map BuildTree Subproblems)
                    where
                    Subproblems = Decompose problem
                    NoOfSubproblems = length SubProblems
```

where

```
MTreeToStream tree = ListOfTreesToStream [tree] [ ]

ListOfTreesToStream [ ] [ ] = [ ]

ListOfTreesToStream [ ] children = ListOfTreesToStream children [ ]

ListOfTreesToStream ((MNODE p op n newchildren) : siblings) oldchildren
                    = (MTREETOKEN p op n)
                      : (ListOfTreesToStream siblings (oldchildren++newchildren))
```

We can use reduction to define a new function BuildStream which constructs the stream directly, so that the tree need not be built at all here. We proceed by writing down a specification for BuildStream, and then reducing:

```
BuildStream problem = MTreeToStream (BuildTree problem)
                    = ListOfTreesToStream [BuildTree problem] [ ]
```

Clearly it is ListOfTreesToStream which does all the work, so define

```
BuildStream problem = BuildStreamsOfTrees [problem] [ ]
                      where
                      BuildStreamsOfTrees problems subproblems
                      = ListOfTreesToStream (map BuildTree problems) subproblems
```

This gives us a specification for BuildStreamsOfTrees. The equations defining BuildStreamsOfTrees directly are derived from those defining ListOfTreesToStream by instantiation and then reduction:

```
BuildStreamsOfTrees [ ] [ ] = [ ]

BuildStreamsOfTrees [ ] subproblems = BuildStreamsOfTrees subproblems [ ]

BuildStreamsOfTrees (problem : siblingproblems) oldsubproblems
                    = (MTREETOKEN problem CombineSolutions NoOfSubproblems)
                      : (BuildStreamsOfTrees siblingproblems
                                             (oldsubproblems++Subproblems))
                      where
                      Subproblems = Decompose problem
                      NoOfSubproblems = length Subproblems
```

Building the cycle

The next step is the only "eureka" step in the derivation, so-called because it is pulled out a hat and then verified rather than derived. It is not completely unexpected as we have

already seen several recursive definitions transformed into a similar form. One of the first examples was iterate, given in Chapter 2, section 2.2.7. The claim, proven in Appendix A, section A.6.3 (Theorem 5), is that an equivalent definition for BuildStreamsOfTrees is

```
BuildStreamsOfTrees [ ] [ ] = [ ]

BuildStreamsOfTrees problems subproblems
 = output
    where
    (output, feedback)
     = SplitStream
        ((map FEEDBACKTAG subproblems) ++
         (join (map LayerOf (problems++feedback))))


    LayerOf problem
     = (OUTPUTTAG (MTREETOKEN problem CombineSolutions NoOfSubproblems))
        : (map FEEDBACKTAG Subproblems)
       where
       Subproblems = Decompose problem
       NoOfSubproblems = length Subproblems
```

The undefined functions and data types are defined in a moment. Before explaining this, let's simplify things by presenting BuildStream itself in this form:

```
BuildStreamOfTrees problem
 = output
    where
    (output, feedback)
     = SplitStream
        (join (map LayerOf (problem : feedback)))


    LayerOf problem
     = (OUTPUTTAG (MTREETOKEN problem CombineSolutions NoOfSubproblems))
        : (map FEEDBACKTAG Subproblems)
       where
       Subproblems = Decompose problem
       NoOfSubproblems = length Subproblems
```

The function LayerOf takes a problem and produces a list containing the problem's node (tagged with OUTPUTTAG), followed by all the problem's subproblems (tagged with FEEDBACKTAG). The function SplitStream picks out the objects in the list and dispatches those marked for output as the function's result, but routes the subproblems, tagged FEEDBACKTAG, back to be decomposed again by LayerOf.

The definitions are as follows: the data type for the tagged list is

```
TaggedStreamItem α β ::= OUTPUTTAG (MultiTreeToken α β)
                       | FEEDBACKTAG α
```

The selection function SplitStream is defined as

```
SplitStream :: [TaggedStreamItem α β] → ([MultiTreeToken α β], α)

SplitStream [ ] = ([ ], [ ])

SplitStream ((OUTPUTTAG token) : rest)
            = (token : rest1, rest2)
              where
              (rest1, rest2) = SplitStream rest

SplitStream ((FEEDBACKTAG subproblem) : rest)
            = (rest1, subproblem : rest2)
              where
              (rest1, rest2) = SplitStream rest
```

The function join flattens a list of lists into a list:

```
join :: [[α]] → [α]

join xss = insert (++) [ ] xss
```

4.9 Application to ray tracing

To finish the chapter, we apply this transformation to a simple recursive ray-tracing program. By transforming the divide-and-conquer formulation into a cyclic definition, large-scale pipeline parallelism in the ray-intersection test is uncovered.

4.9.1 An introduction to ray-tracing

A variety of high-quality computer graphics applications require the generation, from a computer model of a three-dimensional space, of a view which includes shadows, and also models refraction and reflection from shiny and non-shiny surfaces. This is in addition to the more standard requirements for hidden surface removal, perspective, depth-cueing, etc.

The only generally-applicable way of generating such images is by modeling the paths and intensity of rays of light as they are reflected, refracted etc. in the simulated region. The crucial observation behind ray-tracing is that only rays which pass through a pixel, and are incident on the viewer's eye (stereo vision is generally ignored), need be considered, and that these rays can be followed backwards to their source. The classic reference on ray-tracing is Whitted [Whi80].

Tracing rays backwards ensures that only rays of interest are considered. Forward ray tracing is infeasible because, when a curved wavefront is approximately represented by many rays, the unavoidable quantisation can be arbitrarily amplified by unfortunately-placed curved reflecting surfaces.

The system being considered consists of the viewer's eye, the mesh imposed by the pixel pattern of the display device, and a simulated region behind the display device. Our task is to render the surface of the display device with just the colours, brightnesses and hues of the light passing through it from the simulated region to the viewer's eye. For the sake of simplicity we shall refer to the appearance of a pixel as its colour.

The set of objects in the simulated region will be called the *object database*. Each object has a characteristic surface: given an incoming ray, this determines where the contributory rays come from, and how their intensities are combined to produce the outgoing ray.

In essence, the ray tracing algorithm is as follows:

1. For every pixel, compute the ray starting from the viewpoint, which passes through it.

2. For each of these rays, find the first object in the object database which the ray strikes.

3. When a ray strikes a surface, compute the ray's colour. If the surface is a light source in its own right (or is completely dark), this is trivial. If not, calculate which rays contribute to this ray's intensity, and ray-trace them in turn. The intensities of the subrays can then be combined to give the resulting ray's intensity, using a formula modeling the surface's characteristics.

Thus, for each original ray, a tree of sub-rays is constructed during a recursive, divide-and-conquer computation of each pixel's colour and intensity properties.

It must be emphasised before going on to the details that this example is for illustration only:

- For practical use, a far more subtle initial approach is always justified. This version tests every ray against every object; a smarter algorithm would partition the environment, so that one intersection test against a large "envelope" object (called a *bounding volume*) would determine whether tests on objects inside the envelope could possibly succeed.

- This approach applies parallelism to the "divide" phase of the algorithm, during which intersection tests are performed. It leaves the "conquer" phase, during which pixel intensities are actually calculated given their contributory ray trees, to be performed entirely sequentially. For simple models of surfaces' optical properties, the divide phase does dominate [Whi80], but not by a large factor.

4.9.2 A simple divide-and-conquer ray tracer

To begin with, we must generate the list of all the rays passing from the viewer's eye through each pixel of the display device, into the simulated region. Without going into

detail, we assume we have a function which does this, given details of the display mesh and the position of the viewer's eye:

```
GenerateInitialRays :: mesh → point → [Ray]
```

There is no need to define the types mesh, point or Ray here. Now the fundamental question is, given a ray and the object database, what colour should we paint the corresponding pixel? Let us introduce a function to answer this, which we will refine shortly:

```
FindRayColour :: ObjectDatabase → Ray → PixelColour
```

We have no need to refine the type PixelColour, but we do assume that the object database is represented by a list of objects:

```
ObjectDatabase == [Object]
```

At the first level of abstraction, the ray tracer as a whole is defined by

```
RayTracer :: ObjectDatabase → point → [PixelColour]

RayTracer objects viewpoint = map (FindRayColour objects)
                                  (GenerateInitialRays objects viewpoint)
```

The pixel colours are output in the order they were generated by GenerateInitialRays. The ray-tracing is done by the recursive function FindRayColour:

```
FindRayColour objects ray = SurfaceModelFunction ColoursOfSubrays
                            where
                            ColoursOfSubrays = map (FindRayColour objects) Subrays
                            Subrays = GetSubrays ImpactInfo
                            SurfaceModelFunction = GetSurfaceModel ImpactInfo
                            ImpactInfo = FirstImpact objects ray
```

This employs the function FirstImpact, which was introduced back in section 4.5.1:

```
FirstImpact :: ObjectDatabase → Ray → Impact
```

where the data type Impact was defined as

```
Impact ::= NOIMPACT |
           IMPACT Num ImpactInformation
```

Without elaborating fully the type ImpactInformation, we assume that the contributory rays can be found by

```
GetSubrays :: ImpactInformation → [Ray]
```

and that the function which combines the colours of these contributory rays according to the appropriate surface model can be found by

```
GetSurfaceModel :: ImpactInformation → ([PixelColour] → PixelColour)
```

The original ray is also available:

```
GetRay :: ImpactInformation → Ray
```

When a ray strikes an opaque, non-reflective surface, GetSubrays will return an empty list of contributory rays, while GetSurfaceModel will return a constant function giving the colour of the surface.

Expressing the ray-tracer using DivideAndConquer

The first step in the transformation will be to convert the divide-and-conquer formulation of FindRayColour into a cyclic stream definition, using the result of section 4.8. To use that result, we must first express FindRayColour in terms of the generic DivideAndConquer form:

```
FindRayColour objects ray
 = DivideAndConquer CombineSolutions Decompose ray
   where
   Decompose ray = Subrays (FirstImpact objects ray)
   CombineSolutions ray subraycolours = (GetSurfaceModel (FirstImpact objects ray))
                                        subraycolours
```

It should be noted here that this is a highly parallel algorithm: when the many pixels on a typical screen are taken into account there will be work for several million PEs. The aim of the transformation is to organise the avilable parallelism to take advantage of a loosely-coupled multiprocessor.

4.9.3 Transformation to a cyclic stream definition

Once we have the ray tracer expressed using DivideAndConquer, we can now apply the cyclic stream transformation, giving:

```
FindRayColour objects ray
 = EvaluateTree (StreamToMTree (StreamOfContributoryRayTrees [ray]))
   where
   StreamOfContributoryRayTrees rays
    = output
      where
      (output, feedback)
       = SplitStream (DividePhase (rays ++ feedback))


   DividePhase NewAndRecycledRays
    = (insert (++) [ ]
              (map LayerOf NewAndRecycledRays))


   LayerOf ray
    = (OUTPUTTAG (MTREETOKEN ray CombineSubrayColours NoOfSubRays))
       : (map FEEDBACKTAG subrays)
      where
      CombineSubrayColours = GetSurfaceModel impactinfo
      subrays = Subrays impactinfo
      impactinfo = FirstImpact objects ray
      NoOfSubRays = length subrays
```

We can tidy this up a little by separating out the intersection test:

```
FindRayColour objects ray
 = EvaluateTree (StreamToMTree (StreamOfContributoryRayTrees [ray]))
   where
   StreamOfContributoryRayTrees rays
    = output
      where
      (output, feedback)
       = SplitStream (DividePhase (FindImpacts (rays ++ feedback)))
```

```
    where
    DividePhase NewAndRecycledRaysImpacts
     = (insert (++) [ ]
               (map LayerOf' NewAndRecycledRaysImpacts))

    LayerOf' impactinfo
     = (OUTPUTTAG (MTREETOKEN ray CombineSubrayColours NoOfSubRays))
        : (map FEEDBACKTAG subrays)
       where
       CombineSubrayColours = GetSurfaceModel impactinfo
       subrays = Subrays impactinfo
       ray = GetRay impactinfo
       NoOfSubRays = length subrays
```

where

```
FindImpacts rays objects = map (FirstImpact objects) rays
```

4.9.4 Exploiting pipeline parallelism in the cycle

The next step is to employ our pipelined formulation of FindImpacts,

```
FindImpacts rays objects = ( (map TakeImpact) ∘
                           (insert (∘) ident
                                   (map map (map PipelineStage objects)))
                           ∘ (map MakePipeItem) )
                           rays
```

This exploits parallelism successfully provided sufficient rays are present in the feedback cycle (figure 4.9) to keep the pipeline components busy.

4.9.5 Using pixel-wise parallelism

At present we can exploit pipeline parallelism to speed the application of the intersection test to each successive generation of a single pixel's contributory-ray tree. The pipeline will often be idle at the top and the bottom of trees, and (as is often the case) when most of the trees are quite small. Fortunately we can use the pipeline to work on parts of different pixels' contributory-ray trees at the same time.

First, notice that although we have concentrated so far on FindRayColour, the function we really need to evaluate is

```
RayTracer objects viewpoint = map (FindRayColour objects)
                                  (GenerateInitialRays objects viewpoint)
```

Now recall (in fact from Appendix A, section A.6.2) that StreamToMTree was defined in

terms of a StreamToListOfMTrees:

```
StreamToMTree stream = StreamToListOfMTrees 1 stream
```

StreamToListOfMTrees n stream picks up a list of n trees from stream. In deriving StreamToMTree we proved that

```
StreamToListOfMTrees (length trees) (ListOfMTreesToStream trees) = trees
```

Thus, we can very naturally show that

```
map (FirstImpact objects) rays
 = (StreamToListOfMTrees (length initialrays)
                          (StreamOfContributoryRayTrees initialrays))
```

This gives us the final, cyclic, pipelined ray tracer implementation:

```
RayTracer objects viewpoint
 = map EvaluateTree
       (StreamToListOfMTrees (length initialrays)
                             (StreamOfContributoryRayTrees initialrays))
   where
   StreamOfContributoryRayTrees rays
    = output
      where
      (output, feedback)
       = (SplitStream
          ∘ join
          ∘ (map LayerOf')
          ∘ (map TakeImpact)
          ∘ (insert (∘) ident
                    (map map (map PipelineStage objects)))
          ∘ (map MakePipeItem))
         (rays ++ feedback)


   where
```

```
LayerOf' impactinfo
 = (OUTPUTTAG (MTREETOKEN ray CombineSubrayColours NoOfSubRays))
    : (map FEEDBACKTAG subrays)
   where
   CombineSubrayColours = GetSurfaceModel impactinfo
   subrays = Subrays impactinfo
   ray = GetRay impactinfo
   NoOfSubRays = length subrays
```

4.10 Conclusions

This formulation of the ray tracer completes the chapter's illustrative investigation of techniques for transforming programs to bring out parallelism in easily-exploited ways. We return to the example in the next chapter.

The main objective of developing some general methods has only been partially achieved. Some useful techniques have been presented, but not in their most general form, nor so that they might be automated directly. There is plenty of room for further work.

In particular, the transformation from divide-and-conquer into a cyclic stream formulation can be improved. The approach can be extended to include the "conquer" phase, and can be generalised to apply when CombineSolutions is not strict in all the subproblems. This is interesting because a recursive expression evaluator is of this form, and would lead to a structure much like a cyclic-pipeline data flow architecture such as the Manchester Data Flow Machine [GKW85].

To make the approach practical, a considerable level of support is necessary. The work shown was developed using pencil and paper, and so mistakes inevitably creep in. Building software to help is a considerable challenge: systems to *check* derivations and proofs have existed for some considerable time (notable are LCF [GMW79] and its derivatives), but building proofs with such systems is a slow and painstaking task—much slower than pencil and paper. What has not yet been achieved convincingly is computer-aided program transformation and verification which is actually quicker and easier than doing it by hand. A great deal of work is in progress.

4.11 Pointers into the literature

Parallelism in graph reduction

Despite the large amount of research and development work on implementing parallel graph reduction machines, there is very little published material describing how programs might be designed to exploit these designs' capabilities. Goldberg's thesis [Gol88] is the most comprehensive to date, although other studies are under preparation. Goldberg's results are interesting in particular because of the light they throw on the importance of shared memory hardware.

Divide and conquer

Divide-and-conquer has a very long history in algorithm design, and its extension to parallel algorithm design comes very naturally. Thus, much of the standard algorithm design literature serves as a good introduction (see for example Aho, Hopcroft and Ullman [AHU83]). Horowitz and Zorat survey general parallel divide-and-conquer algorithms [HZ83], while Stout gives a survey of divide-and-conquer image processing algorithms [Sto87]. Rayward-Smith and Clark develop a theory for scheduling divide-and-conquer algorithms [RSC88]. Hartel and Vree present an interesting approach to the efficient implementation of a class of divide-and-conquer programs on relatively loosely-coupled parallel graph reduction machines [HV87]. Vree applied program transformation to improve the grain size of a divide-and-conquer implementation of a simulation of tidal motion in the North Sea [Vre87].

Dataflow and pipeline parallelism

Pipelining as a technique in the architecture of conventional computers is the subject of a review article by Ramamoorthy and Li [RL77], and a book by Kogge [Kog81].

Our interpretation of functional programs as specifications of networks of processes is shared by dataflow proponents. Dataflow languages like VAL [McG82], *Id* [NPA86], Lucid [WA85] and SISAL [MSA+85] are essentially forms of pure functional languages, normally restricted at least to first-order procedures (i.e. no function values), and commonly augmented by "syntactic sugar" for recurrences and other forms convenient for scientific computation.

This sugaring can often be thought of as restricting an imperative language so that no variable or structure element may be assigned to more than once. This leads to a slight increase in expressive power, exploited in *Id Nouveau* [AE88], where, for example, an array can be passed to two functions which then interact by assigning to array elements. This capability, shared with committed-choice logic languages like PARLOG [Gre87] and STRAND@STRAND [AI 88], goes beyond what can naturally be expressed in the purely-functional language used in this book.

The dataflow concept has prompted several hardware design projects, aimed at using a dataflow graph representation of a program's consituent instructions at run-time, so that parallelism can be exploited instruction by instruction on a large scale. This differs from conventional look-ahead processors [Kel77] where data dependencies are analysed at run-time in a look-ahead buffer of limited size. A by-product of organising the fully asynchronous instruction scheduling this requires is that PE's can be made very tolerant of large and variable memory access latency [AI86], but the overheads are non-trivial (although for a contrary view see [ACE88]). Criticisms of the approach are summarised by Gajski and his colleagues [GPKK82]. Classical early work in the area includes the Manchester Data Flow Machine project, reviewed in [GKW85], and more thoroughly in part I of [CDJ84]. More recent work includes Arvind's "Monsoon" project, reported in [AN87] and [Pap89].

Elsewhere, considerable progress has been made compiling dataflow languages for conventional shared-memory multiprocessors such as the Cray-XMP [Lee88]. This has arisen from success with the partitioning problem, as reported by Sarkar [Sar89].

Hardware description and derivation

Johnson [Joh84a] argues that the abstraction made in circuit design when moving from an analogue to a digital model of device behaviour precisely matches the functional view. There is some practical support for the view, not least the success of commercial products like ELLA [MPT85]. In the context of systolic designs, it has led to a great deal of successful work in deriving efficient VLSI implementations of parallel algorithms. Such work, notably by Quinton [Qui84], Chen [Che84] and Moldovan [Mol83], has developed a considerable understanding of scheduling computations onto fixed arrays of synchronous PE's.

There are some weaknesses with the functional approach. Certainly when one needs to work below or outside the abstraction of synchronous digital circuits, more general techniques are needed. Quite an effective treatment of this is the higher-order logic (HOL) approach of Hanna and Daeche [HD85] and Fourman [FPZ88]. At a higher level, Sheeran identifies a failure to capture ideas like handshaking, where, for example, an input to a circuit includes an acknowledgement output. Sheeran proposes RUBY, an experimental relational language [She88], to deal with the problem.

The existence of hardware description and specification languages poses the quite contentious question of whether programming is a good model for digital systems design, or at least VLSI circuit design. One theme of this book is that parallel programming is different and more complicated than sequential programming. This certainly extends to digital circuit design, which can be yet more complicated when packaging, power distribution and technology mixing are taken into account. Even in the much simplified arena of a single VLSI chip, the use of compilation techniques in favour of more interactive design tools is difficult to justify when the cost of a device rises exponentially with the chip's size.

The Kahn principle

Kahn observed [Kah74] that the meaning of certain simple systems of interacting processes can be given using the standard fixed-point methods of denotational semantics. For a introduction to the denotational approach to giving a mathematical semantics to a program, see Schmidt [Sch86]. The restriction Kahn imposed was that the behaviour of the component processes be characterisable by functions mapping the history (i.e. stream) of input values to the history of output values. Keller [Kel74] details this simplification in a wider context, but it was not until Faustini's thesis [Fau82] that Kahn's "principle" was formally proven in a general context.

Ray tracing

Although it goes back at least to 1968 [App68], the technique was first presented in detail by Turner Whitted in 1980 [Whi80]. It is now commonly dealt with in introductory computer graphics textbooks. The range of smart ray-tracing algorithms is enormous, but they essentially depend on the notion of a "bounding volume", a simple artificial solid introduced to envelop a real object with a more complex form, so that most rays can bypass the intersection test with the complex shapes. This and other techniques are surveyed in [WHG84]. Kajiya [Kaj83] gives an interesting twist to the bounding volume

approach when applied to scenes generated at random in a "fractal" fashion: the scene itself together with bounding volumes is generated only when a ray might strike it.

Automatic transformation techniques

The emphasis in this chapter has been on developing manual techniques for exploring program transformations. A large body of work has been done towards understanding the algebra of programs enough to give algorithms for manipulating programs into specified forms. A simple example (explored by Wadler [Wad88a]) might be the technique of propagating "++" into functions returning lists to avoid copying, as employed in section A.1.1. A more complicated one might be the elimination of **sub** from recurrences, as given in section A.3. For more examples, see Field and Harrison's textbook [FH88].

Computer-aided program transformation and verification environments

Program verifications cannot be expected to be checked by well-qualified reviewers in the same way that mathematical verifications appearing in the scientific literature are. They are generally too boring! A formal approach carries no more weight than vigorous assertion unless the steps are checked, and so computer support is a necessity, not a luxury. Proof checkers exist, LCF being a prime example [GMW79]. Unfortunately using such a system can be very hard work indeed because of the amount of detail required.

One approach to alleviating the tedium of checked verification is to employ a mechanical theorem prover. The classic work in this area is by Boyer and Moore [BM79], whilst Gordon gives an introduction [Gor88] and Chang and Lee give a more general treatment [CL73].

An interesting alternative, which applies more neatly to program derivation, is to capture proof techniques or derivation strategies as programs in a meta-language (in the case of LCF this was ML—which took on a life of its own [Mil83]). These "tacticals" are built by combining fundamental inference rules of the program logic. Milner [Mil85] gives a good introduction. A particular success of the ML/LCF approach of Gordon, Milner and Wadsworth is the use of a polymorphic type system to ensure that only formally-derived statements achieve the status of theorems. Darlington pioneered the application of this proof development work to program derivation with [Dar81], using the functional language HOPE [BMS80] as both the meta-language and the object language.

Darlington's group have gone on to base a complete programming environment on formal program transformation [De88]. Their aim is to capture software specification, derivation and change control by using HOPE$^+$ to document software modifications as executable meta-programs.

Reviews of related work are given by Pepper [Pep83] and Feather [Fea86].

Chapter 5

Distributed Parallel Functional Programming

This chapter examines the problem of controlling the distribution of a parallel program across a loosely-coupled multiprocessor, and develops an extension to the programming language to resolve it.

The main points are

- In general functional programs demand some additional control if the parallelism which is present is to be exploited efficiently.
- This control over spatial program distribution, although not impervious to compiler technology, can legitimately be thought of as part of the programming task.
- The process network diagram used to illustrate parallelism structure is an appropriate level of abstraction for program design and analysis.
- The process network associated with a program is not uniquely determined by the program's form. In fact, the choice of which process network to use to distribute a program can depend on run-time data values.
- A declarative program annotation is presented which associates a program with its process network. The notation has two natural abstraction mechanisms, and has application in more tightly-coupled multiprocessors.

5.1 Communication patterns

A communication occurs every time a value is stored in or read from a memory location. The communication occurs between a memory device and a PE. The memory may be very close to the PE – perhaps on the same chip. It may be in some other PE's local memory, or in some special memory organ shared between many PE's by means of a high-performance interconnection network. At the far extreme, access to non-local memory may be provided by explicitly-programmed message passing over a communications network.

As well as optimising the use of processing power, we must treat the target architecture's communications capabilities as a critical resource. In a loosely-coupled multiprocessor the number of immediately-accessible PE's or memories for each process is strictly limited. Thus, a ring-shaped process network is far easier to accommodate than a star network, or even a mesh or (hyper)cube. The communications demands of a parallel algorithm must be taken into account in evaluating its suitability for such machines. The importance of communication to parallel program performance is often hidden in small-scale parallel computers, but when really large scale parallelism is to be exploited it is inescapable. In devices fabricated using photolithography, e.g. VLSI, it is already paramount.

5.1.1 The speed-up of a sequential multiprocessor

One way to demonstrate the importance of communications is to consider a computation which could be made faster by replicating PE's without involving any parallelism at all.

Let us imagine a single processor computer, linked to a single, large, memory device. Its performance is, to a large extent, governed by the cycle time of the memory device. The speed of a memory device depends on its capacity, since its operation depends on signal propagation across the memory matrix. In the worst case, this propagation delay is quadratic in the memory matrix's width, but with careful design it can be reduced to a logarithmic factor (see Mead and Conway [MC80], section 8.5.2.3).

For some computation, it might be possible to determine in advance that the memory will be accessed region-by-region, in a sequential fashion. Thus, if we could break the memory device up into many sub-memories, each covering exactly one region, we could hope to incur only the access time of a small memory device at each memory access, rather than the access time of the entire memory. With a single PE, this would save no time, because the signals would still need to travel to and from the correct sub-memory, but if we make many duplicates of the PE and scatter them among the submemories, we could arrange for the program to move to the appropriate PE before accessing each region.

Just this effect is exploited with transputers, where there is a special advantage to keeping each process's memory requirements small so that the on-chip RAM is sufficient. Of course, if we can use the results from more than one region's calculation at a time, we can exploit parallelism too, giving a double bonus.

5.1.2 The ray intersection test example

One of the example program transformations used in the previous chapter (section 4.5.1) is an illustration of this. We had a computation initially given as

```
FindImpacts rays objects
 = map (FirstImpact objects) rays
   where
   FirstImpact objects ray = earliest (map (TestForImpact ray) objects)
                             where
                             earliest impacts = insert earlier NOIMPACT impacts
```

Commonly, both the number of rays and the number of objects are very large. It is

clear, therefore, that this program has no shortage of parallelism: we can use horizontal parallelism to spawn a process to compute every element of the result list in parallel. Notice, though, that every one of these processes,

```
FirstImpact objects ray_i
```

will need extensive access to the list objects – in fact *every* process will be continuously accessing *all* of objects. Only in an architecture with a very powerful interconnection scheme can this level of shared memory traffic be supported. Most successful implementations copy the database instead, and with a moderate number of objects copying works well.

The transformed version of the program took the form of a pipeline, each component of which was responsible for a single element of the objects list:

```
FindImpacts2 rays objects = ( (map TakeImpact) ∘
                            (insert (∘) ident
                                    (map map (map PipelineStage objects)))
                            ∘ (map MakePipeItem) )
                            rays
```

where the pipeline stage is defined by

```
PipelineStage object (PIPEITEM ray impact)
                     = PIPEITEM ray impact'
                       where
                       impact' = earlier impact NewImpact
                       NewImpact = TestForImpact ray object
```

The body of the pipeline is a chain of processes evaluating

```
map PipelineStage object pipeitems
```

The pipeline stage's only communications are with the next and previous stages in the chain. The number of objects handled by each stage can be increased if necessary, allowing complete control over how much local memory is used, and over the communication/computation ratio. This implementation does, indeed, seem to win the "double bonus" promised above!

5.1.3 Is this programming?

This leads us to a rather awkward question: are the two formulations of the ray intersection test given above different parallel algorithms? In sequential programming terms, we must answer "no", for they do represent exactly the same computation. Nonetheless the difference is substantial: the pipelined form elucidates an organisation of the problem which seems practically important.

Because the problem of distributing a parallel computation in space is so complicated, it is reasonable to consider taking the view that it is the programmer's responsibility. If

Figure 5.1: A four-element cyclic graph

we take this view, then we must hope to offer some support in the programming language to make the task easy. We can still hope for theoretical advances to eliminate such details from programmers' daily work; the notation presented might then be an intermediate form in the compilation process.

5.2 Declarative descriptions of process networks

We have discovered that the unadorned text of a program does not constrain its parallel evaluation enough for us to claim that the script serves to describe a parallel algorithm.

Instead of refining programs into descriptions of parallel algorithms using annotations to control operational aspects of evaluation, it seems preferable to apply declarative programming to the problem. A process network (like any graph) can be described by its nodes and its arcs. If we name the nodes in the program a, b, c etc. we can represent the network's arcs using a list of assertions. For example, the four element cyclic graph shown in figure 5.1 can be written

(arc a b) ∧ (arc b c) ∧ (arc c d) ∧ (arc d a)

Here, a, b, c and d are labels identifying expressions in the program, and ∧ denotes logical "and". By asserting arc a b, the programmer is informing the compiler that expressions a and b ought to be computed in parallel, and that the processes evaluating them are expected to interact. Just how they interact is not explained by the process network: one must refer to the definitions of the labelled expressions. In particular, the arcs in the process network described do not carry arrows indicating any particular direction of information flow. We will see examples (see section 5.3.3) where an arc stands for a bidirectional flow.

Let us look a little more closely at a simpler example, a three element chain:

(arc a b) ∧ (arc b c) ∧ (arc c d)

with the associated definitions

```
a = map ((+) 2) b
b = map ((×) 3) c
c = map sqrt d
d = from 1
```

The network assertion arc c d demands that the expressions named c and d each be an independent process, to be allocated to a processor of its own (at least notionally). It further requires the compiler to arrange things so that the processor executing c is linked directly to the processor executing d, because they are expected to communicate during the computation.

Looking at the body of d we find that there are no free variables: the expression is quite self-contained, apart from the code it executes. Its only necessary communication is to deliver its result stream.

The body of c has one free variable (apart from the code it executes), d. Fortunately, we already know that d is placed on a neighbouring processor, so its value is easily available to the processor which must evaluate c. Indeed it is because of this dependency that the arc assertion was made. The compiler can check that all of a program's data dependencies are reflected by arcs in the network, although the programmer may choose to ignore such warnings if the expected level of traffic on the arc concerned is thought to be very small.

Looking at b the pattern should become clear. It is the entire expression which is named and mentioned in the declarative process network description. Thus, b is the complete function application map ((×) 3) c. This might seem somewhat confusing since in our process network diagram this node would be labelled just map ((×) 3), being the function the node applies to its input. We use a shorthand to resolve it, introduced shortly.

5.2.1 A process network language

This technique for associating a program script with a process network diagram forms the basis for an interesting extension to our functional programming language. We add a new keyword, **moreover**, to introduce a "moreover clause" containing a declarative description of the structure of the intended process network, using as labels any names currently in scope. The resulting language is called "Caliban" for somewhat obscure reasons, after the character from Shakespeare's *The Tempest*. Caliban, bereft of his once-great magical power, is much maligned in modern interpretations of the play.

Thus the example above might be written in Caliban as

```
a
where
a = map ((+) 2) b
b = map ((×) 3) c
c = map sqrt d
d = from 1
moreover
(arc a b) ∧ (arc b c) ∧ (arc c d)
```

5.2.2 A shorthand for naming processes

This description is a little complicated, because every time we want to distinguish a process we must use where to give a name to the application. We introduce a shorthand, □f (read "make process f"), to denote a name for the application in which f appears. The function f must appear exactly once in the source program (or a compile-time error should be reported). We can use it to re-express our pipeline as

```
f (g (h d))
where
f = map ((+) 2)
g = map ((×) 3)
h = map sqrt
d = from 1
moreover
(arc □f □g) ∧ (arc □g □h) ∧ (arc □h d)
```

Of course it would be equivalent to use parameterised definitions of f, g and h, such as

```
f xs = map ((+) 2) xs
```

The difference between f and □f is that f is the name of a process to compute the function so named, while □f is the name of a process which applies it. Because our host language allows functions as values, either of these can constitute a sensible process body.

We can be a little informal and use composition rather than application provided the application can be uncovered by the use of reduction at compile-time. Doing this allows us to write

```
(f ∘ g ∘ h) d
where
f = map ((+) 2)
g = map ((×) 3)
h = map sqrt
d = from 1
moreover
(arc □f □g) ∧ (arc □g □h) ∧ (□h d)
```

5.2.3 Abstracting process networks

It is frustrating to have to write out the details of highly-structured process networks like our chain example. It would be much tidier to have some means of defining once-and-for-all what a process chain looks like. This turns out to be very easy, because we have all the mechanisms we require in the host language. In order to specify a chain network given a list of functions, $[f_1, f_2, f_3, \ldots f_{n-1}, f_n]$, we need to build an assertion

$(\mathsf{arc}\ \Box\mathsf{f}_1\ \Box\mathsf{f}_2) \wedge (\mathsf{arc}\ \Box\mathsf{f}_2\ \Box\mathsf{f}_3) \wedge \ldots \wedge (\mathsf{arc}\ \Box\mathsf{f}_{n-1}\ \Box\mathsf{f}_n)$

This is easy. We define a function, which we will call a "network forming operator":

$\mathsf{chain} :: (\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}) \to [(\alpha \to \beta)] \to \mathsf{Bool}$

$\mathsf{chain\ relation}\ [\mathsf{f}] = \mathsf{TRUE}$
$\mathsf{chain\ relation}\ (\mathsf{f}_1 : \mathsf{f}_2 : \mathsf{fs}) = (\mathsf{relation}\ \mathsf{f}_1\ \mathsf{f}_2) \wedge (\mathsf{chain\ relation}\ \mathsf{f}_2\ \mathsf{fs})$

Now suppose we write

$(\mathsf{f} \circ \mathsf{g} \circ \mathsf{h})\ \mathsf{d}$
moreover
$\underbrace{\mathsf{chain\ arc}\ [\Box\mathsf{f},\ \Box\mathsf{g},\ \Box\mathsf{h},\ \mathsf{d}]}$

We can apply reduction to expand this annotation:

$(\mathsf{f} \circ \mathsf{g} \circ \mathsf{h})\ \mathsf{d}$
moreover
$(\mathsf{arc}\ \Box\mathsf{f}\ \Box\mathsf{g}) \wedge \underbrace{(\mathsf{chain\ arc}\ [\Box\mathsf{g},\ \Box\mathsf{h},\ \mathsf{d}])}$

$(\mathsf{f} \circ \mathsf{g} \circ \mathsf{h})\ \mathsf{d}$
moreover
$(\mathsf{arc}\ \Box\mathsf{f}\ \Box\mathsf{g}) \wedge (\mathsf{arc}\ \Box\mathsf{g}\ \Box\mathsf{h}) \wedge \underbrace{(\mathsf{chain\ arc}\ [\Box\mathsf{h},\ \mathsf{d}])}$

$(\mathsf{f} \circ \mathsf{g} \circ \mathsf{h})\ \mathsf{d}$
moreover
$(\mathsf{arc}\ \Box\mathsf{f}\ \Box\mathsf{g}) \wedge (\mathsf{arc}\ \Box\mathsf{g}\ \Box\mathsf{h}) \wedge (\mathsf{arc}\ \Box\mathsf{h}\ \mathsf{d}) \wedge \underbrace{(\mathsf{chain\ arc}\ [\mathsf{d}])}$

$(\mathsf{f} \circ \mathsf{g} \circ \mathsf{h})\ \mathsf{d}$
moreover
$(\mathsf{arc}\ \Box\mathsf{f}\ \Box\mathsf{g}) \wedge (\mathsf{arc}\ \Box\mathsf{g}\ \Box\mathsf{h}) \wedge (\mathsf{arc}\ \Box\mathsf{h}\ \mathsf{d}) \wedge \mathsf{TRUE}$

Because the annotation has the form of a conjunction of Boolean assertions, TRUE represents the empty annotation, and so can be removed.

Other network-forming operators

A chain is not the only useful pattern of communication to capture. Different applications may require more specialised structures, but we can complete an initial toolkit with the functions ladder and fan defined as follows:

```
ladder :: (Bool → Bool → Bool) → [α → β] → [γ → δ] → Bool

ladder relation [ ] [ ] = TRUE
ladder relation (a : as) (b : bs) = (relation a b) ∧ (ladder relation as bs)
```

Thus, ladder arc as bs takes two lists of processes and builds an assertion that they be linked pairwise by the relation. It might more neatly be expressed as

```
ladder relation as bs = all (map2 relation as bs)
                        where
                        all = insert (∧) TRUE
```

This makes clear the relationship between ladder and map2.

The fan operator takes a process and a list of processes and builds an assertion that every process in the list is linked to the first process:

```
fan :: (Bool → Bool → Bool) → α → [β → γ] → Bool

fan relation a bs = all (map (relation a) bs)
```

We will assume vector variants of the network forming operators, defined like

```
VectorLadder relation av bv = ladder relation (VectorToList av) (VectorToList bv)
```

5.2.4 A second abstraction mechanism

A function like chain allows patterns of arcs to be abstracted, and treated as a single link of a higher-level kind. A dual to this *arc abstraction* mechanism is *node abstraction*, a means of packaging up collections of nodes as a single, higher-level form. Let us continue with the process chain example we have been using so far, which is, in fact, used as a pipeline. We could try to capture the notion of a pipeline as a single higher-order combining form:

```
pipeline :: [α → α] → α → α

pipeline fs x = (insert (∘) ident fs) x
                moreover
                (chain arc (map (□) fs))
                ∧ (arc □(last fs) x)
                ∧ (arc □(hd fs) interface)
```

pipeline takes a list of functions, each of which specifies a component process of the pipeline. The **moreover** annotation asserts that the components are each separate processes linked into a chain, while the body part fills in the details of the communications, applying each function to the result of the next in the pipeline. Note that map (□) fs must be reduced so that □ is applied to each element of fs before being interpreted as a reference to the

processes.

The keyword interface is a shorthand for the name of the function's result – the definition above is equivalent to

```
pipeline fs x = result
                where
                result = (insert (∘) ident fs) x
                moreover
                (chain arc (map (□) fs))
                ∧ (arc □(last fs) x)
                ∧ (arc □(hd fs) result
```

The purpose of making this link to result is to connect the output of the process network of this function correctly into the process network of the calling program. In this example, the pipeline has a single output arc, so interface is the name of a single node.

5.2.5 Simplification rules

We have now seen all the components of the Caliban language. Examples of applications will be given shortly, but first a simple program will be analysed. The fundamental notation is based on naming and the **moreover** clause, but this is augmented by the □ operator, the interface pseudonym, the use of network forming operators like chain, and the appearance of **moreover** clauses in auxiliary function definitions like pipeline as well as at the outermost level of a program. We understand the meaning of complex constructions by applying simplification rules and reduction. The rules can be summarised as

$$\begin{aligned} \mathsf{e}_1\{\mathsf{f\ x}\}^1 \textbf{ moreover } \mathsf{e}_2\{\Box\mathsf{f}\} &\equiv \mathsf{e}_1\{\mathsf{a}\} \textbf{ where } \mathsf{a} = \mathsf{f\ x} \textbf{ moreover } \mathsf{e}_2\{\mathsf{a}\} \\ \mathsf{LHS} = \mathsf{e}_1 \textbf{ moreover } \mathsf{e}_2\{\mathsf{interface}\} &\equiv \mathsf{LHS} = \mathsf{a} \textbf{ where } \mathsf{a} = \mathsf{e}_1 \textbf{ moreover } \mathsf{e}_2\{\mathsf{a}\} \\ \mathsf{e}_A\{\mathsf{e}_B \textbf{ moreover } \mathsf{e}_C\} \textbf{ moreover } \mathsf{e}_D &\equiv \mathsf{e}_A\{\mathsf{e}_B\} \textbf{ moreover } \mathsf{e}_C \wedge \mathsf{e}_D \end{aligned}$$

Where

- a is a currently unused name in each case,
- $\mathsf{e}_1\{\mathsf{e}_2\}$ stands for an expression e_1 containing an instance of e_2.
- $\mathsf{e}_1\{\mathsf{e}_2\}^1$ stands for an expression e_1 containing exactly one instance of e_2.
- all the names referred to by e_C are defined in e_A correctly.

A program is compiled by applying these rules, together with reduction as required, until the program has just one **moreover** clause consisting only of a conjunction of arc assertions about named processes. This may involve evaluating a substantial portion of the program, and there is a risk that the compiler will fail to terminate.

The last restriction means that parameters and names defined in nested **where** clauses must be lifted to the outermost lexical level. This may not be possible. It might, therefore, prove impossible to float all **moreover** clauses to the outermost level. For example, this occurs if a recursive function depending on a run-time variable is annotated so that the

Figure 5.2: The expanded process network

nesting of **moreover** clauses cannot be unravelled at compile-time. Such programs specify dynamic process networks, which evolve at run-time. These are not dealt with by this simplification scheme. Some examples are given in section 5.2.7.

5.2.6 An example of simplification

A simple example of the use of these rules – which would normally be applied by the compiler – is given below. This program specifies the process network given in figure 5.2:

```
f x y = combine (left x)
                (right y)
        where
        combine xs ys = map2 op xs ys
        left = pipeline [f1, g1, h1]
        right = pipeline [f2, g2, h2]
        moreover
        (fan □□combine [□left, □right])
        ∧ (arc □□combine interface)
```

There are three points to watch out for here:

- When a two-parameter curried function like combine is applied to both parameters, two □ operators are needed to refer to the complete application.
- The presence of the arc assertion between two processes does not reflect the direction of data transfer.
- At this level, □left and □right appear to be single processes. Since they are instances of pipeline, they actually unfold into pipelines of three processes each. The output of each pipeline is its "**interface**", to which it delivers its output, and to which processes consuming its output are linked.

  The input of each pipeline is connected by the assertion that each pipeline is linked to its parameter x.

We begin by expanding the **interface** and □ shorthands:

```
f x y
 = result

   where

   result = boxboxcombine
   boxboxcombine = combine boxleft
                           boxright
   boxleft = pipeline [f1, g1, h1] x
   boxright = pipeline [f2, g2, h2] y
   combine xs ys = map2 op xs ys

   moreover

   (fan boxboxcombine [boxleft, boxright])
   ∧ (arc boxboxcombine result)
```

Since boxboxcombine = result, arc boxboxcombine result = TRUE. This is evidence of some redundancy in the notation: the **interface** link was not strictly necessary here, but was included to make all the arcs manifest in the **moreover** clause. The next step is to unfold the references to pipeline. We take the first one, and remove the use of "∘", □, **interface** and chain arc straight away:

```
f x y
 = result

   where

   result = boxboxcombine
   boxboxcombine = combine boxleft
                           boxright
```

```
boxleft = result2
          where
          result2 = boxf1
          boxf1 = f1 boxg1
          boxg1 = g1 boxh1
          boxh1 = h1 x
          moreover
          (chain arc [boxf1, boxg1, boxh1])
          ∧ (arc boxh1 x)
          ∧ (arc boxf1 result2)

boxright = pipeline [f2, g2, h2] y
combine xs ys = map2 op xs ys

moreover

(fan boxboxcombine [boxleft, boxright])
∧ (arc boxboxcombine result)
```

Notice the need to introduce result2 to avoid a clash with result. The next step is to try to float the nested **moreover** clause out to join the outer one. To do this we must make sure that all the names it refers to are defined in the outer scope:

```
f x y
 = result
   where
   result = boxboxcombine
   boxboxcombine = combine boxleft
                           boxright

   boxleft = result2
             moreover
               (chain arc [boxf1, boxg1, boxh1])
               ∧ (arc boxh1 x)
               ∧ (arc boxf1 result2)

   result2 = boxf1
   boxf1 = f1 boxg1
   boxg1 = g1 boxh1
   boxh1 = h1 x
   boxright = pipeline [f2, g2, h2] y
   combine xs ys = map2 op xs ys
```

```
    moreover

    (fan boxboxcombine [boxleft, boxright])
    ∧ (arc boxboxcombine result)
```

Merging **moreover** clauses gives

```
f x y
 = result
    where
    result = boxboxcombine
    boxboxcombine = combine boxleft
                            boxright

    boxleft = result2
    result2 = boxf1
    boxf1 = f1 boxg1
    boxg1 = g1 boxh1
    boxh1 = h1 x
    boxright = pipeline [f2, g2, h2] y
    combine xs ys = map2 op xs ys

    moreover

    (chain arc [boxf1, boxg1, boxh1])
    ∧ (arc boxh1 x)
    ∧ (arc boxf1 result2)
    ∧ (fan boxboxcombine [boxleft, boxright])
    ∧ (arc boxboxcombine result)
```

We complete the simplification by doing the same with the other instance of pipeline, and unfolding the uses of chain and fan:

```
f x y
 = result
    where
    result = boxboxcombine
    boxboxcombine = combine boxleft
                            boxright
```

boxleft = result2
result2 = $boxf_1$
$boxf_1$ = f_1 $boxg_1$
$boxg_1$ = g_1 $boxh_1$
$boxh_1$ = h_1 x
boxright = result3
$boxf_2$ = f_2 $boxg_2$
$boxg_2$ = g_2 $boxh_2$
$boxh_2$ = h_2 y
combine xs ys = map2 op xs ys

moreover

(arc $boxf_1$ $boxg_1$) $\wedge$ (arc $boxg_1$ $boxh_1$)
$\wedge$ (arc $boxh_1$ x)
$\wedge$ (arc $boxf_1$ result2)
$\wedge$ (arc $boxf_2$ $boxg_2$) $\wedge$ (arc $boxg_2$ $boxh_2$)
$\wedge$ (arc $boxh_2$ x)
$\wedge$ (arc $boxf_2$ result3)
$\wedge$ (fan boxboxcombine [boxleft, boxright])
$\wedge$ (arc boxboxcombine result)

This definition now uses only Caliban's fundamental mechanisms. We might call this "normal form" Caliban.

5.2.7 Some examples where simplification fails

Simplification may not always succeed in finding a "normal form" Caliban formulation of the input program. This happens with programs whose process network evolves during program execution, as with the primes sieve program (section 4.6.1):

```
primes = sieve (from 2)
         where
         sieve (a : as) = a : (sieve (filter a as))
                          moreover
                          arc □sieve □□filter
```

It also happens if insufficient information is available to determine the process network, as in a program like

```
f xs = map g xs
       moreover
       fan □g (map (□) xs)
```

where xs is a program input, and yet determines the size of the network.

Finally, a rather pathological possibility is that the computation necessary to find the process network fails to terminate, even though the program terminates correctly. This can happen if the output of the process network is not needed at run time, as might occur in this example:

```
f a = if (satisfactory a)
        a
        (map g xs)
      where
      xs = 1 : xs
      moreover
      fan □g (map (□) xs)
```

5.3 Some examples

Having introduced the bones of the Caliban language, we need some examples to see how it works out in practice.

5.3.1 Example: the square root pipeline

This example is derived from an Occam tutorial. It is very simple, being a systolic algorithm of the most basic kind. The problem is to take a list of numbers, and compute the list of their corresponding square roots. The solution is a pipeline algorithm, with one stage for each iteration of the Newton-Raphson approximation technique.

Begin with the standard Newton-Raphson algorithm, from section 4.3.1:

```
solve f f' x0
= until converges xs
   where
   converges 0 = FALSE
   converges i = abs(((xs sub i) - (xs sub (i-1)))/(xs sub i))≤ε,   if i ≥ 1
   xs = generate NextEstimate
        where
        NextEstimate 0 = x0
        NextEstimate i = (xs sub (i-1))
                         - ( f (xs sub (i-1)) / f' (xs sub (i-1)) ),    if n ≥ 1
```

To find a square root, we solve for $f\ x = 0$, where

$$f\ x = x^2 - a$$

so $f'\ x = 2 \times x$. A fair guess to start with is $x_0 = a/2$, so we have

```
sqrt a
= until converges xs
  where
  converges 0 = FALSE
  converges i = abs(((xs sub i) - (xs sub (i-1)))/(xs sub i))≤ε,   if i ≥ 1
  xs = generate NextEstimate
       where
       NextEstimate 0 = a
       NextEstimate i = ((xs sub (i-1)) + a/(xs sub (i-1)))/2, if i ≥ 1
```

Testing for convergence at each step is quite expensive; it turns out to be easier (and much better for a pipelined implementation) to iterate a fixed number of times before finishing:

```
sqrt a
= until finished xs
  where
  finished i = TRUE, if i = NumIterates
  finished i = FALSE, otherwise
  xs = generate NextEstimate
       where
       NextEstimate 0 = a/2
       NextEstimate i = ((xs sub (i-1)) + a/(xs sub (i-1)))/2
```

Applying our standard technique for optimising recurrences (Appendix A, section A.3), this becomes

```
sqrt a = xs sub NumIterates
         where
         xs = (a/2) : (map Transition xs)
         Transition prevx = (prevx + a/prevx)/2
```

If we fix NumIterates at some given value, say four, and apply reduction this can be written

```
sqrt a = (Transition ∘ Transition ∘ Transition ∘ Transition) (a/2)
         where
         Transition prevx = (prevx + a/prevx)/2
```

If we apply fact 2 (Appendix A, section A.4) here we can make the free variable a in Transition a parameter of Transition, and propagate it through using a pair:

```
sqrt a = fst ((Transition' ∘ Transition' ∘ Transition' ∘ Transition')
             (a/2, a))
         where
         Transition' (prevx, a) = ((prevx + a/prevx)/2, a)
```

We actually need to apply this to a stream of incoming values. By distributing map over

"∘" we get

```
map sqrt as = ((map fst)
               ∘ (map Transition')
               ∘ (map Transition')
               ∘ (map Transition')
               ∘ (map Transition')
               ∘ (map MakePair))
              as
              where
              Transition' (prevx, a) = ((prevx + a/prevx)/2, a)
              MakePair a = (a/2, a)
```

Adding the network annotation

This is the final, parallel form of the square root pipeline. We have now to use Caliban to distribute it in the obvious pipeline fashion. We can do this in a number of ways: we could use the pipeline operator given earlier:

```
map sqrt as = pipeline [map fst,
                        map Transition',
                        map Transition',
                        map Transition',
                        map Transition',
                        map MakePair]
                       as
```

Alternatively, we could use chain:

```
map sqrt as = (insert (∘) ident processes) as
              where
              processes = [map fst,
                           map Transition',
                           map Transition',
                           map Transition',
                           map Transition',
                           map MakePair]
              moreover
              (chain arc (map (□) processes))
              ∧ (arc □(last processes) as)
```

(As an aside, notice that the elements of the list processes are not all of the same type, and so fail to satisfy the type scheme even although no run-time type error can occur. The problem disappears after simplification so is not pursued here).

5.3.2 Bundling: a partitioning technique

One of the main functions of the annotation is to control process partitioning. Bundling is a way of doing so which does not disturb existing code. Suppose we wish to place some of the components of the composition above in the same PE. We could modify the code body to get the effect:

```
map sqrt as = (insert (∘) ident processes) as
              where
              processes = [(map fst) ∘ (map Transition'),
                           (map Transition') ∘ (map Transition'),
                           (map Transition') ∘ (map MakePair)]
              moreover
              (chain arc (map (□) processes))
              ∧ (arc □(last processes) as)
```

In making a change to the distribution of the program, we had to change the body of the code. The idea of bundling is to build a data structure containing the component parts of each process, and talk about that instead. Any data structure will do, but for clarity we will tag data structures built for bundling purposes with the constructor BUNDLE:

```
Bundle α ::= BUNDLE α
```

We can now write the partitioned pipeline as follows:

```
map sqrt as = (insert (∘) ident processes) as
              where
              processes = [map fst,
                           map Transition',
                           map Transition',
                           map Transition',
                           map Transition',
                           map MakePair]

              [back, t1, t2, t3, t4, front] = map (□) processes
              partitions = [BUNDLE (back, t1),
                            BUNDLE (t2, t3),
                            BUNDLE (t4, front)]
              moreover
              chain arc partitions
```

Here, we named the components of processes back, t1, t2 ... front (by defining them using pattern matching). Once they have names it is easy to use BUNDLE to bundle them up.

It would have been tidier, though perhaps not quite so clear, to write

```
partitions = map2 Make2Bundle oddones evenones
             where
             Make2Bundle a b = BUNDLE (a, b)
             oddones = OddOnes (map (□) processes)
             evenones = EvenOnes (map (□) processes)
```

where OddOnes and EvenOnes select the odd- and even-indexed elements of the list. The next example gives a more convincing demonstration of bundling at work.

5.3.3 Example: local neighbourhood operations

A two-dimensional local neighbourhood operation takes a matrix and maps each element to a function of its immediate neighbours, producing a new matrix. Local neighbourhood operations are widely used in image processing and in computational physics.

In our vector notation we might define a generic function for applying local neighbourhood operations:

```
ApplyLNO :: ([α] → α) → ≪α≫ → ≪α≫


ApplyLNO op matrix
 = MakeMatrix LocalOperation
   where
   LocalOperation (i,j) = matrix sub (i,j), if OnBoundary matrix (i,j)
   LocalOperation (i,j)
    = op [matrix sub (i-1,j),
          matrix sub (i,j-1),
          matrix sub (i+1,j),
          matrix sub (i,j+1),
          matrix sub (i,j)], otherwise
```

where

```
OnBoundary matrix (i,j) = (i=0) ∨ (j=0) ∨ (i=iBound-1) ∨ (j=jBound-1)
                          where
                          (iBound,jBound) = MatrixBound matrix
```

The parameter op determines what function of the four neighbours is applied. A typical one might be a low-pass filter:

```
LowPass matrix = ApplyLNO average matrix
```

where

```
average [west, south, east, north, home] = (west+south+east+north+home)/5
```

Very commonly we will apply a local neighbourhood operation repeatedly, producing a

sequence of iterates:

```
iterate (ApplyLNO LowPass) InitialMatrix
 = [InitialMatrix,
    ApplyLNO LowPass InitialMatrix,
    ApplyLNO LowPass (ApplyLNO InitialMatrix),
    ApplyLNO LowPass (ApplyLNO LowPass (ApplyLNO InitialMatrix)),
    ⋮
```

Constructing a network-forming operator for a mesh

Although not the only option, for the sake of an example we will distribute this program over a mesh of PE's. To do so, we need a network forming operator which takes a matrix and asserts that each element is computed by a separate PE, and that each PE interacts with the element's four nearest neighbours. This turns out quite surprisingly easy:

```
mesh :: ≪α≫ → Bool

mesh matrix = MatrixAll (ApplyLNO LinkNeighbours matrix)
              where
              LinkNeighbours [west, south, east, north, home]
               = fan arc home [west, south, east, north]
```

where MatrixAll matrix is TRUE just when every element of matrix is TRUE. Just as fan corresponds to map and ladder corresponds to map2, so there is a natural relationship between mesh and ApplyLNO.

Adding the network annotation

We are now ready to express the distribution of the local-neighbourhood operation computation over a mesh of PEs:

```
LowPass matrix = output
                 where
                 output = ApplyLNO average matrix
                 moreover
                 mesh output
```

In fact we can write this more briefly using the **interface** shorthand:

```
LowPass matrix = ApplyLNO average matrix
                 moreover
                 mesh interface
```

This makes clear that the elements of the array are generated in a distributed fashion.

If the result were passed to another mesh-distributed function, the two process networks could be combined to minimise data movement.

Distributing an iterated local neighbourhood operation

If we wanted to iterate this computation, maintaining the expression-to-PE mapping from iteration to iteration, we could try writing

```
IteratedFilter initialmatrix = iterate LowPass initialmatrix
```

where LowPass is the distributed version given above. This doesn't work because the simplification scheme cannot shift the **moreover** clause to the outer level, because the matrix parameter is different for each successive iteration. It actually specifies that a fresh network be used for each iteration. Instead we must build a structure to bundle the values we want each PE to compute – that is, corresponding elements of successive matrices.

This comes out quite easily by permuting indices. What we need is a single matrix, each of whose elements is a stream of elements from successive iterations. Let us define a function StreamOfMatricesToMatrixOfStreams to make this transformation: we demand that

```
(ms sub k) sub (i,j) = ((StreamOfMatricesToMatrixOfStreams ms) sub (i,j))
                                                                sub k
```

This is achieved by the definition

```
StreamOfMatricesToMatrixOfStreams ms
 = MakeMatrix (MatrixBounds (hd ms)) EachStream
   where
   EachStream (i,j) = generate Elements
                      where
                      Elements k = (ms sub k) sub (i,j)
```

(We assume that all the matrices in the stream have the same bounds as the first, hd ms).

We are now ready to give the distributed implementation of IteratedFilter:

```
IteratedFilter initialmatrix = iterate LowPass initialmatrix
                               where
                               LowPass matrix = ApplyLNO average matrix
                               moreover
                               mesh (MatrixMap BUNDLE
                                     (StreamOfMatricesToMatrixOfStreams interface))
```

Here MatrixMap is the natural extension of map to matrices. It is used to introduce the BUNDLE tags, which appear simply to indicate to the reader which data structures are used for bundling purposes.

Partitioning the local-neighbourhood operation

Hardware specially designed for such algorithms may be able to implement this distribution efficiently, but for more general-purpose architectures there is a "grain size" problem: each PE does a great deal of communication for each item of useful computation performed. At each step, each PE receives data from four neighbours, does five arithmetic operations, and then distributes the result to four neighbours.

For efficient execution on more typical hardware, we can break the matrix up, making each PE responsible for a sub-matrix rather than a single element. This strategy reduces the ratio of communication to computation in direct proportion to the ratio between the perimeter and the area of the submatrices. Let us define a partitioning function Partition n m, which builds an n×n matrix of adjacent equal-size submatrices of m. We require n to divide both of m's bounds exactly. The property required is that:

```
mat sub (i,j) = ((Partition n mat) sub (majori,majorj))
                                   sub (minori,minorj)
                where
                majori = i div n
                majorj = j div n
                minori = i mod n
                minorj = j mod n
```

(where **div** denotes integer division, and **mod** denotes the remainder). This specification is satisfied by

```
Partition n matrix
 = MakeMatrix (n,n) EachSubMatrix
   where
   EachSubMatrix (majori,majorj)
    = MakeMatrix (minoriBound,minorjBound) Elements
      where
      Elements (minori,minorj) = matrix sub (imap majori minori, jmap majorj minorj)
      imap majori minori = (majori×minoriBound) + minori
      jmap majorj minorj = (majorj×minorjBound) + minorj
      (iBound,jBound) = MatrixBounds matrix
      minoriBound = iBound div n
      minorjBound = jBound div n
```

Given this partitioning function, the distributed, partitioned, iterated filter program can be written as

```
IteratedFilter initialmatrix = iterate LowPass initialmatrix
                               where
                               LowPass matrix = ApplyLNO average matrix
                               moreover
                               mesh (MatrixMap BUNDLE
                                      (partition MeshDimension
                                       (StreamOfMatricesToMatrixOfStreams interface)))
```

This specifies that the computation be distributed over a MeshDimension×MeshDimension four-connected mesh of PEs. Notice that the body of IteratedFilter remains unchanged.

Transformation into stream-processing form

There are some implementation problems with this formulation, because each component process accesses a sequence of global matrices. A compiler must check the matrix indices to verify that each process accesses only its neighbours. We can simplify matters a great deal by modifying the program so that all the indexing occurs at compile-time. We will claim, but not prove, that

1. We have an inverse for StreamOfMatricesToMatrixOfStreams so that

   ```
   MatrixOfStreamsToStreamOfMatrices (StreamOfMatricesToMatrixOfStreams as) = as
   ```

2. That we can propagate map inside ApplyLNO:

   ```
   map (ApplyLNO f) as = MatrixOfStreamsToStreamOfMatrices
                          (ApplyLNO ((map f) ∘ transpose)
                                    (StreamOfMatricesToMatrixOfStreams as))
   ```

   (where transpose (defined on page 98) interchanges rows and columns in a list-of-lists).

Now recall that

```
iterate f x = output
              where
              output = x : (map f output)
```

so that

```
IteratedFilter initialmatrix
 = output
   where
   output = initialmatrix : (map LowPass output)
```

so that

```
IteratedFilter initialmatrix
 = output
   where
   output = initialmatrix
            : (MatrixOfStreamsToStreamOfMatrices
              (ApplyLNO ((map f) ∘ transpose)
                        (StreamOfMatricesToMatrixOfStreams output)))
```

This can be simplified by using the property that

```
StreamOfMatricesToMatrixOfStreams (initialmatrix : xs)
 = MatrixMap2 (:) initialmatrix (StreamOfMatricesToMatrixOfStreams xs)
```

(where MatrixMap2 is the natural extension of map2 to matrices). This lets us write

```
IteratedFilter initialmatrix
 = MatrixOfStreamsToStreamOfMatrices output
   where
   output = MatrixMap2 (:) initialmatrix
            (ApplyLNO ((map f) ∘ transpose)
                      output)
```

Now when we add the **moreover** clause,

```
   moreover
   mesh (MatrixMap BUNDLE
         (partition n output))
```

we can unfold the program so that all communication paths are manifest at compile-time, and carry streams. We have built a mesh of processes, each interacting in both directions with their four nearest neighbours. The arcs's of the process network correspond to multiple bidirectional communication channels – as was promised.

5.4 Implementation of static network programs

Provided sufficient parameters are supplied, and the computation required to build the process network terminates, a program which employs the shorthand and abstraction mechanisms available can be simplified to "normal form", in which there is just one **moreover** clause qualifying the entire program, consisting of a simple conjunction of arc assertions applied to expression names.

In this section we explain how normal form programs can be compiled to efficient object code for a loosely-coupled multiprocessor. The work described here is still in progress: no implementation of Caliban exists yet.

5.4.1 Compiler structure

- **Process separation:** The expression each process is to compute is separated into a distinct **process** construct. All the definitions on which this expression depends are also included in each **process** construct, with the exception of any definition mentioned in the **moreover** clause.

  References to names referred to in the **moreover** clause are replaced by calls to special communications code, detailed in a moment.

- **Process compilation:** Each **process** construct is compiled using conventional compilation technology (as described in Chapter 3).

- **Mapping and configuration:** The logical network specified by the **moreover** clause is analysed to find how best to embed it in the available multiprocessor network. For reconfigurable networks this involves generating a table of interconnection switch settings. For non-configurable networks it involves finding a graph embedding which maintains locality as well as possible. For dynamic-routing networks it involves choosing a layout which will minimize network congestion, and allocating network addresses.

  The mapping phase may fail, if the logical network makes demands on the physical network which cannot be met – for example requiring too many local neighbours.

  The output consists of the network configuration code, which depends on the network design, together with a binding of process names to PE identifiers.

- **Link-editting and load module construction:** Finally, each process is linked with libraries and the run-time system as required. The linker produces a file ready for loading on the multiprocessor, making sure that the right code is copied across the network to the right PE as required.

The most complicated part of the implementation lies in the run-time system, necessary to handle inter-processor communication correctly. Rather than go into the compiler phases in great detail, we will concentrate on when communication occurs, and what has to be done when it does.

5.4.2 When does communication occur?

Communication may occur when a process (as separated in the first phase of the compiler) refers to a name which is itself identified as a (different) process. The compiler should check that this happens just when an arc assertion links the two processes. If a process refers to a name which is not identified in the **moreover** clause as a different process, then the expression to which the name refers is incorporated in the process.

If such a name is referred to by more than one process, then its defining expression is duplicated in the body of each one. This is very natural when the expression is already in normal form, as most function definitions are – so code is copied to those PE's which might refer to it. If the expression is not in normal form, it is often still sensible to copy it, but if the recomputation involved is substantial some warning to the programmer is

probably justified. Local recomputation is quite commonly preferred over having a single global copy of an object.

5.4.3 Channels: the implementation of communication

We will present an implementation scheme which handles all possible cases; much of this might be simplified by an optimizing compiler. For the time being we will consider only channels corresponding to stream communications. We can generalise later.

Let us call the link between two processes a *channel*. Channels may be created at run-time – in fact we will assume that all channels are. Moreover, there may be several channels linking two PE's, for reasons which should become clear later. Each channel is implemented by a pair of drivers, the *sender* and the *receiver*, responsible for managing the link. The channel carries successive elements of the stream. How the elements themselves are represented is discussed later.

The receiver

The receiver tries to maintain a full buffer of received values, so that the receiving process need never be delayed waiting for the sender to respond. We can think of the receiver sending the sender a bag of tokens, each allowing the sender to write one value to the receiver's buffer. Every time the receiving process consumes a value from its buffer, it sends the corresponding token back to the sender. A value is delivered by the receiver to the receiving process by copying into the heap of the receiving process. This is necessary to make sure that the buffer space is freed for subsequent use.

The sender

The sender end of a channel corresponds quite closely to the notion of a process, since it is the sender which generates demand for values. The distinction is that our **process** is identified with the expression to be evaluated. There may be several references to the expression, so there may be several channels linking to it. Thus, there may be several senders each holding a reference to the expression.

When the expression constructs a stream, it may happen that after some computation the different senders refer to different parts of the stream. This situation may persist because a sender may be blocked awaiting a token from its corresponding receiver. In this event, a blocked sender may hold a pointer to an early part of the stream while another sender demands many more elements. In this event the intermediate list elements must be stored (in the heap, as usual) indefinitely. This is all managed quite naturally by a garbage collector provided every sender process, and all to which it refers, is treated as non-garbage. We are committed to ensuring that sender processes are independently destroyed when they are no longer needed.

If a single PE runs out of heap space, the entire computation may deadlock. Thus, some emergency arrangement for claiming space from neighbouring PEs may be justified if the architecture can support remote memory access at all efficiently.

Observe that after computation has begun, two senders pointing to the same expression can evolve so that they point to different places in the stream being computed. Thus they do correspond to our original notion of a process.

5.4.4 Proto-channels, channel creation and channel deletion

Although the process network remains static, the channels linking a pair of PEs can be created at run-time. For simplicity we will assume that they are all created at run-time, although an optimising compiler will perform some channel creation at compile-time in order to uncover other optimisation opportunities.

A channel is created when a process makes a reference to an expression which has been placed on another PE. To achieve this, each such reference in the original program is replaced by a special object, a *proto-channel*. A proto-channel can be implemented as a box (see section 3.1.7), containing a pointer to code in the run-time system together with a representation of the name of the expression referred to.

When a proto-channel is evaluated, the run-time system sets up a channel to the appropriate PE (presumed nearby). A sender process is created on the other PE, and a receiver object is created on the current PE. The initiating process is blocked until a response from the sender arrives. When the first CONS cell of the stream is transmitted by the sender to the receiver, the initiating process is re-awakened, and the original box representing the remote value is replaced by a CONS cell containing the value received (the hd of the cell) together with a reference to the receiver (as the tl of the cell. This reference to the receiver is another special object, a *channel reference*, also represented as a box.

When the tl of the CONS cell is needed, the receiver is interrogated. If it has a value already in its buffer, this is built into a new CONS cell and returned, and a token is sent back to the sender to signify that the buffer space is available. If the value is not available the process is suspended until it is received.

Channel deletion and garbage collection

Channels are collected during the normal process of garbage collection local to each PE. An object is not garbage if and only if it is referred to by some existent sender. A channel is deleted when its receiver becomes garbage. A conventional single-processor garbage collection scheme can be used. The only addition which might be required is some means to trigger garbage collection on other PE's in the hope of freeing space locally.

5.4.5 Representation of stream elements

Thus far we have considered streams as just chains of CONS cells. We have ignored the hd components, the actual values being carried. In the absence of any strictness information, these values must be passed unevaluated, as pointers (in fact proto-channels) to suspended function applications (i.e. boxes) located on the sending PE. In the case of tuples, we can carry pointers to suspensions of each tuple element.

When the receiving function needs the head of a CONS cell it has received, it will find a proto-channel, a sender will be spawned on the sending PE and the value will be sent over a new channel.

This is unfortunate for two reasons:

1. It incurs a large overhead compared with the sequential implementation.

2. If the proto-channel is passed on unevaluated to a third PE, a non-local communication channel could be needed when the object is finally evaluated.

The first problem is unfortunate, but the second is intolerable. We must insist that no proto-channel is ever sent over a channel to a PE which is not a neighbour of its home. It must either be evaluated and used by the receiving process, or it must be discarded.

A partial solution, at least, is to be found in recent work on strictness analysis of list programs, for example by Burn [Bur87a] and Wadler [Wad87]. Alternatively, the programmer could be required to introduce assertions about strictness in order to constrain the evaluation order.

Non-stream communications

We must be able to use a channel to pick any object from another PE – not just a stream, but also scalars, tuples, trees, vectors, matrices and so on. We have concentrated on streams because they form the natural incarnation of a communication link in the functional programming language. The reason is that there is just one order in which to examine the CONS cells which form a list. With trees, vectors and matrices there are many orders in which to traverse the data structure. We are forced to take a very conservative approach.

In the case of a vector or matrix, the simplest solution is to send a vector or matrix of proto-channels. This leaves the elements unevaluated until the receiver needs them. If the elements are streams, pipeline parallelism can be exploited by computing elements eagerly as usual. If the elements are scalars, the parallelism available may be very limited.

For a tree, we send a constructor, e.g. NODE, with proto-channels as its parameters.

Unfortunately, this conservative approach allows no producer-consumer, pipeline parallelism unless streams are involved at some point. This does seem the only predictable and controllable choice. Although a more eager scheme is desirable in many cases, it can often have a very bad effect, concentrating PE power away from the tasks most urgently at hand.

5.4.6 Multitasking

It should now be clear that each PE must be multiplexed between the various sender processes placed upon it. A slight complication here is that the processes must share the PE fairly: none can be allowed to monopolize the PE indefinitely. Thus, processes must be time-sliced. The different processes will often share common sub-expressions, so some synchronisation control must be imposed to prevent evaluation being attempted by several processes of the same expression simultaneously.

It could be quite straightforward to employ tightly-coupled multiprocessors as PEs in our loosely-coupled network, as the synchronisation necessary for time-sliced multitasking is sufficient to synchronise multiple truly-parallel processes.

5.4.7 Communications optimisations

What has been described applies to the general case. In particular examples many optimisations can be applied. We have already seen how strictness analysis can avoid the need for passing proto-channels over channels by evaluating objects before sending them.

If a stream has only one channel consuming it, the sender's code can be compiled inside the expression, so that instead of building a CONS cell, the value is sent directly along the channel. This is the starting point for a series of powerful optimisations. To begin with, one can extend the applicability by observing that if the channel has several consumers who can all receive their values in lock-step, then they can share the output of a single sender process. If the sender is the only process on its PE, and the consumers are alone on their PEs too, then it may be possible to optimise out the channel synchronisation. The neighbouring PEs simply swap values on agreed clock ticks. This is elaborated by Bailey, Cuny and MacLeod in [BCM87].

In "neighbour-coupled" machines, where access to a neighbour's local memory is almost as efficient as to a PE's own, much copying can be avoided. A channel need carry only a pointer to the object being transferred. A copy must still be made if the object is forwarded to a third PE.

An optimisation for the general case might be to check that an object has not already been copied to this PE before following a proto-channel to another to get it. This can be done using a hash table, as used for the same purpose in the FLAGSHIP parallel graph reduction machine [WSWW87].

Finally, one might hope that hardware or microcode support for the communication and scheduling operations might be provided. The overheads of discovering a proto-channel, setting up the link, spawning the sender and waiting for a value are a serious threat to the feasibility of the scheme when non-stream objects are communicated.

5.5 A simple guide to the effect of arc

To finish the discussion of implementation strategies, we give a description of the effect of the arc assertion. Suppose we have a program fragment summarized by

```
x = output
    where
    output = f a
    a = g b
    moreover
    (arc output a)
    ∧ ...b ...
```

Figure 5.3: The process network for example x

- **Placement of output:** The result, output, will be delivered to a PE at or next to the PE where it is required. Thus, if the output is to be displayed on a graphics device, it will be placed on or adjacent to a PE with access to the graphics hardware.
- **Placement for input:** If input is required, it will be manifest as a free variable named in the **moreover** clause, such as b in the example. The process network will be placed so that its communication with the producer of b is neighbour-to-neighbour.
- **Partitioning of components:** The expression output is placed on a PE of its own, together with all the expressions to which it refers – except a, which is placed elsewhere. For example, if f is a function, the code for f is copied to output's PE. Similarly, a is placed on a different PE, of its own, together with a copy of g.
- **Placement of components:** The PE carrying output is chosen so that it enjoys neighbour-to-neighbour communications with the PE carrying a.
- **Evaluation parallelism:** The PEs carrying output and a interact because the expression output refers to a. If a is a scalar or a list, its evaluation will proceed in parallel with output.

  If a is vector, matrix or tree, evaluation of a will not begin until output demands it. At that point, output will be blocked, waiting for a's value. This precludes pipeline parallelism, but horizontal parallelism may yet keep the PE's usefully employed.

  However, vertical parallelism can still occur if a returns a structure containing streams, which are examined by output.

Finally, note that output may not be able to proceed in parallel with a if a depends on x. If a is a stream, and successive elements depend on one another due to such a recursive stream definition, then the pipeline parallelism available may be restricted by the span of this dependency. We return to this point in section 5.10.

The process network is illustrated in figure 5.3.

5.6 Semi-static process networks

A Caliban program may specify a process network which is not entirely determined at compile-time. In the simplest case, as with the size of the matrix in the ApplyLNO example, the network might depend on just one parameter. We could delay compilation until this parameter is known, and then generate a specialised version to apply to the remaining parameters.

Another example might be the ray tracer, where the length of the ray intersection test pipeline depends on the number of objects in the object database. In fact, of course, the length of the pipeline is limited by the number of PEs available. We could write a definition of the distributed ray tracer like this:

```
RayTracer objects viewpoint
 = map EvaluateTree
       (StreamToListOfMTrees (length initialrays)
                             (StreamOfContributoryRayTrees initialrays))
   where
   StreamOfContributoryRayTrees rays
    = output
      where
      (output, feedback)
       = (Split
          ∘ join
          ∘ (map LayerOf')
          ∘ (map TakeImpact)
          ∘ IntersectionPipelineComponents
          ∘ (map MakePipeItem))
         (rays ++ feedback)
   where


   IntersectionPipelineComponents
    = (insert (∘) ident
              (map map (map PipelineStage objects)))

   partitions = map BUNDLE
                (PartitionList (NumFreePEs-1)
                               (map (□) IntersectionPipelineComponents)
   moreover
   (chain partitions)
   ∧ (arc output (hd partitions))
   ∧ (arc (last partitions) output)
```

where NumFreePEs is the number of available PE's. We reserve one PE for output, responsible for the join, map LayerOf' and map TakeImpact processes. The "++" and map MakePipeItem operations are automatically collected in the last component of Intersection-

PipelineComponents. The function PartitionList n xs gathers the list xs into a list of n lists:

```
PartitionList n xs
 = PartitionListWithSize bitsize xs
   where
   bitsize = ceiling ((length xs) / n)
   PartitionListWithSize bitsize
    = (take bitsize xs)
        : (PartitionList n (drop bitsize xs))
```

where ceiling x yields the smallest integer larger that the real number x. The constructor BUNDLE is mapped over partitions to indicate that the list is for bundling purposes only.

This program is partitioned automatically to make use of just the resources available. A slightly more subtle version might first ensure that the object database is big enough to justify distribution over NumFreePEs-1 with a worthwhile grain size.

Kedem and Ellis give an interesting example [KE84] of a program for parallel ray-casting whose process network depends on the structure of the object database in a much more complicated way. The database takes the form of an expression in the algebra of Constructive Solid Geometry (CSG). The expression's shape varies from problem to problem and is typically quite a severely unbalanced tree. They employ an efficient embedding algorithm to map this tree into their architecture's mesh of PEs at the beginning of each computation.

It might be quite reasonable for a computation to go through a series of phases, each requiring a different process network. The communications would be reconfigured after each phase, giving the effect of an evolving network. This has not yet been captured in the Caliban network description language.

5.7 Dynamic process networks

It is quite possible to write down Caliban programs whose process network is not determined until all parameters are present. An example might occur in computational fluid dynamics, where the grid is refined between iterations to cover regions of turbulence more finely. The computation would start with a small mesh of PEs, but as areas of interest are detected, and the grid is refined, more PEs could be called in to cover regions showing poor convergence. Some PEs might finish their tasks early. They could make themselves available to be reused at another point of the grid.

We have an algorithm of the form

```
solve f a
 = generate EachMatrix
   where
   EachMatrix 0 = a
   EachMatrix (i+1)
    = MakeMatrix Bounds EachElement
      where
      EachElement
       = solution
         where
         PointSolution = ...
         solution = PointSolution, if PointError ≤ ε
         solution = solve f submatrix, otherwise
         submatrix = ...
```

Much has been simplified here. The important point is that solve is occasionally called recursively on a smaller mesh submatrix (using a refined grid). Depending on the solution scheme (i.e. how PointSolution is defined), several possible process networks might be used. Let us suppose a mesh is used: we might write the **moreover** clause

```
moreover
mesh interface
```

We find that the process network can develop into a mesh-shaped tree of meshes. This interesting area is not covered by the explanation given here of how Caliban programs might be implemented , but could prove fruitful on a class of more tightly-coupled machines where locality is still important.

5.8 Related Work

Caliban builds on a fast sequential implementation of a functional programming language, as presented by [Jon87]. It should be contrasted with dynamic-schedule approaches to parallel implementation of functional languages, for example as proposed in chapter 24 of Peyton Jones' textbook [Jon87] (parallel graph reduction), and in [AN87] (dataflow).

5.8.1 Occam

Caliban's aims are similar to those of Occam [PM87]. It differs in three principle respects:

1. **Functional base language.** Caliban inherits the expressive power of a full, lazy, higher-order functional language, along with its highly dynamic store use.

   Caliban retains the functional base language's very simple and attractive transformation properties. Like all functional languages, Caliban pays for its theoretical simplicity with its innate determinacy: a process cannot make decisions based on the order of completion of subtasks. This limits the applicability of the language.

2. **Abstract Networks.** Caliban does not demand that the programmer's process network be explicitly mapped to physical channels. That responsibility is devolved to the compiler – more precisely, to the post-compilation mapping phase. There would appear to be no reason, in principle, why an Occam implementation should not do the same.

3. **Dynamic networks.** Caliban facilitates the description of process networks whose size, and possibly form, cannot be determined until at least some parameters are present. This can be used to describe run-time dynamic networks, or, perhaps more interestingly, to express a family of process networks for different choices of particular parameters. A good example of such a program is given in [KE84].

   Occam does not allow dynamic networks, although simple parameters such as a processor array's size can be given as a manifest constant. This restriction was imposed, however, solely to simplify implementation: [May87] shows how a variant of Occam with recursion could describe a tree of processes.

To summarise, Caliban fulfils a very similar role to Occam, and promises similar performance, but offers an improvement in expressive power, and a basis for more powerful program transformation and verification techniques. Caliban is substantially more reliant than Occam on advanced compiler technology.

5.8.2 "Para-Functional" Programming

This is an extension to the lazy functional programming language "ALFL" proposed by Paul Hudak in [Hud86b], called "PARALFL". An expression "e" can be annotated by a second expression, "p", whose value indicates the PE on which "e" is to be executed:

```
e $on p
```

To simplify use of this mechanism, the value of the reserved identifier "$self" is defined to be the index of the PE upon which the expression concerned is executed. This can be used in "e" as well as in "p", thus subverting referential transparency. PE's are indexed by integers. Related ideas appear in [KL82] (by Keller and Lindstrom) and [Bur84b] and [Bur87b] (by Burton).

Hudak's approach has the merit of simple implementation. The notion of a program's "process network" – which lies at the root of the Caliban approach – seems to be well hidden in the text of a para-functional program. Caliban offers some abstraction here, by leaving responsibility for mapping of a logical process network to actual processors with an automatic post-compilation phase.

Caliban also makes some attempt to ensure that all process interactions appear explicitly in the program script. PARALFL has no such aspiration, with the result that unexpected interdependencies between processes mapped to distant processors could mean very disappointing performance.

5.8.3 Flo

This parallel functional programming language was developed by Floating Point Systems Inc. and is described in [You85]. Flo is based on Backus' FP [Bac78], augmented with streams and a reverse function composition operator "→". It is aimed at providing a high-level language for building autonomous parallel applications programs running on their proprietary scientific co-processors. These are micro-programmed vector processors offering a variety of built-in high level functions, such as the Fast Fourier Transform and matrix multiplication. Flo is responsible for partitioning large problems between multiple PE's.

Flo includes various operators for partitioning arrays, distributing them over pools of PE's, and collecting the results. A very simple example which they give is the function "fun" (whose type specification has been omitted):

```
DEFINE fun = f → [OnAny(D1) → g, OnAny(D2) → h]
```

This is functionally equivalent to our definition

```
fun x = (g y, h y)
        where
        y = f x
```

The "OnAny" operator results in a change of "context": the input stream to "g" is copied to a PE selected from D1 (a set of PE identifiers), where "g" is applied. Run-time mechanisms select PE's as available.

Flo is interesting in being motivated by practical concerns raised by trying to run scientific applications very fast on fairly conventional hardware. The limited control over process placement probably derives from the use of a bus as the communications medium: each PE is effectively equidistant from all the others. This is made more feasible by the use of PE's with substantial fundamental operations, making large granularity easy to achieve.

5.8.4 Graph Grammar-based Specification of Interconnection Structures

This work, reported in [BC87], aims to simplify the description of process network families in an interactive parallel program development environment. A graph family arises when a parallel algorithm is designed to be portable between similar architectures of differing sizes – for example, after testing on a small program development configuration. It is necessary to give a formal description of how the graph generalises for larger configurations.

Details of the approach are rather complex, and merit more thorough study. The formalism employed, a restriction of aggregate-rewriting graph grammars to allow only three kinds of rewrite, would appear to be at least as powerful as Caliban. The use of graph grammars explicitly offers the prospect of a sound theory for embedding program-generated graphs in physical communications networks.

Bailey and Cuny may gain some additional expressive power over Caliban by completely separating the way the graph is constructed from the program's recursive structure.

5.9 Future Research

Much of this chapter has been devoted to preliminary investigation of areas deserving more extensive study. Here, some of the more interesting areas are summarised:

Compilers

The development of a pilot implementation of Caliban is the most pressing next step. A transputer network is a very attractive target architecture.

Dynamic networks may stretch the communications capabilities of present-day transputers. Caliban's dynamic networks can make use of software-configurable networks with relatively long switching times, which may fit well with novel technologies, based, for example, on optics.

Programming Environments

Caliban's design is based on the hypothesis that process networks are a useful way for a programmer to think about parallel algorithms. They demand graphical presentation. A Caliban programming environment could illustrate a program's process network, perhaps showing traffic levels on arcs and load levels on bubbles, derived from simulation statistics. Program transformation and analysis tools could be included, including strictness analysis, cycle-starvation analysis, "granularity" analysis (identifying processes which may have a high communication-to-computation ratio) etc.

Semantics

Giving a mathematical explanation of what Caliban's annotations mean is an interesting area. Hudak has given an "execution tree" semantics for PARALFL [Hud86a], and Williams [Wil88] has extended and refined this for a Caliban-like language. Caliban seems to demand a richer domain of semantic values, to include arbitrary, possibly cyclic, graphs. The graph grammar approach taken by Bailey and Cuny may prove of use.

As mentioned earlier, of most value would be a semantics which assigns a process network family to a function, corresponding to the networks which might result for different parameter choices.

5.10 Pointers into the literature

Communication in parallel algorithms

Vitanyi [Vit86], Feldman and Shapiro [FS88] and others have investigated the constraints imposed by the physical universe on communication in parallel computations (although

Deutsch [Deu85] suggests the real world admits more possibilities than presently exploited). The importance of such arguments when computer manufacture can employ all three dimensions for wiring can be disputed, although heat dissipation places a limit on three dimensional packing density.

However, the situation is much clearer in two dimensions, and VLSI complexity theory addresses the problem of accounting for communication in algorithm design and analysis with considerable success. A good introductory work is Ullman's textbook [Ull84]. Although there are many different VLSI complexity measures, they all take the area of the wiring into account as well as the number of "active" data operations performed. Some theories also account for the signal propagation delay in the wiring, whose length is determined by the layout and size of the circuit.

It is clear that good VLSI algorithms make good algorithms for loosely-coupled multiprocessors – in fact one might think of a loosely- or neighbour-coupled multiprocessor as a "universal" VLSI machine, being programmable to implement any algorithm with similar behaviour but at a substantial interpretation overhead. One might expect that this overhead would mask some of the importance of communications connectivity. The question of the existence and nature of a universal parallel computer is the subject of continuing work by Valiant, see for example [Val81].

The nature of parallel programming

Much has been made of the "von Neumann bottleneck", a term coined by Backus [Bac78]. Backus argues that programming in an imperative style imposes the presence of a single word-at-a-time memory access path on the design of programs – reducing programming to the scheduling of traffic to and from memory. That conventional high-performance von Neumann machines need have no such bottleneck is not as important as the damage done by the von Neumann model of computation to program design. Sutherland and Mead [SM77] extend this argument in a substantial way: they argue that sequential computation has artificially dominated the study and teaching of computer science from its beginning. The technological accident responsible for this, that switching has been more expensive and slower than wiring, they argue, is no longer valid – and yet its heritage is still with us.

An example Sutherland and Mead use concerns the parallel interchange sort algorithm, where a PE is responsible for every element, and neighbours swap if their elements are out of order. This algorithm is expensive in sequential terms, because to sort n elements it requires $O(n^2)$ (order n^2: proportional to n^2 when n becomes large) comparisons as opposed to Quicksort (see section 4.2.1) which needs only $O(n \log_2 n)$ comparisons in the average case. However, Quicksort involves global communications at every step, while the interchange sort involves neighbours only. We can expect the interchange sort to give by far the better performance for a large range of cases, although when sorting a very large data set Quicksort must win.

Of course there are far better parallel sorting algorithms; see Ullman [Ull84] and the classic work by Thompson ([TK77] and [Tho81]).

Partial evaluation

The simplification process by which a normal-form Caliban program is derived from one using the abstraction mechanisms is a kind of partial evaluation. Partial evaluation is the application of a program to some but not all of its parameters, so that simplifications can be made to save time when subsequent parameters are provided. It was pioneered by Ershov [Ers82], who called it "mixed computation". Jones' group at Copenhagen have made great progress in understanding the structure of a partial evaluator, and the simplifications and analyses possible. Most exciting has been their construction of a self-applicable partial evaluator, MIX, and its application to compiler generation: MIX takes a program, and its first parameter, and generates a new, simplified program such that

```
(MIX p a) b = p a b
```

They apply MIX to an interpreter Int for another programming language, l. Int takes a program in l, p_l, and its input:

```
Int p_l input
```

It yields the output resulting from running the program p_l on input. Now we can get a compiled implementation of p_l by evaluating

```
MIX Int p_l
```

Thus, the partial application MIX Int plays the part of a compiler. In fact the specialised compiler is generated by

```
MIX MIX Int
```

Repeating the process, we can observe that the partial application MIX MIX plays the part of a compiler generator – when given an interpreter it produces a compiler. Thus, we can generate the compiler generator by writing

```
MIX MIX MIX
```

The idea is attributed to Futamura [Fut71].

More practically directed applications include ray-tracing, which has been investigated by Mogenson [Mog87]. Mogenson takes a simple ray-tracer (more realistic than the implementations given here), and partially-evaluates it with details of the scene but not the viewpoint. The residual program can then be applied to different viewpoints, and a considerable saving is observed.

Starvation and Deadlock

Recall the function to compute the list of Fibonacci numbers:

```
fibs = 1 : 1 : (map2 (+) fibs (tl fibs))
```

This recurrence is too trivial for real parallelism, but suppose we could decompose map2 (+) into a pipeline,

```
xs = 1 : 1 : (map f (map g (map2 h xs (tl xs))))
```

Now there might seem to be pipeline parallelism available. However, closer inspection reveals that each element xs **sub** n depends on the immediately preceding element of the stream, xs **sub** (n-1). There can be only a single locus of computation in the cycle.

We can increase the amount of parallelism available by increasing the data dependency gap. For example, the (different!) program

```
ys = 1 : 1 : 1 : (map f (map g (map2 h ys (tl ys))))
```

has two loci of computation. The amount of parallelism could depend on a run-time variable, as in

```
f initialvalues = xs
                  where
                  xs = initialvalues ++ (map f (map g (map2 h xs (tl xs))))
```

This actually happens in the cyclic-pipeline implementation of the ray tracer, given in section 4.9.

This program will deadlock: if length initialvalues ≤ 1 there are zero loci of computation. It is important to realise that deadlock in a functional language is quite independent of any parallel activity, and reflects nothing more than a particular form of undefinedness. Semantically, deadlock is indistinguishable from non-termination, $\perp$. Thus, the program transformations in this book can be used to transform a deadlocking program into one which is simply undefined: from

```
fibishs = 1 : (map2 (+) fibishs (tl fibishs))
```

it is not hard to derive the recurrence

```
fibishs sub 0 = 1
fibishs sub (n+1) = (fibishs sub n) + (fibishs sub (n-1))
```

which is clearly ill-founded. Thus, deadlock is a semantic property of a program, unaffected by how the program is distributed, or use of parallel evaluation.

A simple test exists to predict when deadlock will occur, called the *cycle sum test*. A number is calculated for each cycle in the data flow graph, giving the data dependency between successive elements. The cycle sum test was introduced and justified by W.W. Wadge [Wad81] in the context of the dataflow language Lucid. Wadge verifies it by direct reference to the denotational semantics of the language: there is no need for any operational reasoning.

Chapter 6

Epilogue

Much of this book has been concerned with details. It is the rôle of the concluding chapter to regain a broader perspective on what has been achieved, and where the research is going. Let us begin by returning to to Eckert's advice quoted in the introduction, that

> Any steps which are controlled by the operator, who sets up the machine, should be set up only in a serial fashion.

Things have changed a great deal since the time when Eckert was writing. Obviously the capabilities of the hardware have improved, but the hardware's structure has not really addressed the parallel programming problem. Much more significant to the argument have been the advances in software technology: Eckert was writing before the first compiler was written, so did not take into account the possibility that the compiler as well as the hardware could exploit parallelism without the programmer being involved. The functional approach to parallel programming takes this idea to its limit, by removing the step-by-step imperative perception of program execution completely. The line is drawn at programs whose behaviour is non-deterministic—which is where the trouble Eckert refers to really starts. The happy coincidence which forms the basis of this book is that this class of languages is also very easy to manipulate mathematically.

There are alternative approaches. One of the most common is simply to ignore Eckert's problem, and employ a great deal of care and discipline in writing explicitly-parallel multiprocessor programs. This approach may be rescued by the advent of simple mathematical systems of reasoning about parallel programs in the general, non-deterministic case. Research aimed at providing such a system is very much still in progress. Another area of recent success has been the development of parallelising compilers for imperative languages.

Meanwhile, the problem of ensuring the correctness of computer programs has become more and more acute. This is especially interesting in application areas like natural and medical science, where technology has made the computer a ubiquitous tool, and often means that a computer program is not just a test of a scientific model—but is the only tested model in existence. The problem is understanding such models, testing them, and presenting them in the literature. However, formality must be tempered by the need to build and use computer programs quickly, and to run them very fast. This demands a high level of computer support and very well-designed and well-explained tools and documentation.

This book is a manifesto for a programme of research aimed at making derivation and transformation of computer programs an accessible and effective tool to enable the non-specialist to produce more reliable computer programs more quickly.

Appendix A

Proofs and Derivations

In the body of the book it has been useful to state several laws relating expressions in the functional notation. Rather than interrupt the narrative flow, their proofs appear in this appendix.

A.1 ListToTree and TreeToList, simple versions

This is an example of a common requirement during a program transformation by data type transformation—that we can get the original representation back. It was used to produce a divide-and-conquer implementation of map. We have some auxiliary functions:

```
take n (a:as) = a : (take (n−1) as),        if n ≠ 0
take n [ ] = [ ],                           if n ≠ 0
take 0 as = [ ]
```

and

```
drop n (a:as) = drop (n−1) as,              if n ≠ 0
drop n [ ] = [ ],                           if n ≠ 0
drop 0 as = as
```

A list is converted into its binary tree representation by the function ListToTree1:

```
ListToTree1 [ ] = EMPTY
ListToTree1 [a] = LEAF a
ListToTree1 (a0:a1:as) = NODE (ListToTree1 (take m (a0:a1:as)))
                              (ListToTree1 (drop m (a0:a1:as)))
                         where
                         m = (length (a0:a1:as))/2
```

To convert it back to a list again, we have

```
TreeToList1 EMPTY = [ ]
TreeToList1 (LEAF a) = [a]
TreeToList1 (NODE subtree1 subtree2) = (TreeToList1 subtree1)
                                       ++ (TreeToList1 subtree2)
```

We have, for all n and as,

```
(take n as) ++ (drop n as) = as
```

This can be shown using total structural induction on n. The proof is omitted in case the reader should attempt it as an exercise. Theorem 1 is our main concern:

Theorem 1 *We require that for all finite, total lists* as*,*

```
TreeToList1(ListToTree1 as) = as
```

The proof uses total structural induction, but unfortunately the standard ordering on lists doesn't do the job. Instead we employ a "bisection" ordering, with basis [a] and

```
as ++ bs ≻ as
```

and

```
as ++ bs ≻ bs
```

for any as, bs $\neq$ [ ]. The case of empty as must be shown separately, but is trivially satisfied.

Proof:

By total structural induction on the bisection ordering.

Empty case:
```
TreeToList1(ListToTree1 [ ]) = TreeToList1 EMPTY
                             = [ ]
```

as required.

Base case:
```
TreeToList1(ListToTree1 [a]) = TreeToList1 (LEAF a)
                             = [a]
```

as required.

Inductive Step: Assuming that for all finite and total as and bs,

```
TreeToList1(ListToTree1 as) = as
```

and

TreeToList1(ListToTree1 bs) = bs

we must show that

TreeToList1(ListToTree1 (as ++ bs)) = (as ++ bs)

Apply reduction to the LHS:

```
LHS = TreeToList1 (ListToTree1 (as ++ bs))
    = TreeToList1 (NODE (ListToTree1 (take m (as ++ bs)))
                        (ListToTree1 (drop m (as ++ bs))) )
      where
      m = (length (as++bs)) / 2
```

We can define

cs = take m (as ++ bs)

and

ds = drop m (as ++ bs)

So we have

```
LHS = TreeToList1 (NODE (ListToTree1 cs)
                        (ListToTree1 ds))
      where
      m = (length (as++bs)) / 2
    = (TreeToList1 (ListToTree1 as))
      ++ (TreeToList1 (ListToTree1 bs))
```

Our inductive assumptions hold for any choice of as and bs, so this is

LHS = cs ++ ds = (take m (as ++ bs)) ++ (drop m (as ++ bs))

Using the property of take, drop and "++" claimed earlier, this is just

LHS = (as++bs)

as required.

A.1.1 Removing the inefficiency

ListToTree1 and TreeToList1 are inefficient for several reasons:

- the length of the parameter list is calculated at each recursion.
- both take and drop scan the input list at each recursion.
- the append operator "++" scans and reconstructs its left parameter.

We can use program transformation to remove each of these inefficiencies.

Removing the length recalculation

First let us define a version of ListToTree1 which calculates the length of its input list. The intention is that

```
ListToTree1' as n = ListToTree1 as
where
n = length as
```

Define:

```
ListToTree1' [ ] 0 = EMPTY
ListToTree1' [a] 1 = [a]
ListToTree1' (a0:a1:as) n = NODE (ListToTree1' (take m (a0:a1:as)) m)
                                 (ListToTree1' (drop m (a0:a1:as)) m)
                            where
                            m = n/2
```

Now we redefine ListToTree1 to use this modified version:

```
ListToTree1 as = ListToTree1' as (length as)
```

It is easy to verify the equivalence using recursion induction.

Removing the re-scanning in take and drop

Let us define

```
split n as = (take n as, drop n as)
```

Now we can re-express ListToTree1' to use it:

```
ListToTree1' [ ] 0 = EMPTY
ListToTree1' [a] 1 = [a]
ListToTree1' (a0:a1:as) n = NODE (ListToTree1' front m) (ListToTree1' back m)
                            where
                            (front, back) = split m (a0:a1:as)
                            m = n/2
```

We can derive a more efficient version of split. Instantiate its definition for a non-empty parameter list:

```
split n (a:as) = (a: (take (n−1) as), drop (n−1) as),          if n ≠ 0
```

But we can rewrite this as another instance of split:

```
split n (a:as) = (a: front, back),          if n ≠ 0
                 where
                 (front, back) = split (n−1) as
```

All that remains is to derive the equation for the other cases. For the empty list case:

```
split n [ ] = (take n [ ] , drop n [ ]),      if n ≠ 0
            = ([ ], [ ]),                     if n ≠ 0
```

When n = 0,

```
split 0 as = (take n as , drop n as)
           = ([ ], as)
```

Avoiding reconstruction in "++"

The final transformation avoids the use of "++", which is inefficient because it must always make a copy of its left parameter. Instead we use the list constructed by the left parameter, but modify the left parameter expression so that the right parameter is placed on the end instead of [ ]. The property we exploit is

```
(f x) ++ (g y) = f' x (g y)
```

where

```
f' x as = (f x) ++ as
```

The optimisation comes by applying equational reasoning to the definition of f' above, so that the "++" is not needed. This optimisation is straightforward enough to be considered for inclusion as an automatic process in optimising compilers (the interested reader might compare it with the use of difference lists in Prolog, as introduced by Clark and Tarnlund

[CT77]. In our case, we define

```
TreeToList1' tree rest = (TreeToList1 tree) ++ rest
```

Note that

```
TreeToList1 tree = TreeToList1' tree [ ]
```

Instantiate the definition of TreeToList1' for the empty tree:

```
TreeToList1' EMPTY rest
        = (TreeToList1 EMPTY) ++ rest
        = [ ] ++ rest
        = rest
```

Now instantiate it for the LEAF case:

```
TreeToList1' (LEAF a) rest
        = (TreeToList1 (LEAF a)) ++ rest
        = [a] ++ rest
        = a : rest
```

(note that this is where the "++" disappears). The NODE case is the most complicated:

```
TreeToList1' (NODE subtree1 subtree2) rest
        = (TreeToList1 (NODE subtree1 subtree2)) ++ rest
        = (TreeToList1 subtree1) ++ (TreeToList1 subtree2) ++ rest
```

At this point we can use the definition of TreeToList1', backwards:

```
TreeToList1' (NODE subtree1 subtree2) rest
        = (TreeToList1 subtree1) ++ (TreeToList1' subtree2 rest)
```

And now do the same to the whole RHS, getting rid of "++" altogether:

```
TreeToList1' (NODE subtree1 subtree2) rest
        = (TreeToList1' subtree1 (TreeToList1' subtree2 rest))
```

(This optimisation of "++" can be incorporated as a compiler optimisation, and Wadler has characterised where it can be applied [Wad88a]. It is a form of linearisation; see Field and Harrison [FH88]).

This completes our optimisation process. Collecting the results, we have ListToTree1:

```
ListToTree1 as = ListToTree1' as (length n)
```

```
ListToTree1' [ ] 0 = EMPTY
ListToTree1' [a] 1 = [a]
ListToTree1' (a0:a1:as) n = NODE (ListToTree1' front m) (ListToTree1' back m)
                            where
                            (front, back) = split m (a0:a1:as)
                            m = n/2
```

where

```
split 0 as = ([ ], as)
split 0 [ ] = ([ ], [ ]),                    if n ≠ 0
split n (a:as) = (a: front, back),           if n ≠ 0
                 where
                 (front, back) = split (n−1) as
```

and TreeToList1:

```
TreeToList1 tree = TreeToList1' tree [ ]

TreeToList1' EMPTY rest = rest

TreeToList1' (LEAF a) rest = a : rest

TreeToList1' (NODE subtree1 subtree2) rest = (TreeToList1' subtree1
                                                (TreeToList1' subtree2 rest))
```

A.2 ListToTree and TreeToList, shuffled versions

In this section we show the correctness of an alternative approach to the problem of representing a list as a binary tree. In this version each node has all the even-indexed elements of the list it represents to its left, and all the odd-indexed ones to its right. We make use of some auxiliary functions, whose rôle here is analogous to the rôles of take, drop and "++" in the straightforward versions given in the previous section:

```
EvenOnes [ ] = [ ]
EvenOnes [a0] = [a0]
EvenOnes (a0:a1:as) = a0:(EvenOnes as)


OddOnes [ ] = [ ]
OddOnes [a0] = [ ]
OddOnes (a0:a1:as) = a1:(OddOnes as)
```

```
merge (a0:evens) (a1:odds) = a0:a1:(merge evens odds)
merge as [ ] = as
```

The functions we are interested in are:

```
ListToTree2 [ ] = EMPTY
ListToTree2 [a] = LEAF a
ListToTree2 (a0:a1:as) = NODE (ListToTree2 (EvenOnes (a0:a1:as)))
                              (ListToTree2 (OddOnes (a0:a1:as)))
```

and

```
TreeToList2 EMPTY = [ ]
TreeToList2 (LEAF a) = [a]
TreeToList2 (NODE evensubtree oddsubtree)
            = (merge (TreeToList2 evensubtree) (TreeToList2 oddsubtree))
```

That they satisfy the specification as required is the subject of the next theorem:

Theorem 2 *For all finite and total lists* as*,*

```
TreeToList2 (ListToTree2 as) = as
```

The natural approach for the proof is total structural induction. The base case as = [ ] is trivial. The obvious inductive step is to show the property for a:as assuming it for as. This fails, and the reason is not hard to find. The algorithm looks *two* elements ahead into the input list (via EvenOnes and OddOnes). A better inductive step is to assume the property for as and a1:as, and try to show it for a0:a1:as. This is valid provided we make sure all possible values for as are covered—a special proof must be given for as = [a].

It is surely no coincidence to find that the definition of ListToTree2 does indeed deal with the [a] case specially!

Proof

By total structural induction on the length of the list as.

Base cases: Trivial for both as = [ ] and as = [a].

Inductive step: Assuming that for all finite and total lists as,

```
TreeToList2 (ListToTree2 as) = as
```

and

```
TreeToList2 (ListToTree2 (a1:as)) = a1:as
```

we must show that

TreeToList2 $\underbrace{\text{(ListToTree2 (a0:a1:as))}}$ = a0:a1:as

Apply reduction to the LHS:

$$\begin{aligned}\text{LHS} &= \text{TreeToList2 (NODE (ListToTree2 } \underbrace{\text{(EvenOnes (a0:a1:as)))}} \\ &\qquad\qquad\qquad\qquad \text{(ListToTree2 } \underbrace{\text{(OddOnes (a0:a1:as))))}} \\ &= \underbrace{\text{TreeToList2 (NODE (ListToTree2 (a0:(EvenOnes as)))}} \\ &\qquad\qquad\qquad\qquad \underbrace{\text{(ListToTree2 (a1:(OddOnes as))))}} \\ &= \text{merge } \underbrace{\text{(TreeToList2 (ListToTree2 (a0:(EvenOnes as))))}} \\ &\qquad \underbrace{\text{(TreeToList2 (ListToTree2 (a1:(OddOnes as))))}}\end{aligned}$$

The underbraced expressions here are instances of our second inductive assumption, giving

$$\begin{aligned}\text{LHS} &= \underbrace{\text{merge (a0:(EvenOnes as)) (a1:(OddOnes as))}} \\ &= \text{a0:a1:}\underbrace{\text{(merge (EvenOnes as) (OddOnes as))}}\end{aligned}$$

By Lemma 1, this is simply a0:a1:as as required.

The alert reader will realise that Lemma 1 has not yet been exhibited. However, it does seem to be crucial to the algorithm's operation. We've succeeded in reducing our original problem so that all remains is this lemma—an activity much like stepwise refinement of programs. Fortunately, the lemma is easily proved:

Lemma 1 *For all finite and total lists* as*,*

merge (EvenOnes as) (OddOnes as) = as

Proof

By total structural induction on the length of as.

Base cases: Trivial for both as = [ ] and as = [a].

Inductive step: Assuming that for all finite and total lists as,

merge (EvenOnes as) (OddOnes as) = as

and

merge (EvenOnes (a1:as)) (OddOnes (a1:as)) = a1:as

we must show that

merge $\underbrace{\text{(EvenOnes (a0:a1:as))}}$ $\underbrace{\text{(OddOnes (a0:a1:as))}}$ = a0:a1:as

Apply reduction to the LHS:

$$\begin{aligned}\text{LHS} &= \underbrace{\text{merge (a0:(EvenOnes as)) (a1:(OddOnes as))}} \\ &= \text{a0:a1:}\underbrace{\text{(merge (EvenOnes as) (OddOnes as))}}\end{aligned}$$

Our first inductive assumption applies here, to give a0:a1:as as required.

We did not need the second inductive assumption here. This is somewhat disturbing—often a sign of some error. But curiously, in the proof of Theorem 2, we didn't use the first inductive assumption: between the two proofs we did eventually discharge both assumptions.

A.3 Turning recurrences into cyclic networks

In introducing the functional language employed in this book, an idiom was employed for recurrences—what in an imperative language would simply be written as a loop. Although very clear and concise, this idiom has an inefficiency because of the use of the **sub** operator to select values from previous iterations. In this section we transform such a recurrence into a cyclic process network formulation. This removes the use of the **sub** operator, and also elucidates some potential parallelism. We work with the Newton Raphson example.

We solve for $fx = 0$ with $f'x = \frac{d(\text{f x})}{dx}$, and using an initial estimate x_0:

xs **sub** 0 = x_0
xs **sub** i = (xs **sub** (i−1)) − (f (xs **sub** (i−1)) / f' (xs **sub** (i−1))), **if** n ≥ 1

with the implementation using the recurrence idiom:

```
solve f f' x0
 = until converges xs
    where
    converges 0 = FALSE
    converges i = abs( ((xs sub i) − (xs sub (i−1)))/(xs subi) ) ≤ ε,   if i ≥ 1
    xs = generate NextEstimate
         where
         NextEstimate 0 = x0
         NextEstimate i = (xs sub (i−1))
                          − ( f (xs sub (i−1)) / f' (xs sub (i−1)) ),   if n ≥ 1
```

We now transform this into a cyclic process network. Unfold generate in the definition of xs:

$$
\begin{aligned}
\mathsf{xs} &= \mathsf{map\ NextEstimate}\ \underbrace{(\mathsf{from\ 0})} \\
&= \underbrace{\mathsf{map\ NextEstimate}\ (\mathsf{0{:}(from\ 1)})} \\
&= \underbrace{(\mathsf{NextEstimate\ 0})}\ :\ (\mathsf{map\ NextEstimate\ (from\ 1)}) \\
&= \mathsf{x_0}\ :\ (\mathsf{map\ NextEstimate\ (from\ 1)})
\end{aligned}
$$

Now we must decompose (the ≥ 1 case of) NextEstimate into the transition function and the indexing function:

$$
\begin{aligned}
\mathsf{NextEstimate\ i} &= (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i{-}1})) - (\ \mathsf{f}\ (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i{-}1}))\ /\ \mathsf{f'}\ (\mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i{-}1}))\ ), \quad \textbf{if } \mathsf{n} \geq 1 \\
&= \mathsf{prevx} - ((\mathsf{f\ prevx})/(\mathsf{f'\ prevx})) \\
&\quad \textbf{where} \\
&\quad \mathsf{prevx} = \mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i{-}1}) \\
&= (\mathsf{Transition} \circ (\mathsf{Index\ xs}))\ \mathsf{i} \\
&\quad \textbf{where} \\
&\quad \mathsf{Transition\ prevx} = \mathsf{prevx} - ((\mathsf{f\ prevx})/(\mathsf{f'\ prevx})) \\
&\quad \mathsf{Index\ xs\ i} = \mathsf{xs}\ \underline{\mathbf{sub}}\ (\mathsf{i{-}1})
\end{aligned}
$$

We can decompose this a little further (recall that $\mathsf{xs}\ \underline{\mathbf{sub}}\ \mathsf{i} = ((\underline{\mathbf{sub}})\ \mathsf{xs})\ \mathsf{i}$):

$$
\begin{aligned}
\mathsf{NextEstimate\ i} &= (\mathsf{Transition} \circ ((\underline{\mathbf{sub}})\ \mathsf{xs}) \circ (\mathsf{subtract\ 1}))\ \mathsf{i} \\
&\quad \textbf{where} \\
&\quad \mathsf{subtract\ n\ m} = \mathsf{m} - \mathsf{n}
\end{aligned}
$$

So now we have

$$
\mathsf{xs} = \mathsf{x_0}\ :\ (\mathsf{map}\ \underbrace{(\mathsf{Transition} \circ ((\underline{\mathbf{sub}})\ \mathsf{xs}) \circ (\mathsf{subtract\ 1}))}\ (\mathsf{from\ 1}))
$$

It is easy to verify (using partial structural induction on the list's length) that $\mathsf{map\ (f} \circ \mathsf{g)} = (\mathsf{map\ f}) \circ (\mathsf{map\ g})$, so that this is

$$
\begin{aligned}
\mathsf{xs} &= \mathsf{x_0}\ :\ \underbrace{(((\mathsf{map\ Transition}) \circ (\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{xs})) \circ (\mathsf{map\ (subtract\ 1)}))\ (\mathsf{from\ 1}))} \\
&= \mathsf{x_0}\ :\ (((\mathsf{map\ Transition}) \circ (\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{xs})))\ \underbrace{(\mathsf{map\ (subtract\ 1)\ (from\ 1)})}\ )
\end{aligned}
$$

Clearly $\mathsf{map\ (subtract\ 1)\ (from\ 1)} = \mathsf{from\ 0}$, giving

$$
\begin{aligned}
\mathsf{xs} &= \mathsf{x_0}\ :\ \underbrace{(((\mathsf{map\ Transition}) \circ (\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{xs})))\ (\mathsf{from\ 0})\ )} \\
&= \mathsf{x_0}\ :\ (\mathsf{map\ Transition}\ \underbrace{(\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{xs})\ (\mathsf{from\ 0}))})
\end{aligned}
$$

By the definition of $\underline{\mathbf{sub}}$, $\mathsf{map}\ ((\underline{\mathbf{sub}})\ \mathsf{xs})\ (\mathsf{from\ 0}) = \mathsf{xs}$, so this is

$$
\mathsf{xs} = \mathsf{x_0}\ :\ (\mathsf{map\ Transition\ xs})
$$

This is a definition of a process network to generate a stream of successive estimates. To

complete the task we must convert the until converges part into a process network too. We employ a similar approach; first decompose converges:

$$
\begin{aligned}
\mathsf{converges\ i} &= \mathsf{abs(\ (thisx - prevx)/thisx\ )} \leq \epsilon \\
&\quad \textbf{where} \\
&\quad \mathsf{thisx = xs\ \underline{\mathbf{sub}}\ i} \\
&\quad \mathsf{prevx = xs\ \underline{\mathbf{sub}}\ (i{-}1)} \\
&= \mathsf{Test\ (xs\ \underline{\mathbf{sub}}\ i)\ (xs\ \underline{\mathbf{sub}}\ (i{-}1))} \\
&\quad \textbf{where} \\
&\quad \mathsf{Test\ thisx\ nextx = abs(\ (thisx - prevx)/thisx\ )} \leq \epsilon \\
&= \mathsf{(Test \circ\circ ((\underline{\mathbf{sub}})\ xs)\ (((\underline{\mathbf{sub}})\ xs) \circ (subtract\ 1)))\ i} \\
&\quad \textbf{where} \\
&\quad \mathsf{Test\ thisx\ nextx = abs(\ (thisx - prevx)/thisx\ )} \leq \epsilon
\end{aligned}
$$

Recall from Chapter 2 that (f ∘∘ g h) x = f (g x)(h x). Now take the application of until and unfold it:

$$
\begin{aligned}
\mathsf{until\ converges\ xs} &= \mathsf{select}\ \underbrace{\mathsf{(map\ converges\ (from\ 0))}}\ \mathsf{xs} \\
&= \underbrace{\mathsf{select\ (FALSE : (map\ converges\ (from\ 1)))\ xs}} \\
&= \mathsf{select\ (map\ converges\ (from\ 1))\ (tl\ xs)} \\
&= \mathsf{select}\ \underbrace{\mathsf{(map\ (Test \circ\circ ((\underline{\mathbf{sub}})\ xs)\ (((\underline{\mathbf{sub}})\ xs) \circ (subtract\ 1)))\ (from\ 1))}} \\
&\qquad \mathsf{(tl\ xs)}
\end{aligned}
$$

Using the property that map (f ∘∘ g h) = (map2 f) ∘∘ (map g)(map h), we have

$$
\begin{aligned}
\mathsf{until\ converges\ xs} &= \mathsf{select\ (\ (map2\ Test) \circ\circ (map\ ((\underline{\mathbf{sub}})\ xs))} \\
&\qquad\qquad\qquad\qquad \underbrace{\mathsf{(map\ (((\underline{\mathbf{sub}})\ xs) \circ (subtract\ 1)))}} \\
&\qquad\qquad\qquad\qquad \mathsf{(from\ 1)\ )} \\
&\qquad \mathsf{(tl\ xs)} \\
&= \mathsf{select\ (\ (map2\ Test}\ \underbrace{\mathsf{(map\ ((\underline{\mathbf{sub}})\ xs)\ (from\ 1))}} \\
&\qquad\qquad \mathsf{(map\ ((\underline{\mathbf{sub}})\ xs)\ (map\ (subtract\ 1)\ (from\ 1))))\ )} \\
&\qquad\qquad \mathsf{(tl\ xs)}
\end{aligned}
$$

Now we again use the property map ((sub) xs) (from 0) = xs to get

$$
\begin{aligned}
\mathsf{until\ converges\ xs} &= \mathsf{select\ (map2\ Test\ (tl\ xs)} \\
&\qquad\qquad\qquad \mathsf{(map\ ((\underline{\mathbf{sub}})\ xs)}\ \underbrace{\mathsf{(map\ (subtract\ 1)\ (from\ 1))}}\ \mathsf{))} \\
&\qquad \mathsf{(tl\ xs)} \\
&= \mathsf{select\ (map2\ Test\ (tl\ xs)}\ \underbrace{\mathsf{(map\ ((\underline{\mathbf{sub}})\ xs)\ (from\ 0))}}\ \mathsf{)} \\
&\qquad \mathsf{(tl\ xs)} \\
&= \mathsf{select\ (map2\ Test\ (tl\ xs)\ xs)\ (tl\ xs)}
\end{aligned}
$$

This completes the transformation to process network form. Putting it all together we have

```
solve f f' x0
 = select (map2 Test (tl xs) xs) (tl xs)
    where
    xs = x0 : (map Transition xs)
    Test thisx nextx = abs( (thisx − prevx)/thisx ) ≤ ϵ
    Transition prevx = prevx − ((f prevx)/(f' prevx))
```

We can introduce parallelism into this definition by separating the arithmetic operations into processes:

```
solve f f' x0
 = select (Map2Test (tl xs) xs) (tl xs)
    where
    xs = x0 : (MapTransition xs)
    Map2Test thisxs nextxs = map abs ( (map2 (/) (map2 (−) thisxs prevxs) thisxs) )
    MapTransition prevxs = map2 (−) prevx (map2 (/) (map f prevx)(map f' prevx))
```

The graphical representation of this network is given in Figure 4.3, back in Chapter 4, section 4.3.1, where the transformation is employed to express a parallel implementation of the recurrence.

The transformation can be applied automatically, by a compiler, provided that at each step the references backwards to previous iterations are at a fixed offset. The technique can be summarised as follows:

1. Find the state transition function in terms of indexing into the list of iterates. In our examples these were NextEstimate and NextFib.

2. Take the definition of the list of iterates (written in terms of generate), and unfold generate. Apply reduction to generate the initial state or states for which values are given directly by the transition function.

3. Decompose the remaining, recursive, case of the state transition function into the body itself, and the functions used to collect the values from previous iterations. In our examples these were

   ```
   NextEstimate = Transition ∘ ((sub) xs) ∘ (subtract 1)
   ```

   and

   ```
   NextFib = ((+) ∘∘ (((sub) fibs) ∘ (subtract 1))
                     (((sub) fibs) ∘ (subtract 2)) ) n
   ```

4. Distribute the map introduced by generate into this composition, to produce, for example,

   xs = map NextEstimate (from 1)
      = ((map Transition) ∘ (map ((**sub**) xs)) ∘ (map (subtract 1))) (from 1)

5. Unfold the composition, giving for example,

   xs = map Transition (map ((**sub**) xs) (map (subtract 1) (from 1)))

6. Apply the equation map (subtract n) (from m) = from (m−n), producing, for example,

   xs = map Transition (map ((**sub**) xs) (from 0))

7. Apply the equations map ((**sub**) xs) (from 0) = xs, map ((**sub**) xs) (from 1) = tl xs, etc. to get, for example,

   xs = map Transition xs

The transformation-based programming environment implemented by John Darlington and his colleagues at Imperial College [De88] is designed specifically to allow transformations like this to be developed, encoded and reused.

A.4 The ray-tracer pipeline

In this example, a sequential search process is distributed over a pipeline. We have an unspecified function

```
TestForImpact :: Ray → Object → Impact
```

where Impact is a data type which describes the interaction between a ray and an object. We must find the earliest impact made by a ray, so we are also given a selection function earlier:

```
earlier :: Impact → Impact → Impact
```

The original formulation was

```
FindImpacts rays objects
 = map (FirstImpact objects) rays
   where
   FirstImpact objects ray = earliest (map (TestForImpact ray) objects)
                             where
                             earliest impacts = insert earlier NOIMPACT impacts
```

The claim is that this is equivalent to a pipelined formulation:

```
FindImpacts2 rays objects = ( (map TakeImpact) ∘
                              (insert (∘) ident
                                      (map map (map PipelineStage objects)))
                              ∘ (map MakePipeItem) )
                              rays
```

where the pipeline stage is defined by

```
PipelineStage object (PIPEITEM ray impact)
                     = PIPEITEM ray impact'
                       where
                       impact' = earlier impact NewImpact
                       NewImpact = TestForImpact ray object
```

and the stages are linked by lists of PipeItem's:

PipeItem α β ::= PIPEITEM α β

with construction and projection functions:

```
MakePipeItem ray = PIPEITEM ray NOIMPACT
TakeImpact (PIPEITEM ray impact) = impact
```

Theorem 3 *We claim that for all finite and total lists* rays *and* objects*,*

```
FindImpacts rays objects = FindImpacts2 rays objects
```

Before giving the proof we give some identities we will use. Proofs are left as exercises for the reader:

Fact 1 *Combining* insertright *and* map*:*

```
insertright op x (map g xs) = insertright h x xs
                              where
                              h a b = op (g a) b
```

Fact 2 *Abstracting a free variable from the operator parameter of* insertright*:*

```
insertright (f a) x bs = fst (insertright f' (x, a) bs)
                         where
                         f' b (x, a) = (f a b x, a)
```

For our purposes this fact is better expressed in terms of our data types:

```
insertright (f ray) NOIMPACT objects
 = TakeImpact (insertright f' (MakePipeItem ray) objects)
   where
   f' object (PIPEITEM ray impact) = PIPEITEM (f ray object impact) ray
```

(since PIPEITEM ray impact *is essentially equivalent to* (ray, impact) *but with a mnemonic tag to aid readability*[1]*).*

Fact 3 *Expressing* insertright *using a chain of compositions:*

```
insertright op x xs = (insert (∘) ident (map op xs)) x
```

Fact 4 *Propagating* map *into a chain of compositions:*

```
map (insert (∘) ident fs) = insert (∘) ident (map map fs)
```

Proof:

By reduction and use of the above facts.

Take the LHS:

```
FindImpacts rays objects = map (FirstImpact objects) rays
```

Let us consider to FirstImpact alone:

```
FirstImpact objects ray
 = insert earlier NOIMPACT (map (TestForImpact ray) objects)
```

Using Fact 1 gives

[1] There is a subtle difference; see section C.2.

```
FirstImpact objects ray
 = insertright TestAndCompare NOIMPACT objects
    where
    TestAndCompare object impact = earlier (TestForImpact ray object) impact
```

Abstract ray from TestAndCompare as a parameter:

```
FirstImpact objects ray
 = insertright (TestAndCompare' ray) NOIMPACT objects
    where
    TestAndCompare' ray object impact = earlier (TestForImpact ray object) impact
```

This is where Fact 2 comes into play, introducing the PipeItem data type: giving

```
FirstImpact objects ray
 = TakeImpact (insertright TestAndCompare" (MakePipeItem ray) objects)
    where
    TestAndCompare" object (PIPEITEM ray impact)
     = PIPEITEM (TestAndCompare' ray object impact) ray
     = PIPEITEM (earlier (TestForImpact ray object) impact) ray
```

Fact 3 introduces the chain of compositions:

```
FirstImpact objects ray
 = TakeImpact ((insert (∘) ident (map TestAndCompare" objects))
                (MakePipeItem ray))
    where
    TestAndCompare" object (PIPEITEM ray impact)
     = PIPEITEM (earlier (TestForImpact ray object) impact) ray


   = (TakeImpact ∘
      (insert (∘) ident
              (map TestAndCompare" objects))
      ∘ MakePipeItem)
     ray
    where
    TestAndCompare" object (PIPEITEM ray impact)
     = PIPEITEM (earlier (TestForImpact ray object) impact) ray
```

Putting this back into its context in the LHS, we can propagate the map into the composition (using Fact 4):

```
FindImpacts rays objects
 = map (FirstImpact objects) rays
 = map (TakeImpact ∘
        (insert (∘) ident
                (map TestAndCompare" objects))
        ∘ MakePipeItem)
       rays

   = ((map TakeImpact) ∘
      (insert (∘) ident
              (map map (map TestAndCompare" objects)))
      ∘ (map MakeItem))
     rays
   where
   TestAndCompare" object (PIPEITEM ray impact)
    = PIPEITEM (earlier (TestForImpact ray object) impact) ray
```

This is trivially equal to the RHS.

A.5 The sieve of Eratosthenes

This derivation is particularly fascinating. We have a filtering function, which takes a number p (which will be prime), and a list of numbers as, and produces the list of elements of as which are not divisible by p:

```
FilterMultiples p (a:as) = a : (FilterMultiples p as),   if not(divides p a)
                         =     FilterMultiples p as,    if divides p a
```

This has the effect of "crossing out" every multiple of p from the list of numbers as. Eratosthenes' approach was to repeat this for every prime, in increasing order. Clearly the resulting list would consist only of prime numbers—but how do we find the primes in the first place? Happily, after doing all the crossings out up to a prime p, the next uncrossed-out number must also be prime: no factor of p is greater than p, and all smaller factors have already been eliminated.

This leads us to an iterative formulation. We start with the list of natural numbers (excluding 1 for convenience). At each iteration, we filter the remaining numbers with the latest prime:

```
sieves = generate NextSieve
         where
         NextSieve 0 = from 2
         NextSieve (n+1) = FilterMultiples newprime (sieves sub n)
                           where
                           newprime = hd (sieves sub n)
```

At each step, the first element in the list is guaranteed prime, and is used in the next step for crossing out. From this iteration, the list of primes itself is easily found:

```
primes = generate FindPrime
         where
         FindPrime n = hd (sieves sub n)
```

We just collect the first element of the list at each iteration. It is not hard, using the techniques developed for removing **sub** from recurrences, to simplify this definition to just

```
primes = map hd (iterate g (from 2))
         where
         g (a:as) = FilterMultiples a as
```

Recall one of the alternative definitions of iterate:

```
iterate f x = x : (iterate f (f x))
```

(So that iterate f x = [x, f x, f(f x)...]). Now define a function sieve so that

```
primes = sieve (from 2)
         where
         sieve as = map hd (iterate g as)
```

Instantiate sieve for non-empty as, and then unfold the definition of iterate

```
sieve (a:as) = map hd (iterate g (a:as))
             = map hd ((a:as):(iterate g (g (a:as))))
             = a : (map hd (iterate g (g (a:as))))
             = a : (sieve (g (a:as)) )
             = a : (sieve (FilterMultiples a as))
```

This gives the definition as required:

```
primes = sieve (from 2)
         where
         sieve (a:as) = a : (sieve (FilterMultiples a as))
```

A.6 Transforming divide-and-conquer into a cycle

The next clutch of proofs support the derivation of a cyclic formulation of the divide-and-conquer algorithm form. The derivation itself appears in Chapter 4 section 4.8. The starting point is the higher-order function to capture the divide-and-conquer form:

```
DivideAndConquer :: (α → [β] → β) → (α → [α]) → α → β

DivideAndConquer CombineSolutions Decompose problem
 = Solve problem
   where
   Solve problem = CombineSolutions problem (map Solve SubProblems)
                   where
                   SubProblems = Decompose problem
```

A.6.1 Introducing an intermediate tree

We introduce an intermediate data structure to represent how the problem is broken down into subproblems:

```
MultiTree α β ::= MNODE α (α → [α] → β) Num [MultiTree α β]
```

We define

```
DivideAndConquer' CombineSolutions Decompose problem
 = EvaluateTree (BuildTree problem)
   where
   BuildTree problem = MNODE problem
                               CombineSolutions
                               NoOfSubproblems
                               (map BuildTree Subproblems)
                       where
                       Subproblems = Decompose problem
                       NoOfSubproblems = length SubProblems
```

where

```
EvaluateTree (MNODE problem CombineSolutions n subtrees)
             = CombineSolutions problem (map EvaluateTree subtrees)
```

Theorem 4 DivideAndConquer' = DivideAndConquer

Proof

By equational reasoning:

First, introduce an auxiliary function Solve':

```
DivideAndConquer' CombineSolutions Decompose problem
 = Solve' problem
   where
   Solve' problem = (EvaluateTree ∘ BuildTree) problem

   BuildTree problem = MNODE problem
                             CombineSolutions
                             NoOfSubproblems
                             (map BuildTree Subproblems)
                       where
                       Subproblems = Decompose problem
                       NoOfSubproblems = length SubProblems
```

Then apply reduction:

```
Solve' problem
 = EvaluateTree (MNODE problem
                       CombineSolutions
                       NoOfSubproblems
                       (map BuildTree Subproblems))
   where
   Subproblems = Decompose problem
   NoOfSubproblems = length SubProblems


 = CombineSolutions problem (map EvaluateTree (map BuildTree Subproblems))
   where
   Subproblems = Decompose problem
   NoOfSubproblems = length SubProblems


 = CombineSolutions problem (map (EvaluateTree ∘ BuildTree) Subproblems)
   where
   Subproblems = Decompose problem
   NoOfSubproblems = length SubProblems
```

At this point a fold step applies, giving

```
Solve' problem
 = CombineSolutions problem (map Solve' Subproblems)
   where
   Subproblems = Decompose problem
   NoOfSubproblems = length SubProblems
```

This has precisely the form of the original definition of DivideAndConquer.

A.6.2 The breadth-first tree–stream interconversion

In this section we derive a pair of breadth-first tree–stream interconversion functions. The tree is transformed into a list of tokens, each carrying details of a node:

```
MultiTreeToken α β ::= MTREETOKEN α (α → [α] → β) Num
```

The functions we must derive have the type specifications

```
MTreeToStream :: MultiTree α β → [MultiTreeToken α β]
```

and

```
StreamToMTree :: [MultiTreeToken α β] → Multitree α β
```

It turns out to be easier to derive a slightly more general pair of functions,

```
ListOfMTreesToStream :: [MultiTree α β] → [MultiTreeToken α β]
StreamToListOfMTrees :: Num → [MultiTreeToken α β] → [Multitree α β]
```

so that

```
MTreeToStream tree = ListOfMTreesToStream [tree]
```

and

```
StreamToMTree stream = StreamToListOfMTrees 1 stream
```

The first parameter to StreamToListOfMTrees must be the number of trees we must extract from the incoming stream—in this case just one.

The specification

The specification comes in two parts. Firstly, we obviously require that the type specifications be satisfied, and that we can get the trees back again:

```
StreamToListOfMTrees (length trees) (ListOfMTreesToStream trees) = trees
```

However, we also demand that the list representation be generated in "breadth-first" order, and this needs specifying. The idea is that the tree is made up of successive generations,

so that each node of each generation is the same distance from the root. We can formalise this by writing down some functions for separating off the first generation of a list of trees from the subsequent ones:

```
RootsOf :: [MultiTree α β] → [MultiTreeToken α β]

RootsOf [ ] = [ ]
RootsOf ((MNODE p op n subtrees) : trees) = (MTREETOKEN p op n)
                                              : (RootsOf trees)
```

RootsOf trees produces a list of tokens, each representing the root node of the corresponding tree in trees. Notice that we expect n to be the number of children of this node—i.e. length subtrees.

RootsOf's counterpart is SubtreesOf, which picks out each node's children:

```
SubtreesOf :: [MultiTree α β] → [MultiTree α β]

SubtreesOf [ ] = [ ]
SubtreesOf ((MNODE p op n subtrees) : trees) = subtrees ++ (SubtreesOf trees)
```

These two functions allow us to decompose trees into generations. All that remains is to find a way to put them back together again:

```
JoinLayers :: [MultiTreeToken α β] → [Multitree α β] → [Multitree α β]

JoinLayers [ ] [ ] = [ ]
JoinLayers ((MTREETOKEN p op n) : l1) l2 = (MNODE p op n (take n l2))
                                             : (JoinLayers l1 (drop n l2))
```

It is not hard to verify (using partial structural induction) that these functions operate as intended: for all lists of trees, trees,

```
trees = JoinLayers (RootsOf trees) (SubtreesOf trees)
```

These functions establish a well-founded ordering based on generations. Now to the specification that the list be generated in breadth-first order. What we mean is that the output list should consist of each complete generation, one-at-a-time, from the roots:

```
ListOfMTreesToStream trees = (RootsOf trees) ++
                             (RootsOf (SubtreesOf trees)) ++
                             (RootsOf (SubtreesOf (SubtreesOf trees))) ++ ···
```

This is simply captured recursively:

```
ListOfMTreesToStream [ ] = [ ]

ListOfMTreesToStream trees = (RootsOf trees) ++
                             (ListOfMTreesToStream (SubtreesOf trees))
```

This is an adequate implementation for ListOfMTreesToStream.

It will also be fruitful to note how to separate the generations when they are represented in the stream form. For this, we must know the number n of nodes in the first generation. Then we have simply that

```
FirstGeneration n tokens = take n tokens
SubsequentGenerations n tokens = drop n tokens
```

To find n we specify that

```
SizeOfNextGeneration (RootsOf trees) = length (SubtreesOf trees)
```

We have to use an implementation which doesn't need the tree form. If tokens = RootsOf trees then

```
SizeOfNextGeneration tokens = sum (map NumberOfChildren tokens)
                              where
                              NumberOfChildren (MTREETOKEN p op n) = n
```

Before proceeding, take note of two equalities which are easily verified by reduction (again, n = length (RootsOf trees)):

```
FirstGeneration n (ListOfMTreesToStream trees) = RootsOf trees
```

and

```
SubsequentGenerations n (ListOfMTreesToStream trees)
=
ListOfMTreesToStream (SubtreesOf trees)
```

Deriving StreamToListOfMTrees

While the translation from trees to lists was easily derived from its specification, synthesising an executable definition for StreamToListOfMTrees is rather more difficult. Its specification is just

```
StreamToListOfMTrees n stream = trees
where
stream = ListOfMTreesToStream trees
n = length trees
```

Let us instantiate StreamToListOfMTrees for two cases: when the list of trees (and therefore stream) is empty, and when it consists of one or more generations. For streams = [ ], it is clear that n must also be zero, and we construct the empty list of trees:

```
StreamToListOfMTrees 0 [ ] = [ ]
```

For the non-empty case we know that we can decompose stream so that

```
stream = (FirstGeneration n stream) ++ (SubsequentGenerations n stream)
```

We observed earlier that

```
FirstGeneration n (ListOfMTreesToStream trees) = RootsOf trees
```

and

```
SubsequentGenerations n (ListOfMTreesToStream trees)
=
ListOfMTreesToStream (SubtreesOf trees)
```

We have

```
StreamToListOfMTrees n stream
 = JoinLayers (RootsOf trees) (SubtreesOf trees)
 = JoinLayers gen1 (SubtreesOf trees)
    where
    gen1 = FirstGeneration n stream

 = JoinLayers gen1 subtrees
    where
    gen1 = FirstGeneration n stream
    subtrees = StreamToListOfMTrees m (ListOfMTreesToStream (SubtreesOf trees))
    m = length (SubtreesOf trees)
```

(by hypothesis)

```
= JoinLayers gen1 subtrees
  where
  gen1 = FirstGeneration n stream
  subtrees = StreamToListOfMTrees m (SubsequentGenerations n
                                      (ListOfMTreesToStream trees))
  m = length (SubtreesOf trees)

= JoinLayers gen1 subtrees
  where
  gen1 = FirstGeneration n stream
  subtrees = StreamToListOfMTrees m (SubsequentGenerations n stream)
  m = length (SubtreesOf trees)

= JoinLayers gen1 subtrees
  where
  gen1 = FirstGeneration n stream
  subtrees = StreamToListOfMTrees m (SubsequentGenerations n stream)
  m = SizeOfNextGeneration (RootsOf trees)

= JoinLayers gen1 subtrees
  where
  gen1 = FirstGeneration n stream
  subtrees = StreamToListOfMTrees m (SubsequentGenerations n stream)
  m = SizeOfNextGeneration gen1
```

This completes the derivation, since StreamToListOfMTrees n stream no longer refers to trees. The definitions are collected below for clarity:

```
ListOfMTreesToStream :: [MultiTree α β] → [MultiTreeToken α β]

ListOfMTreesToStream trees = (RootsOf trees) ++
                             (ListOfMTreesToStream (SubtreesOf trees))
```

and

```
StreamToListOfMTrees :: [MultiTreeToken α β] → [Multitree α β]

StreamToListOfMTrees 0 [ ] = [ ]

StreamToListOfMTrees n stream
 = JoinLayers gen1 subtrees,        n ≠ 0
   where
   gen1 = FirstGeneration n stream
   subtrees = StreamToListOfMTrees m (SubsequentGenerations n stream)
   m = SizeOfNextGeneration gen1
```

The definition of StreamToListOfMTrees can be made more efficient using the optimisations of section A.1.1. In particular, the first generation, its length, and the subsequent generations can all be computed in a single pass.

A.6.3 Verifying the cyclic definition

By applying reduction (see section 4.8.2) to combine BuildTree with ListOfMTreesToStream, we reached the following recursive definition for BuildStreamsOfTrees, a function which decomposes a list of rays directly into the stream representation of their subray trees:

```
BuildStreamsOfTrees [ ] [ ] = [ ]

BuildStreamsOfTrees [ ] subproblems = BuildStreamsOfTrees subproblems [ ]

BuildStreamsOfTrees (problem:siblingproblems) oldsubproblems
                    = (MTREETOKEN problem CombineSolutions NoOfSubproblems)
                       : (BuildStreamsOfTrees siblingproblems
                                              (oldsubproblems++Subproblems))
                      where
                      Subproblems = Decompose problem
                      NoOfSubproblems = length Subproblems
```

In section 4.8.2 we claim that this is equivalent to a definition which is not recursive as such, but uses a cyclic stream definition:

```
BuildStreamsOfTrees' [ ] [ ] = [ ]
```

```
BuildStreamsOfTrees' problems subproblems
 = output
    where
    (output, feedback)
     = SplitStream
        ((map FEEDBACKTAG subproblems) ++
         (join (map LayerOf (problems++feedback))))


    LayerOf problem
     = (OUTPUTTAG (MTREETOKEN problem CombineSolutions NoOfSubproblems))
        : (map FEEDBACKTAG Subproblems)
       where
       Subproblems = Decompose problem
       NoOfSubproblems = length Subproblems
```

where SplitStream separates a stream of tagged objects into two streams of untagged ones:

```
SplitStream :: [TaggedStreamItem α β] → ([MultiTreeToken α β], α)

SplitStream [ ] = ([ ], [ ])

SplitStream ((OUTPUTTAG token) : rest)
            = (token : rest1, rest2)
              where
              (rest1, rest2) = SplitStream rest


SplitStream ((FEEDBACKTAG subproblem) : rest)
            = (rest1, subproblem : rest2)
              where
              (rest1, rest2) = SplitStream rest
```

and join flattens a list of lists into a list:

```
join :: [[α]] → [α]

join xss = insert (++) [ ] xss
```

Theorem 5 BuildStreamsOfTrees = BuildStreamsOfTrees'

Proof

By recursion induction. We must show that BuildStreamsOfTrees' satisfies each of the three equations defining BuildStreamsOfTrees. This verifies that BuildStreamsOfTrees $\sqsubseteq$ BuildStreamsOfTrees' (see section 2.5.5). We omit a proof of the equality itself, because we know that BuildStreamsOfTrees is defined for all parameter values of interest.

First Equation: We must show that

```
BuildStreamsOfTrees' [ ] [ ] = [ ]
```

This follows trivially from the first equation defining BuildStreamsOfTrees'.

Second Equation: We must show that

```
BuildStreamsOfTrees' [ ] subproblems = BuildStreamsOfTrees' subproblems [ ]
```

If we unfold the RHS we get

```
RHS = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG [ ]) ++
           (join (map LayerOf (subproblems++feedback))))
```

The LHS unfolds to

```
LHS = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG subproblems) ++
           (join (map LayerOf (feedback))))
```

The elements of the list subproblems are tagged so they are emitted in the right-hand feedback stream:

```
LHS = output
      where
      (output, feedback) = (output, subproblems++feedback')
      (output, feedback')
       = SplitStream
          ((map FEEDBACKTAG [ ]) ++
           (join (map LayerOf (feedback))))
```

Simplifying to get rid of feedback' we get

```
LHS = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG [ ]) ++
           (join (map LayerOf (subproblems++feedback))))
```

But this is precisely the same as the RHS.

Third Equation: We must show that

```
BuildStreamsOfTrees' (problem:siblingproblems) oldsubproblems
 = (MTREETOKEN problem CombineSolutions NoOfSubproblems)
    : (BuildStreamsOfTrees' siblingproblems
                            (oldsubproblems++Subproblems))
   where
   Subproblems = Decompose problem
   NoOfSubproblems = length Subproblems
```

We apply reduction to the LHS:

```
LHS = BuildStreamsOfTrees' (problem:siblingproblems) oldsubproblems
    = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG oldsubproblems) ++
           (join (map LayerOf ((problem:siblingproblems)++feedback)) ))
    = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG oldsubproblems) ++
           (join ((LayerOf problem)
                  : (map LayerOf (siblingproblems++feedback))) ))
```

Unfolding LayerOf gives

```
LHS = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG oldsubproblems) ++
           (join ((OUTPUTTAG (MTREETOKEN problem
                                         CombineSolutions
                                         NoOfSubproblems))
                  : (map FEEDBACKTAG Subproblems))
                 : (map LayerOf (siblingproblems++feedback)) ))
          where
          Subproblems = Decompose problem
          NoOfSubproblems = length Subproblems
```

Reducing the application of join gives

```
LHS = output
      where
      (output, feedback)
       = SplitStream
          ((map FEEDBACKTAG oldsubproblems) ++
           [OUTPUTTAG (MTREETOKEN problem
                               CombineSolutions
                               NoOfSubproblems)] ++
           (map FEEDBACKTAG Subproblems) ++
           (map LayerOf (siblingproblems++feedback)))
          where
          Subproblems = Decompose problem
          NoOfSubproblems = length Subproblems
```

We can now float the MTREETOKEN structure out to the output:

```
LHS = (MTREETOKEN problem CombineSolutions NoOfSubproblems)
       : output'
      where
      (output', feedback)
       = SplitStream
          ((map FEEDBACKTAG oldsubproblems) ++
           (map FEEDBACKTAG Subproblems) ++
           (map LayerOf (siblingproblems++feedback)))
      where
      Subproblems = Decompose problem
      NoOfSubproblems = length Subproblems
```

We know that map FEEDBACKTAG is distributive over "++", giving

```
LHS = (MTREETOKEN problem CombineSolutions NoOfSubproblems)
        : output'
        where
        (output', feedback)
         = SplitStream
            ((map FEEDBACKTAG oldsubproblems++Subproblems) ++
             (map LayerOf (siblingproblems++feedback)))
        where
        Subproblems = Decompose problem
        NoOfSubproblems = length Subproblems
```

Now notice that

```
output' = BuildStreamsOfTrees' siblingproblems
                               (oldsubproblems++Subproblems)
```

so we have

```
LHS = (MTREETOKEN problem CombineSolutions NoOfSubproblems)
        : BuildStreamsOfTrees' siblingproblems
                               (oldsubproblems++Subproblems)
        where
        Subproblems = Decompose problem
        NoOfSubproblems = length Subproblems
```

This is identical to the RHS.

Appendix B

Common Definitions

This appendix collects definitions of commonly used symbols and functions. Intermediary definitions in program derivations are not generally included.

B.1 Symbols

$==$

Equality of types, used for defining synonyms for types. For example $\mathsf{name} == [\mathsf{Char}]$.

$::=$

Algebraic data type declaration. Used to construct a new data type from tagged alternatives (separated by — and recursion. The tags are known as *constructors*.

$::$

Type specification/assertion. For example, $\mathsf{f} :: \tau$ assert the f is a member of the type τ. It is generally used to give a partial specification of the object to aid the reader's understanding and to aid compiler checking.

$\rightarrow$

$\alpha \rightarrow \beta$ is the type of a function which takes one parameter of type α, and returns a result of type β. $\alpha \rightarrow (\beta \rightarrow \gamma)$ is the type of a two-parameter function. The brackets here can be omitted.

$\underbrace{\quad}$

An underbrace is used in this book to mark an expression which is shortly to be rewritten.

$\vee$

Logical or.

$\wedge$

Logical and.

++
: The infix form of the append function, defined below.

:, [ ]
: Shorthand forms of the CONS and NIL constructors of the List data type.

"..."
: The expression "abc" is shorthand for the list of characters 'a' : 'b' : 'c' : [ ].

∘
: The infix form of the function compose. f ∘ g denotes a function which applies g to its parameter, and then applies f to the result.

∘∘
: The infix form of the function compose2. f ∘∘ g h is a two parameter function which applies g to its first parameter, h to its second parameter, and applies f to the two results.

⊥
: Read "bottom", ⊥ denotes a computation which does not terminate. When looking at snapshots of a computation, ⊥ can be thought of as standing for a value which has not yet been computed.

⊑
: Read "approximates". Informally x ⊑ y if further computation might refine x until it is equal to y.

≺, ≻
: These symbols are used in this book for an ordering relation (analogous to ≤ and >) which is "well founded"—that is there exists no infinite chain of decreasing values.

#
: An application a b can be marked a # b is its result must be undefined if b is undefined—in which case it is called a "strict" application. A function definition f a b c = e can be marked f a $b_{\#}$ c = e if all applications of f to its second parameter are strict.

□
: The expression □f denotes the expression in which f is applied. This must be uniquely determined. When written (□), reduction must be applied until □ is applied to a value.

B.2 Types

BinaryTree:

BinaryTree α ::= EMPTY |
LEAF α |
NODE (BinaryTree α) (BinaryTree α)

Bool:

Bool ::= TRUE | FALSE

Bundle:

Bundle α ::= BUNDLE α

BUNDLE is used only as a visible signal to the reader that the parameter data structure is being used for bundling.

Char:

This type contains all the characters, and might be defined by the equation

Char ::= 'a' | 'b' | 'c' ... 'z' | 'A' | 'B' ... 'Z' | '0' | '1' ... '9' ...

It would normally include the characters of the ASCII code, and be ordered in the same way.

Impact:

Impact ::= NOIMPACT |
IMPACT Num ImpactInformation

List:

List α ::= NIL | CONS α (List α)

MultiTree:

MultiTree α β ::= MNODE α ($\alpha \rightarrow [\alpha] \rightarrow \beta$) Num [MultiTree α β]

In this kind of tree, each node carries a function as well as a list of subtrees. The number should be equal to the number of subtrees.

MultiTreeToken:

MultiTreeToken α β ::= MTREETOKEN α ($\alpha \rightarrow [\alpha] \rightarrow \beta$) Num

This type is used in the breadth-first stream representation of the MultiTree type.

PipeItem:

PipeItem α β ::= PIPEITEM α β

This is simply a tagged pair type, used instead of just (α, β) for ease of readability. It is need when pipelining and insert operation.

Sample:

Sample ::= HI | LO | XX

This is an approximation to the signal level on a wire, used in specifying digital circuits.

Signal:

Signal == [Sample]

TaggedStreamItem:

TaggedStreamItem α β ::= OUTPUTTAG (MultiTreeToken α β)
| FEEDBACKTAG α

This type is like a "union" type: it includes two different typed objects, requiring that they be tagged to indicate which. It is used in the cyclic formulation of the divide phase of DivideAndConquer.

B.3 Functions

append (++):

append :: $[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$

append (a : as) bs = a : (append as bs)
append [ ] bs = bs

The application append as bs is normally written as++bs.

ApplyLNO:

$$\text{ApplyLNO} :: ([\alpha] \to \alpha) \to \ll\alpha\gg \to \ll\alpha\gg$$

```
ApplyLNO op matrix
 = MakeMatrix LocalOperation
   where
   LocalOperation (i,j)
    = matrix sub (i,j), if OnBoundary matrix (i,j)
   LocalOperation (i,j)
    = op [matrix sub (i−1,j),
          matrix sub (i,j−1),
          matrix sub (i+1,j),
          matrix sub (i,j+1),
          matrix sub (i,j)], otherwise
```

arc:

$$\text{arc} :: \alpha \to \alpha \to \text{Bool}$$

This relation is used to build assertions about process distribution. The assertion arc a b requires the compiler to place the processes which compute expressions a and b on separate processors, but to arrange for them to be able to communicate with one another efficiently. Note that arc a b is equivalent to the assertion arc b a.

DivideAndConquer:

$$\begin{aligned}\text{DivideAndConquer} :: {} & (\alpha \to \beta) \\ & \to (\alpha \to [\beta] \to \beta) \\ & \to (\alpha \to [\alpha]) \\ & \to (\alpha \to \text{Bool}) \\ & \to \alpha \\ & \to \beta\end{aligned}$$

```
DivideAndConquer SimplySolve CombineSolutions Decompose Trivial problem
= Solve problem
  where
  Solve problem = SimplySolve problem,                          if Trivial problem
  Solve problem = CombineSolutions problem
                                    (map Solve SubProblems)     otherwise
                  where
                  SubProblems = Decompose problem
```

abs:

```
abs :: Num → Num
abs x = x,      if x ≥ 0
abs x = −x,     otherwise
```

all:

```
all :: [Bool] → Bool
all = insert (∧) TRUE
```

chain:

```
chain :: (Bool → Bool → Bool) → [(α → β)] → Bool

chain relation [f] = TRUE
chain relation (f1 : f2 : fs) = (relation f1 f2) ∧ (chain relation f2 fs)
```

compose (∘):

```
compose :: (β → γ) → (α → β) → α → γ

compose f g = f ∘ g = h
                     where
                     h x = f (g x)
```

compose2 (∘∘):

compose2 :: $(\beta_1 \to \beta_2 \to \gamma) \to (\alpha \to \beta_1) \to (\alpha \to \beta_2) \to \alpha \to \gamma$

compose2 f g1 g2 = f ∘∘ g1 g2 = h
where
h x = f (g1 x) (g2 x)

cond:

cond :: Bool $\to \alpha \to \alpha \to \alpha$

cond TRUE a b = a
cond FALSE a b = b

construct:

construct :: $[\alpha \to \beta] \to \alpha \to \beta$

construct [ ] x = [ ]
construct (f : fs) x = (f x) : (construct fs x)

const:

const :: $\alpha \to \alpha$
const x = x

divides:

divides :: Num → Num → Bool

divides p a is TRUE is p divides a exactly, False otherwise.

drop:

drop :: Num $\to [\alpha] \to [\alpha]$

drop n (a : as) = drop (n−1) as, **if** n ≠ 0
drop n [ ] = [ ], **if** n ≠ 0
drop 0 as = as

earlier:

```
earlier :: Impact → Impact → Impact

earlier NOIMPACT NOIMPACT = NOIMPACT
earlier (IMPACT dist1 info1) NOIMPACT = (IMPACT dist1 info1)
earlier NOIMPACT (IMPACT dist2 info2) = (IMPACT dist2 info2)
earlier (IMPACT dist1 info1)
        (IMPACT dist2 info2) = (IMPACT dist1 info1), if dist1 ≤ dist2
earlier (IMPACT dist1 info1)
        (IMPACT dist2 info2) = (IMPACT dist2 info2), if dist1 > dist2
```

EvaluateTree:

```
EvaluateTree :: (MultiTree α β) → β

EvaluateTree (MNODE problem CombineSolutions n subtrees)
             = CombineSolutions problem (map EvaluateTree subtrees)
```

EvenOnes:

```
EvenOnes :: [α] → [α]

EvenOnes [ ] = [ ]
EvenOnes [a0] = [a0]
EvenOnes (a0 : a1 : as) = a0 : (EvenOnes as)
```

fan:

```
fan :: (Bool → Bool → Bool) → α → [β → γ] → Bool

fan relation a bs = all (map (relation a) bs)
```

filter:

```
filter :: (α → Bool) → [α] → [α]

filter predicate [ ] = [ ]
filter predicate (a : as) = a : (filter predicate as),        if predicate a
filter predicate (a : as) =     (filter predicate as),        otherwise
```

FindImpacts:

```
FindImpacts :: [Ray] → [Object] → [Impacts]
FindImpacts rays objects = map (FirstImpact objects) rays
```

FirstImpact:

```
FirstImpact :: [Object] → Ray → Impact

FirstImpact objects ray = earliest (map (TestForImpact ray) objects)
                          where
                          earliest impacts = insert earlier NOIMPACT impacts
```

from:

```
from :: Num → [Num]
from n = n : ( from (n + 1) )
```

fst:

```
fst :: (α, β) → α
fst (a, b) = a
```

generate:

```
generate :: (Num → α) → [α]
generate f = map f (from 0)
```

hd:

hd :: $[\alpha] \to \alpha$
hd (x : xs) = x

ident:

ident :: $\alpha \to \alpha$
ident x = x

insert:

insert :: $(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$

insert (op) base [ ] = base
insert (op) base [a1, a2, a3, $\cdots$ aN] = a1 op a2 op a3 $\cdots$ op aN

The function parameter is written (op) here because it is convenient to use it in infix form on the RHS. This function is applicable only when (op) is associative.

insertleft:

insertleft :: $(\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$

insertleft f base [ ] = base
insertleft f base (a : as) = insertleft f (f base a) as

insertright:

insertright :: $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$

insertright f base [ ] = base
insertright f base (a : as) = f a (insertright f base as)

iterate:

iterate :: $(\alpha \to \alpha) \to \alpha \to [\alpha]$

iterate f x = x : (iterate f (f x))

A useful alternative definition of iterate is

```
iterate f x = output
              where
              output = x : (map f output)
```

join:

```
join :: [[α]] → [α]

join as = insert (++) [ ] as
```

ladder:

```
ladder :: (Bool → Bool → Bool) → [α → β] → [γ → δ] → Bool

ladder relation [ ] [ ] = TRUE
ladder relation (a : as) (b : bs) = (relation a b) ∧ (ladder relation as bs)
```

length:

```
length :: [α] → Num

length [ ] = 0
length (a : as) = 1 + (length as)
```

A more efficient definition (when strictness analysis annotations are interpreted as call-by-value parameter passing) is

```
length as = length' 0 as
            where
            length' n [ ] = n
            length' n (a:as) = length' (n+1) as
```

ListToTree1:

```
ListToTree1 :: [α] → BinaryTree α
```

```
ListToTree1 [ ] = EMPTY
ListToTree1 [a] = LEAF a
ListToTree1 (a0:a1:as) = NODE (ListToTree1 (take m (a0:a1:as)))
                              (ListToTree1 (drop m (a0:a1:as)))
                         where
                         m = (length (a0:a1:as))/2
```

ListToTree2:

```
ListToTree2 :: [α] → BinaryTree α
```

```
ListToTree2 [ ] = EMPTY
ListToTree2 [a] = LEAF a
```

```
ListToTree2 (a0:a1:as) = NODE (ListToTree2 (EvenOnes (a0:a1:as)))
                              (ListToTree2 (OddOnes (a0:a1:as)))

                         where
                         EvenOnes [ ] = [ ]
                         EvenOnes [a0] = [a0]
                         EvenOnes (a0 : a1 : as) = a0 : (EvenOnes as)

                         OddOnes [ ] = [ ]
                         OddOnes [a0] = [ ]
                         OddOnes (a0 : a1 : as) = a1 : (OddOnes as)
```

ListToVector:

```
ListToVector :: [α] → <α>
```

This is specified (but not implemented) by the requirement that for all 0 ≤ i ≤ (length as)−1,

(ListToVector as) **sub** i = as **sub** i

MakeList:

MakeList :: Num $\rightarrow$ (Num $\rightarrow \alpha$) $\rightarrow$ [α]

MakeList length f = VectorToList (MakeVector length f)

MakeMatrix:

MakeMatrix :: (Num, Num) $\rightarrow$ ((Num, Num) $\rightarrow \alpha$) $\rightarrow \ll\alpha\gg$

This is specified (but not implemented) by the requirement that for all 0 $\leq$ i $\leq$ xBound and 0 $\leq$ j $\leq$ yBound,

(MakeMatrix (xBound,yBound) f) **sub** (i,j) = f (i,j)

MakePipeItem:

MakePipeItem :: Ray $\rightarrow$ PipeItem Ray Impact
MakePipeItem ray = PIPEITEM ray NOIMPACT

MakeVector:

MakeVector :: Num $\rightarrow$ (Num $\rightarrow \alpha$) $\rightarrow <\alpha>$

This is specified (but not implemented) by the requirement that for all 0 $\leq$ i $\leq$ bound,

(MakeVector bound f) **sub** i = f i

map:

map :: ($\alpha \rightarrow \beta$) $\rightarrow$ [α] $\rightarrow \beta$

map f [ ] = [ ]
map f (x:xs) = (f x) : (map f xs)

map2:

map2 :: $(\alpha \to \beta \to \gamma) \to [\alpha] \to [\beta] \to [\gamma]$

map2 op (a : as) (b : bs) = (op a b) : (map2 op as bs)
map2 op [ ] [ ] = [ ]

MatrixAll:

MatrixAll :: $\ll$Bool$\gg \to$ Bool

This is specified by the requirement that for all 0 $\leq$ i $\leq$ xBound and 0 $\leq$ j $\leq$ yBound,

m **sub** (i,j) = TRUE

where (xBound,yBound) = MatrixBounds m.

MatrixBounds:

MatrixBounds :: $\ll\alpha\gg \to$ (Num,Num)

This is specified by the requirement that

MatrixBounds (MakeMatrix (xBound,yBound) f) = (xBound,yBound)

MatrixMap:

MatrixMap :: $(\alpha \to \beta) \to \ll\alpha\gg \to \ll\beta\gg$

We require that

MatrixMap f (MakeMatrix (xBnd,yBnd) g)
 = MakeMatrix (xBnd,yBnd) (f $\circ$ g)

MatrixMap2:

MatrixMap2 :: $(\alpha \to \beta \to \gamma) \to \ll\alpha\gg \to \ll\beta\gg \ll\gamma\gg$

We require that

MatrixMap2 f (MakeMatrix (xBnd,yBnd) g)
 (MakeMatrix (xBnd,yBnd) h) = MakeMatrix (xBnd,yBnd) (f $\circ\circ$ g h)

mesh:

```
mesh :: ≪α≫ → Bool

mesh matrix = MatrixAll (ApplyLNO LinkNeighbours matrix)
              where
              LinkNeighbours [west, south, east, north, home]
               = fan arc home [west, south, east, north]
```

MTreeToStream:

```
MTreeToStream :: MultiTree α β → [MultiTreeToken α β]

MTreeToStream tree = ListOfMTreesToStream [tree]

where

ListOfMTreesToStream :: [MultiTree α β] → [MultiTreeToken α β]

ListOfMTreesToStream trees = (RootsOf trees) ++
                             (ListOfMTreesToStream (SubtreesOf trees))
```

not:

```
not :: Bool → Bool
not TRUE = FALSE
not FALSE = TRUE
```

OddOnes:

```
OddOnes :: [α] → [α]

OddOnes [ ] = [ ]
OddOnes [a0] = [ ]
OddOnes (a0 : a1 : as) = a1 : (OddOnes as)
```

OnBoundary:

```
OnBoundary :: ≪α≫ → (Num,Num) → Bool

OnBoundary matrix (i,j) = (i=0) ∨ (j=0) ∨ (i=iBound−1) ∨ (j=jBound−1)
                          where
                          (iBound,jBound) = MatrixBound matrix
```

pair:

```
pair :: α → β → (α, β)
pair a b = (a, b)
```

pipeline:

```
pipeline :: [α → α] → [α] → [α]

pipeline fs xs = (insert (∘) ident (map map fs)) xs
```

ply:

```
ply :: [(α → β)] → [α] → [β]

ply [ ] [ ] = [ ]
ply (f : fs)(x : xs) = (f x) : (ply fs xs)
```

replicate:

```
replicate :: Num → α → [α]

replicate 0 x = [ ]
replicate (n+1) x = x : (replicate n x)
```

reverse:

```
reverse :: [α] → [α]

reverse [ ] = [ ]
reverse (x : xs) = (reverse xs) ++ [x]
```

select:

```
select :: [Bool] → [α] → α

select (FALSE : tests) (x : xs) = select tests xs
select (TRUE : tests) (x : xs) = x
```

snd:

```
snd :: (α, β) → β
snd (a, b) = b
```

split:

```
split :: Num → [α] → ([α],[α])

split 0 as = ([ ], as)
split 0 [ ] = ([ ], [ ]),                    if n ≠ 0
split n (a:as) = (a: front, back),           if n ≠ 0
                 where
                 (front, back) = split (n−1) as
```

SplitStream:

```
SplitStream :: [TaggedStreamItem α β] → ([MultiTreeToken α β], α)

SplitStream [ ] = ([ ], [ ])


SplitStream ((OUTPUTTAG token) : rest)
            = (token : rest1, rest2)
              where
              (rest1, rest2) = SplitStream rest


SplitStream ((FEEDBACKTAG subproblem) : rest)
            = (rest1, subproblem : rest2)
              where
              (rest1, rest2) = SplitStream rest
```

StreamOfMatricesToMatrixOfStreams:

```
StreamOfMatricesToMatrixOfStreams :: [≪α≫] → ≪[α]≫

StreamOfMatricesToMatrixOfStreams ms
 = MakeMatrix (MatrixBounds (hd ms)) EachStream
   where
   EachStream (i,j) = generate Elements
                      where
                      Elements k = (ms sub k) sub (i,j)
```

StreamToMTree:

```
StreamToMTree :: [MultiTreeToken α β] → Multitree α β

StreamToMTree stream = StreamToListOfMTrees 1 stream

where

StreamToListOfMTrees :: [MultiTreeToken α β] → [Multitree α β]

StreamToListOfMTrees 0 [ ] = [ ]

StreamToListOfMTrees n stream
 = JoinLayers gen1 subtrees,        n ≠ 0
   where
   gen1 = FirstGeneration n stream
   subtrees = StreamToListOfMTrees m (SubsequentGenerations n stream)
   m = SizeOfNextGeneration gen1
```

sub :

This subscripting operator is used for lists, vectors and matrices. For lists its definition is

```
(sub) :: [α] → Num → α

(a : as) sub 0 = a
(a : as) sub (n+1) = as sub n
```

sum:

sum :: [Num] → Num

sum as = insert (+) 0 as = insertleft (+) 0 as = insertright (+) 0 as

take:

take :: Num → [α] → [α]

take n (a : as) = a : (take (n−1) as), **if** n ≠ 0
take n [ ] = [ ], **if** n ≠ 0
take 0 as = [ ]

TakeImpact:

TakeImpact :: PipeItem Ray Impact → Impact
TakeImpact (PIPEITEM ray impact) = impact

TestForImpact:

TestForImpact :: Ray → Object → Impact

This function's definition is not given here to avoid unnecessary detail. It checks whether Ray intersects with Object. If not it returns NOIMPACT. If so, it returns an Impact data object containing details of how far along the ray the impact occurred, where the rays (if any) contributing to this ray's colour come from, and details of the surface characteristics in the form of a function which combines the colours of the contributory rays to yield the colour of the original ray.

tl:

tl :: [α] → [α]
tl (x : xs) = xs

transpose:

```
transpose :: [[α]] → [[α]]

transpose rows = [ ],                                          if rows = [ ]
transpose rows = (map hd rows) : (transpose (map tl rows))     otherwise
```

The important use for this function is in transforming an infinite stream of (finite length) lists into a (finite length) list of infinite streams—and back again.

TreeToList1:

```
TreeToList1 :: BinaryTree α → [α]

TreeToList1 EMPTY = [ ]
TreeToList1 (LEAF a) = [a]
TreeToList1 (NODE subtree1 subtree2) = (TreeToList1 subtree1)
                                       ++ (TreeToList1 subtree2)
```

TreeToList2:

```
TreeToList2 :: BinaryTree α → [α]

TreeToList2 EMPTY = [ ]
TreeToList2 (LEAF a) = [a]

TreeToList2 (NODE evensubtree oddsubtree)
            = (merge (TreeToList2 evensubtree) (TreeToList2 oddsubtree))
              where
              merge (a0 : evens) (a1 : odds) = a0 : a1 : (merge evens odds)
              merge as [ ] = as
```

until:

```
until :: (Num → Bool) → [Num] → Num

until predicate xs = select (map predicate (from 0)) xs
                     where
                     select (FALSE : tests) (x : xs) = select tests xs
                     select (TRUE : tests) (x : xs) = x
```

VectorBound:

VectorBound :: $<\alpha>$ → Num

We specify that

VectorBound (MakeVector bound f) = bound

VectorToList:

VectorToList :: $<\alpha>$ → $[\alpha]$

This is specified by the requirements that

(VectorToList (MakeVector bound f)) **sub** i = (MakeVector bound f) **sub** i

and

length (VectorToList (MakeVector bound f)) = bound

Appendix C

Programming in a real functional language

The programming language used in this book is not precisely the same as any commonly-available programming language. In fact only a small part of a real programming language is used, so the translation process required is really very small. It differs only superficially from several more accessible languages:

- Miranda[1] [Tur86]. A Miranda interpreter and program development environment is commercially available from its originator, D.A. Turner.
- Orwell (available at Oxford University)
- Lazy ML [Aug84]
- Haskell [HWA+88]. This language proposal will hopefully result in a widely accessible public-domain implementation, but none exists at the time of writing.

The language SASL (also originated by Turner) may be suitable for experimentation. For our purposes its resembles Miranda but lacks a type system. A third, similar, language implementation distributed by Turner's group, KRC, is not suitable because it lacks the **where** construct.

The LispKit system, which is described in Henderson's textbook [Hen80], and SUGAR, described by Glaser, Hankin and Till [GHT84] might also be suitable for experimentation, but lack pattern-matching as well as a type system.

Strict languages

The language used here is lazy: a parameter expression is evaluated only if and when the application in which it appears needs its value to return a result. An implementation must employ normal-order reduction (see page 51), unless strictness analysis indicates that applicative order will be safe.

In a strict (that is, call-by-value) language, a parameter is always evaluated before it is passed to the function body. Implementations of such languages are much more common.

[1] Miranda is a trademark of Research Software Ltd.

Unfortunately translating a program written in a lazy language into a program which will work under a strict interpretation is quite complicated, and not recommended. Details are given in [GHT84].

C.1 Differences from Miranda

The main purpose of this appendix is to give enough information for the programs in this book to be tried out under the Miranda system. Users of other implementations must glean what they can. Reasons for the differences are summarised at the end of the appendix.

Lexical convention for constructors

All constructors (e.g. NIL, CONS, LEAF, NODE, etc.) appeared in upper case, while all other identifiers were of mixed case. In Miranda, any identifier starting with a capital is defined to be a constructor, and all other names must begin in lower case.

Type variables

Type variables were referred to as α, β, γ etc. In Miranda they are written *, **, *** etc.

Pattern matching and guards

Miranda's syntax includes ours as a special case, but does not demand that all equations defining an object be mutually exclusive. Miranda's semantics differs: patterns are tested sequentially from the top of the page downwards. This means that functions like the non-strict or of page 52 will not work as expected. The components of a particular equation's pattern are also tested sequentially, in an unspecified order.

Vectors and matrices

Miranda has no vectors or matrices. Lists and lists of lists can be used instead, provided efficiency is not a serious concern.

Built-in operators

Present-day keyboards have tied Miranda to forms like "<=" where "$\leq$" appears in this book, "*" for "$\times$", "->" for "$\rightarrow$", "&" for "$\wedge$" and so on. Function composition, f $\circ$ g, is written "f . g" in Miranda. There is no counterpart to "$\circ\circ$".

The subscripting operator **sub** is used for lists, vectors and matrices. In Miranda, its only counterpart is the infix "!" operator for indexing lists.

The list type and its shorthand

The [a, b, ...], ":" and [ ] notations were introduced as shorthand for a list data type defined by

List α ::= NIL | CONS α (List α)

In Miranda they are different (but isomorphic) types.

C.1.1 Examples

Binary trees: as in this book:

```
BinaryTree α ::= EMPTY |
                 LEAF α |
                 NODE (BinaryTree α) (BinaryTree α)
```

In Miranda:

```
binary_tree * ::= Empty |
                  Leaf * |
                  Node (binary_tree *) (binary_tree *)
```

Square root: as in this book:

```
sqrt :: Num → num

sqrt a = until converges xs
         where
         converges 0 = FALSE
         converges (i+1) = abs( ((xs sub (i+1)) − (xs sub i))/(xs sub(i+1)) ) ≤ ε
         xs = generate NextEstimate
              where
              NextEstimate 0 = a/2
              NextEstimate (i+1) = ((xs sub i) + a/(xs sub i))/2


until :: (Num → Bool) → [Num] → Num

until predicate xs = select (map predicate (from 0)) xs
                     where
                     select (FALSE:tests) (x:xs) = select tests xs
                     select (TRUE:tests) (x:xs) = x
```

In Miranda:

```
sqrt :: num -> num

sqrt a = until converges xs
         where
         converges 0 = False
         converges (i+1) = abs( ((xs ! (i+1))
```

```
                          - (xs ! i))/(xs ! (i+1)) ) <= epsilon
        xs = generate NextEstimate
             where
             NextEstimate 0 = a/2
             NextEstimate (i+1) = ((xs ! i) + a/(xs ! i))/2

until :: (num -> bool) -> [num] -> num

until predicate xs = select (map predicate (from 0)) xs
                     where
                     select (False:tests) (x:xs) = select tests xs
                     select (True:tests) (x:xs) = x
```

C.2 Reasons for the differences

The lexical differences, such as the uses of "→" instead of "->", and the admission of capitals in ordinary identifiers, were simply to improve readability, at the suggestion of the reviewers.

The only significant change is in the rules concerning overlapping patterns and guards. When reasoning about programs, it is important to be able to treat equations independently of one another, so they must not overlap. However, in a practical programming language design different criteria apply:

- **Guards:** in a programming language, it is important that a compiler be able to verify that a program is well formed. To check whether guards overlap is not computable in general. By contrast, in a language used for specifying and verifying programs, the onus is on the human.

- **Patterns:** patterns differ from guards because a compiler can perform a full analysis of overlapping and missing cases. The sequential order of testing patterns used by Miranda simplifies and shortens programs: one can write

  EachElement (0,0) = edge
  EachElement (i,0) = edge
  EachElement (0,j) = edge
  EachElement (i,j) = f (a **sub** (i,j−1)) (a **sub** (i−1,j))

  whereas we had to introduce guards to disambiguate the four equations.

A simple resolution of this could be to introduce a syntactic construct for definition by sequential pattern/guard matching.

Miranda and most other languages with pattern matching have avoided having to use a parallel pattern testing mechanism like our general normalisation strategy because it is difficult to implement without a large run-time overhead. This leads to rather more awkward rules for program syntax and semantics than ours. By contrast, committed choice logic languages like PARLOG exploit parallelism in guard evaluation as a positive feature.

Tagged tuples

One subtlety touched on only slightly in the text is the use of constructors to tag data types which are really only aggregates of their components. The main example (see page 180) was

PipeItem α α ::= PIPEITEM α α

The constructor PIPEITEM was introduced solely so provide a visual cue to what is going on. It does not serve to distinguish different alternative cases. We assumed that we could have used the pair type

PipeItem α α == (α, α)

instead. In Miranda they are nearly but not precisely equivalent. Suppose we define

f (PIPEITEM a b) = TRUE

In Miranda, f $\perp$ = $\perp$. By contrast, if we write

g (a, b) = TRUE

then f $\perp$ = TRUE. In Miranda, the rule is that the parameter must be evaluated enough to uncover the constructors appearing on the LHS before the equation can be applied. Some other languages, notably Haskell, treat data types with just one alternative as a special case. Such a constructor is called "irrefutable", and a precise equivalence with pairs holds.

Bibliography

[AB84] Arvind and J.D. Brock. Resource managers in parallel programming. *Journal of Parallel and Distributed Computing*, 1:5–21, 1984.

[ACE88] Arvind, David E. Culler, and Kattamuri Ekanadham. The price of asynchronous parallelism: An analysis of dataflow architectures. 1988. In [JR89].

[AE88] Arvind and Kattamuri Ekanadham. Future scientific programming on parallel machines. In *Supercomputing: 1st International Conference Proceedings, Athens, Greece, June 1987*, pages 639–686. Springer Verlag, June 1988.

[AGM] Samson Abramsky, Dov Gabbay, and Tom Maibaum. *Handbook of Logic in Computer Science*. Oxford University Press. Forthcoming.

[AH87] S. Abramsky and C.L. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[AI 88] AI Limited. STRAND-88 language definition. Technical report, AI Limited, Greycaine Rd. Watford, Herts, UK, 1988.

[AI86] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessing. Memo 226–5, MIT Computation Structures Group, July 1986. Reprinted in [Tha87, pages 140–164.].

[AN87] Arvind and R.S. Nikhil. Executing a program in the MIT tagged dataflow architecture. 1987. In [dBNT87a, pages 1–29].

[AN88] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: A new loop parallelisation technique. 1988. In [Gan88, pages 221–235].

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Joint Computer Conference*, volume 32, pages 37–45, 1968.

[AS85] S. Abramsky and R. Sykes. SECD-M: a virtual machine for applicative multiprogramming. In *Functional Programming and Computer Architecture*. Springer Verlag, 1985. LNCS 201.

[ASS85] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[Aug84] L. Augustsson. A compiler for Lazy ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.

[Bar84] H.P. Barendregt. *The Lambda Calculus – Its Syntax and its Semantics.* Number 3 in Studies in Logic. North Holland, second edition, 1984.

[BBK87] D.I. Bevan, G.L. Burn, and R.J. Karia. Overview of a parallel reduction machine project. 1987. In [dBNT87b, pages 394–411].

[BC87] D.A. Bailey and J.E. Cuny. An approach to programming process interconnection structures: Aggregate rewriting graph grammars. 1987. In [dBNT87a, pages 112–123].

[BCM87] Duane A. Bailey, Janice E. Cuny, and Bruce B. MacLeod. Reducing communication overhead: a parallel code optimisation. *Journal of Parallel and Distributed Computing*, 4:505–520, 1987.

[Bev87] D.I. Bevan. Distributed garbage collection using reference counting. 1987. In [dBNT87a, pages 176–187].

[BGS82] R.A. Brooks, R.P. Gabriel, and G.L. Steele. An optimising compiler for lexically-scoped Lisp. *SIGPLAN Notices*, 17(6), June 1982.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice Hall, 1982.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[BMS80] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 Lisp Conference, Stanford, California*, pages 136–143, August 1980. Also Univ. of Edinburgh Dept. of Computer Science report CSR-62-80.

[Boo80] R. Book. *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

[Bur84a] R.M. Burstall. Programming with modules and typed functional programming. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 103–112. ICOT, Japan, 1984.

[Bur84b] F.W. Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Transactions on Programming Languages and Systems*, 6(2):159–174, April 1984.

[Bur87a] G.L. Burn. Evaluation transformers – a model for the parallel evaluation of functional languages (extended abstract). 1987. In [Kah87].

[Bur87b] F.W. Burton. Functional programming for concurrent and distributed computing. *The Computer Journal*, 30(5):437–450, 1987.

[BvEG+87a] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an intermediate language for graph rewriting. 1987. In [dBNT87b, pages 159–174].

[BvEG+87b] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. 1987. In [dBNT87b, pages 141–158].

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[CDJ84] F.B. Chambers, D.A. Duce, and G.P. Jones, editors. *Distributed Computing.* Academic Press, 1984.

[CGMN80] T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, and A.C. Norman. SKIM – the s, k, i reduction machine. In *Proceedings of the 1980 ACM Lisp Conference*, pages 128–135, August 1980.

[Che84] Marina C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Conference Record of the 11th Annual ACM Symposium on the Principles of Programming Languages.* ACM, June 1984.

[Chu41] A. Church. *The Calculi of λ-conversion.* Princeton University Press, 1941.

[CL73] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[CT77] K.L. Clark and S.A. Tarnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Information Processing 77: Proceedings of the IFIP Congress 77*, pages 939–944. Elsevier/North-Holland, 1977.

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman/Wiley, 1986.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.

[Dar81] J. Darlington. The structured description of algorithm derivations. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 221–250. North-Holland Publishing Company, 1981.

[Dar82] J. Darlington. Program transformation. 1982. In [DHT82].

[dBNT87a] J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors. PARLE, *Parallel Architectures and Languages Europe*, volume II. Springer Verlag, June 1987. LNCS 259.

[dBNT87b] J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors. PARLE, *Parallel Architectures and Languages Europe*, volume I. Springer Verlag, June 1987. LNCS 258.

[DCF+87] John Darlington, Martin Cripps, Tony Field, Peter G. Harrison, and Mike J. Reeve. The design and implementation of ALICE: a parallel graph reduction machine. 1987. In [Tha87].

[De88] John Darlington and et al. An introduction to the FLAGSHIP programming environment. 1988. In [JR89].

[Deu85] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London*, A(100):97–117, 1985.

[DHT82] J. Darlington, P. Henderson, and D.A. Turner. *Functional Programming and its Applications*. Cambridge University Press, 1982.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations and Equations*. Prentice-Hall, 1986.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*. ACM SIGPLAN, 1982.

[Eck46] J.P. Eckert, Jr. A parallel channel computing machine. 1946. Lecture 45 in [GAB+48].

[Ell82] J.R. Ellis. BULLDOG*: a Compiler for VLIW Architectures*. MIT Press, 1982.

[Ers82] A.P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[Fai82] J. Fairbairn. *The Design and Implementation of a Simple Untyped Language based on the λ-calculus*. PhD thesis, University of Cambridge, 1982.

[Fau82] A.A. Faustini. *The Equivalence of an Operational and a Denotational Semantics for Pure Dataflow*. PhD thesis, Department of Computer Science, University of Warwick, April 1982.

[Fea86] M.S. Feather. A survey and classification of some program transformation approaches and techniques. IFIP TC-2 Working Conference on Program Specification and Transformation (preprint), Bad Tolz, F.R.G., April 1986.

[FH88] A.J. Field and P. Harrison. *Functional Programming.* Addison Wesley, 1988.

[FK86] J.H. Fasel and R.M. Keller, editors. *Graph Reduction: Proceedings of a Workshop, Santa Fe, New Mexico, USA.* Springer Verlag, 1986. LNCS 279.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. 1988. In [Gan88].

[FPZ88] M.P. Fourman, W.J. Palmer, and R.M. Zimmer. Core tools for the next generation of electronics CAD. In *UK IT 88 Conference Publication*, pages 428–430, Department of Trade and Industry, Kingsgate House, 66–74 Victoria St. London, 1988. Information Engineering Directorate.

[FS88] Y. Feldman and E. Shapiro. Spatial machines: Towards a more realistic approach to parallel computation. Technical report, Department of Computer Science, Weizmann Institute for Science, Rehovat 76100, Israel, 1988.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. 1987. In [Kah87].

[GAB+48] H.H. Goldstine, H.A. Aitken, A.W. Burks, J.P. Eckert, Jr., J.B. Mauchly, and J. von Neumann. Theory and techniques for design of electronic digital computers, lectures given at the moore school of engineering, university of philadelphia, 8th july – 31 august 1946. Reports 48–7 to 48–10, Moore School of Electrical Engineering, University of Pennsylvania, June 1948. Reprinted as [GAB+85].

[GAB+85] H.H. Goldstine, H.A. Aitken, A.W. Burks, J.P. Eckert, Jr., J.B. Mauchly, and J. von Neumann. *The Moore School Lectures*, volume 9 of *Charles Babbage Institute Reprint Series for the History of Computing.* MIT Press and Tomash Publishers, 1985.

[Gan88] H. Ganzinger, editor. *ESOP '88: the second European Symposium on Programming.* Springer Verlag, 1988. LNCS 300.

[GH86a] Hugh Glaser and Sean Hayes. Another implementation technique for applicative languages. In *Proceedings of ESOP '86: the European Symposium on Programming, Saarbrucken*, pages 70–81. Springer Verlag, 1986. LNCS 213.

[GH86b] B. Goldberg and P. Hudak. Alfalfa: Distributed graph reduction on a hypercube multiprocessor. Preprint, Yale University Department of Computer Science, November 1986.

[GH86c] J.V. Guttag and J.J. Horning. Report on the LARCH shared language. *Science of Computer Programming*, 6:103–134, 1986.

[GHT84] H. Glaser, C.L. Hankin, and D. Till. *Principles of Functional Programming.* Prentice Hall, 1984.

[GKS87] J.R.W. Glauert, J.R. Kennaway, and M.R. Sleep. DACTL: a computational model and compiler target language based on graph reduction. Report SYS-C87-03, School of Information Systems, University of East Anglia, 1987.

[GKW85] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), January 1985.

[GM85] Narain Gehani and Andrew McGettrick, editors. *Software Specification Techniques.* Addison Wesley, 1985.

[GM86] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-oriented Programming*, pages 295–363. MIT Press, 1986.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Springer Verlag, 1979. LNCS 78.

[Gog88] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. Report SRI-CSL-88-1, SRI International, Menlo Park, CA 94025, USA, January 1988.

[Gol87] Benjamin Goldberg. Detecting sharing of partial applications in functional programs. 1987. In [Kah87].

[Gol88] Benjamin F. Goldberg. *Multiprocessor Execution of Functional Programs.* Research report, Yale University Department of Computer Science, April 1988.

[Gor88] Michael J.C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall, 1988.

[GPKK82] D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, pages 58–69, February 1982. Reprinted in [Tha87, pages 165–176].

[Gre87] S. Gregory. *Parallel Logic Programming in Parlog.* Addison-Wesley, 1987.

[Gri71] D. Gries. *Compiler Construction for Digital Computers.* Wiley, 1971.

[GT79] Joseph Goguen and Joseph Tardo. An introduction to OBJ: a language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, 1979. Reprinted in [GM85].

[GT84] Hugh W. Glaser and Phil Thompson. Lazy garbage collection. *Software Practice and Experience*, 17(1):1–4, 1984.

[HB84] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th Annual Symposium on Principles of Programming Languages*. ACM SIGPLAN, 1984.

[HBJ88] Chris Hankin, Geoff Burn, and Simon Peyton Jones. A safe approach to parallel combinator reduction. *Theoretical Computer Science*, 56(1):17–36, January 1988.

[HD85] F.K. Hanna and N. Daeche. Specification and verification using higher-order logic. In C.J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications*. Elsevier Science Publishers B.V. (North Holland), 1985.

[Hen80] P. Henderson. *Functional Programming, Application and Implementation*. Prentice Hall, 1980.

[HG84] P. Hudak and B. Goldberg. Experiments in diffused combinator reduction. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[HG85] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. *IEEE Transactions on Computers*, C-34(10), October 1985.

[HK84] Paul Hudak and David Kranz. A combinator-based compiler for a functional language. In *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, pages 122–132. ACM SIGACT/SIGPLAN, 1984.

[HM76] Peter Henderson and James H. Morris. A lazy evaluator. Technical Report Series, number 85, Computing Laboratory, University of Newcastle upon Tyne, UK, 1976.

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules. Technical Report CSL-111, SRI International, Menlo Park, California, USA, January 1980. Also in [Boo80].

[HOS85] C.M. Hoffman, M.J. O'Donnell, and R.I. Strandh. Implementation of an interpreter for abstract equations. *Software Practice and Experience*, 15(12):1185–1204, December 1985.

[Hud86a] P. Hudak. The denotational semantics of a para-functional programming language. Report YALEU/DCS/TR-484, Yale University Department of Computer Science, July 1986.

[Hud86b] P. Hudak. Para-functional programming. *IEEE Computer*, pages 60–70, August 1986.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. 1987. In [AH87].

[Hug83] J. Hughes. The design and implementation of programming languages. Technical Monograph PRG-40, Oxford University Programming Research Group, July 1983.

[Hug84] J. Hughes. Why functional programming matters. Report 16, Programming Methodology Group, University of Göteborg and Chalmers Institute of Technology, Sweden, November 1984.

[Hug87] John Hughes. Backwards analysis of functional programs. Departmental Research Report CSC/87/R3, Department of Computer Science, University of Glasgow, March 1987.

[HV87] Pieter H. Hartel and Willem G. Vree. Parallel graph reduction for divide-and-conquer applications. Preprint, Computing Science Department, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam, February 1987.

[HWA+88] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, J. Launchbury, S. Peyton Jones, R. Nikhil, M. Reeve, D. Wise, and J. Young. Report on the functional programming language Haskell. Draft proposed standard circulated by IFIP WG.2.8., December 1988.

[HZ83] E. Horowitz and A. Zorat. Divide and conquer for parallel processing. *IEEE Transactions on Computers*, C-32(6):582–585, 1983.

[JCH85] S.L. Peyton Jones, C. Clack, and N. Harris. GRIP — a parallel graph reduction machine. Internal Note 1665, Dept. of Computer Science, University College London, February 1985.

[Joh84a] S.D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* Distinguished Dissertation Series. MIT Press, 1984.

[Joh84b] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984. Published as ACM SIGPLAN Notices Vol. 19 no. 6.

[Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Jon84] S.B. Jones. A range of operating systems written in a purely functional style. Report TR.16, University of Stirling, Dept. of Computer Science, September 1984.

[Jon87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[JR89] C.R. Jesshope and K.D. Reinartz, editors. *CONPAR 88*. Cambridge University Press, 1989. (to appear).

[JW75] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer Verlag, second edition, 1975.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. 1988. In [Gan88].

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*. North-Holland, 1974.

[Kah87] Gilles Kahn, editor. *Functional Programming Languages and Computer Architecture, Portland, Oregon*. Springer Verlag, 1987. LNCS 274.

[Kaj83] J.T. Kajiya. New techniques for ray tracing procedurally-defined objects. *ACM Transactions on Graphics*, 2(3), July 1983.

[KE84] G. Kedem and J.L. Ellis. The raycasting machine. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers (ICCD '84)*, pages 533–538, October 1984.

[Kel74] R.M. Keller. A fundamental theorem of asynchronous parallel computation. In T. y. Feng, editor, *Parallel Processing, Proceedings of the Sagamore Computer Conference*. Springer Verlag, August 1974. LNCS 24.

[Kel77] R.M. Keller. Look-ahead processors. *Computing Surveys*, 9(1), March 1977.

[KKLW81] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218. ACM, January 1981.

[KKR+86] D.A. Kranz, R. Kelsey, J.A. Rees, P. Hudak, J. Philbin, and N.I. Adams. ORBIT: an optimizing compiler for scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986. See also [Kra88].

[KL82] R.M. Keller and G. Lindstrom. Approaching distributed database implementations through functional programming concepts. Technical Report UUCS 82-013, Department of Computer Science, University of Utah, June 1982.

[KL84] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. *IEEE Computer*, 17(7), July 1984.

[Klo90] J.W. Klop. Term rewriting systems. 1990. In [AGM].

[Kog81] P.M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

[Kra88] David A. Kranz. ORBIT*: An Optimizing Compiler for Scheme.* Research report, Department of Computer Science, Yale University, February 1988. See also [KKR+86].

[KSL86] Robert M. Keller, Jon W. Slater, and Kevin T. Likes. Overview of Rediflow II development. 1986. In [FK86, pages 203–214].

[KSS81] H.-T. Kung, R. Sproull, and G. Steele, editors. *VLSI Systems and Computations.* Computer Science Press, Rockville, Md, 1981.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Lee88] Ching-Cheng Lee. Experience of implementing applicative parallelism on CRAY-XMP. 1988. In [JR89].

[LH83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[May87] D. May. Communicating processes and occam. Technical Note 20, Inmos Ltd., 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, UK, 1987.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison Wesley Publishing Company, 1980.

[McC67] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems.* North-Holland, 1967.

[McG82] J.R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, 1982.

[Mil83] R. Milner. A proposal for standard ML. Internal Report CSR-157-83, Dept. of Computer Science, University of Edinburgh, 1983.

[Mil84] R. Milner. Using algebra for concurrency. 1984. In [CDJ84].

[Mil85] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages.* Prentice-Hall International, 1985. Also *Philosophical Transactions of the Royal Society*, Series A, Vol. 312 (1984).

[MNV73] Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8), August 1973.

[Mog87] Torben Mogenson. The application of partial evaluation to ray-tracing. Preprint, Institute of Datalogy, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, Denmark, 1987.

[Mol83] D.I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[Moo84] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 Symposium on Lisp and Functional Programming*, pages 235–246. ACM, 1984.

[MPT85] J.D. Morison, N.E. Peeling, and T.L. Thorp. The design rationale for ELLA, a hardware design and description language. In C.J. Koomen and T. Motooka, editors, *Computer Hardware Description Languages and their Applications*. Elsevier Science Publishers B.V. (North Holland), 1985.

[MSA+85] J.R. McGraw, S.K. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: reference manual. Manual M-146, Rev.1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[NPA86] Rishiyur S. Nikhil, Keshav Pingali, and Arvind. Id nouveau. CSG Memo 265, MIT Laboratory for Computer Science, Cambridge, MA 02139, July 1986.

[Pap89] Gregory M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachussetts Institute of Technology, 77, Massachussetts Ave, Cambridge MA 02139, June 1989. (expected).

[PCSH87] S.L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP: a high performance architecture for parallel graph reduction. 1987. In [Kah87].

[Pep83] P. Pepper, editor. *Program Transformation and Programming Environments*, volume 8 of *NATO ASI Series F (Computer and System Sciences)*. Springer Verlag, 1983.

[PM87] Dick Pountain and David May. *A Tutorial Introduction to Occam Programming*. BSP Professional Books, Ornsley Mead, Oxford OX2 0EL, UK, 1987.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th International Symposium on Computer Architecture*, June 1984.

[RL77] C.V. Ramamoorthy and H.F. Li. Pipeline architecture. *Computing Surveys*, 9(1), March 1977.

[RSC88] V.J. Rayward-Smith and A.J. Clark. Scheduling divide-and-conquer task systems on identical parallel machines. 1988. In [JR89].

[Rus78] R.M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1), January 1978.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* Pitman/MIT Press, 1989.

[Sch86] D.A. Schmidt. *Denotational Semantics, A Methodology for Language Development.* Allyn and Bacon, inc., 1986.

[She88] Mary Sheeran. Retiming and slowdown in RUBY. In G. Milne, editor, *Proceedings of the IFIP WG10.2 International Workshop on Design for Behavioural Verification.* North Holland, 1988.

[SM77] I.E. Sutherland and C.A. Mead. Microelectronics and computer science. *Scientific American*, 236(9):210–228, September 1977.

[Ste78] Guy L. Steele Jr. RABBIT: a compiler for scheme. AI Laboratory Technical Report 474, Artificial Intelligence Laboratory, Massachussetts Institute of Technology, May 1978.

[Ste84] G. L. Steele. *Common Lisp.* Digital Press, Burlington, Massachusetts, 1984.

[Sto77] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Sto85] William Stoye. *The Implementation of Functional Languages using Custom Hardware.* PhD thesis, University of Cambridge, December 1985. Published as Computer Laboratory Technical Report Number 81.

[Sto87] Quentin F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal of Parallel and Distributed Computing*, 4:95–115, 1987.

[Suf82] Bernard Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1:157–202, 1982.

[Tha87] S.S. Thakkar, editor. *Selected Reprints on Dataflow and Reduction Architectures.* IEEE Computer Society Press, 1987.

[Tho81] C.D. Thompson. The VLSI complexity of sorting. 1981. In [KSS81, pages 108–118].

[TK77] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected computer. *Communications of the ACM*, 29(4):263–271, 1977.

[TMS87] Jack A. Test, Mat Myszewski, and Richard C. Swift. The Alliant FX/Series: A language driven architecture for parallel processing of dusty deck Fortran. 1987. In [dBNT87b, pages 345–356].

[Tur79] David A. Turner. A new implementation technique for applicative languages. *Software - Practice And Experience*, 9(1):31–49, 1979.

[Tur86] D.A. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12), December 1986.

[Ull84] J.D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[Val81] L.G. Valiant. Universality considerations in VLSI circuits. *IEEE Transactions on Computers*, C-30(2):135–140, 1981.

[vdBvdH86] P.M. van den Broek and G.F. van der Hoeven. Combinatorgraph reduction and the Church-Rosser theorem. Preprint INF-86-15, Department of Informatics, Twente University, The Netherlands, 1986.

[Vit86] Paul M.B. Vitányi. Nonsequential computation and laws of nature. In *VLSI Algorithms and Architecture, Loutraki, Greece*. Springer Verlag, July 1986. LNCS 227.

[Vre87] Willem G. Vree. The grain size of parallel computations in a functional program. Preprint, Computer Science Department of the Dutch Water Board Authority, Nijverheidsstraat 1, Post Box 5809, 2280 Rijswijk ZH, The Netherlands, 1987.

[WA85] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[Wad81] W.W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[Wad86] P. Wadler. Plumbers and dustmen: Fixing a space leak with a garbage collector. Article distributed on the Functional Programming Mailing List (moderated by J. Glauert, University of East Anglia)., April 1986.

[Wad87] P. Wadler. Strictness on non-flat domains. 1987. In [AH87].

[Wad88a] P. Wadler. The concatenate vanishes. Article distributed on the Functional Programming Mailing List (moderated by J. Glauert, University of East Anglia)., January 1988.

[Wad88b] Philip Wadler. Deforestation: Transforming programs to eliminate trees. 1988. In [Gan88, pages 344–358].

[WF84] C.-l. Wu and T.-y. Feng, editors. *Tutorial: Interconnection Networks for Parallel and Distributed Processing*. IEEE Computer Society Press, 1984.

[WHG84] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1), January 1984.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6), June 1980.

[Wil84] R. Wilensky. *LISPcraft*. W.W. Norton, 1984.

[Wil88] Paul T.G. Williams. An approach to the design of a parallel functional language. Master's thesis, Department of Computing, Imperial College, London, 1988.

[WJW+75] W. Wulf, R.K. Johnson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke. *The Design of an Optimizing Compiler*. Elsevier, New York, 1975.

[Wol89] Michael Wolfe. *Optimising Supercompilers for Supercomputers*. Pitman/MIT Press, 1989.

[WSWW87] Ian Watson, John Sarjeant, Paul Watson, and Viv Woods. Flagship computational models and machine architecture. *ICL Technical Journal*, 5(3):555–574, May 1987.

[WW87] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. 1987. In [dBNT87a, pages 432–443].

[WW88] J.H. Williams and E.L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Proceedings of the Fifteenth Annual Symposium on Principles of Programming Languages, San Diego, USA*. ACM SIGPLAN, 1988.

[You85] M.F. Young. A functional language and modular architecture for scientific computing. In *Functional Programming Languages and Computer Architecture, Nancy, France*. Springer Verlag, September 1985. LNCS 201.

Index