Large Scale Numerical Software Development Using Functional Languages

A
THESIS
submitted in fulfilment of the requirements
for the degree of
DOCTOR OF PHILOSOPHY
in the
Department of Computing Science
Newcastle University
NE1 7RU

Chris Angus B.Sc. (Hons) (University of Leeds)

May, 1998

NEWCASTLE UNIVERSITY LIBRARY
098 50718 3
Thesis L6264

Abstract

Functional programming languages such as Haskell allow numerical algorithms to be expressed in a concise, machine-independent manner that closely reflects the underlying mathematical notation in which the algorithm is described. Unfortunately the price paid for this level of abstraction is usually a considerable increase in execution time and space usage.

This thesis presents a three-part study of the use of modern purely-functional languages to develop numerical software.

- In Part I the appropriateness and usefulness of language features such as polymorphism, pattern matching, type-class overloading and non-strict semantics are discussed together with the limitations they impose. Quantitative statistics concerning the manner in which these features are used in practice are also presented.
- In Part II the information gathered from Part I is used to design and implement FSC, an experimental functional language tailored to numerical computing, motivated as much by pragmatic as theoretical issues. This language is then used to develop numerical software and its suitability assessed via benchmarking it against C/C++ and Haskell under various metrics.
- In Part III the work is summarised and assessed.

Acknowledgements

Many thanks to Dr Chris Phillips and Dr Dave Harrison for their support and guidance over the past three years. Personal thanks go to my family, friends and especially Tor and Rob. This work was supported by EPSRC grant number 94313569.

Contents

Chapter 1

Introduction

The theme of this thesis is the development of efficient numerical software using modern functional languages together with the design decisions, pragmatic considerations and techniques required from a modern functional language specialised to this area.

Typically, numerical software is based on algorithms associated with discrete approximation and linear algebra. These algorithms often have high computational requirements and dominate the field of scientific computing.

An efficient implementation of an algorithm expressed in a procedural language is often very different from the underlying mathematical definition.

The use of functional programming languages for scientifically-significant computations offers the advantages over procedural languages of modularity, conciseness, clarity, ease of proof, ease of introducing problem-oriented notation and independence from hardware peculiarities [15].

In the field of scientific computing, domain-specific knowledge also plays a vital role as improvements in efficiency are often possible when problem properties are known. This thesis discusses how many of the ideas and techniques available in modern functional languages may be used in the context of numerical software development without suffering the lack of efficiency normally associated with functional languages; how domain specific knowledge may be expressed within this framework; and the necessary pragmatic concessions which must be made to achieve these goals.

The test-bed for these concepts is the development of numerical software in an experimental functional language, FSC. FSC is essentially a Haskellesque[49, 50] language specialised for numerical computation compiling to SISAL[19] .

1.1 The Problem

Since computer performance is increasing all the time we are now in the position of being able to solve problems of greater complexity than was previously possible. Unfortunately it is a sad fact that as problem complexity increases, program complexity often increases also[1]. It is for exactly this reason that clarity, conciseness and correctness of code are important. A *"functional (or applicative) style is necessary to realise the promise of general-purpose parallel programming"* [105]. This promise draws on the fact that there is no underlying concept of a physical machine, store or sequencing, and although implicit parallel functional programming has often been promised, and to an extent delivered, we believe that in the area of scientific programming this promise can be achieved[2], with applicative languages such as SISAL [18, 19] substantiating this claim.

1.1.1 Functional Programming and Scientific Computing.

The main advantages of using a functional language in this application domain are clear; programs can be written more quickly, more concisely, at a higher level (resembling more closely traditional mathematical notation) and are more amenable to formal reasoning, analysis and transformation-based compilation. Some of the disadvantages in this area are also clear; I/O is not as straightforward as in imperative languages[3]; low-level machine actions are hard to program; often the price paid for the level of abstraction is an increased execution time and space usage (especially for lazy functional languages). We also need to consider issues relating to numerical analysis; execution efficiency is paramount; legacy code exists and will continue to do so for many years to come[4]. In the design of any realistic tool for work in this area each of these points should be taken into account, maximising the advantages and minimising the disadvantages while still respecting the numerical analysis aspects, making necessary compromises along the way between mathematical/theoretical elegance and pragmatic considerations.

1.2 Summary of Original Work

The original work reported in this thesis consists of:

- A study of the suitability and use of the purely functional language Haskell[49, 50] to implement numerical algorithms.
- An investigation of the development of algorithms in a style differing from the usual array-intensive definition.
- A non-pivoting version of LU factorisation which does not require all leading left submatrices to be non-singular and which does not introduce *fill-in*.

[1] Picking through a PVM or MPI message-passing code should convince the reader of this.

[2] Given that the algorithm is not strictly sequential.

[3] This is as much a matter of unfamiliarity as conceptual difficulty especially with the increasing use of monads (see later).

[4] Asking practitioners to re-do 40+ years of work is not feasible.

- Quantitative statistics regarding the manner and frequency with which specific functional-language features are used in practice.
- The design, development and evaluation of FSC, a strict, Haskell-like functional language specialised to scientific computing.
- The definition of a user-defined transformation language enabling domain-specific knowledge and optimisations to be embedded in a program script and a monadic translation scheme for the interpretation of such transformations.
- The development of a purely functional I/O system, suited to this area.
- The development of an array abstract datatype which may be used in conjunction with pattern-matching expressions, translation schemes for syntactic sugarings of array operations/definitions and optimisations based on the GCD[5]-test avoiding the need for intermediate-array building.
- A scheme for efficiently constructing recursively-defined arrays.
- The development and evaluation of an efficient multi-parameter type-class system suited to the area of scientific computing.

1.3 Overview of Thesis

The central theme of this thesis is the use of techniques from modern functional programming to produce clear efficient implementations of numerical algorithms in a manner that preserves as much of the spirit of functional programming as possible.

This thesis is organised in three parts. Part I acts as an introduction to functional scientific computing and describes work carried out using existing modern functional languages. In Part II the language FSC is introduced and used to construct numerical software. Part III offers conclusions and suggests further work.

PART I: Functional Implementations of Numerical Methods

Chapter 2: Functional Programming
The main concepts and idioms behind functional programming are introduced together with a discussion of commonly used functional (or mostly functional) programming languages.

Chapter 3: Numerical Methods
The problem domain we are working in is introduced and a brief overview of scientific computing is presented.

[5]GCD-test: A dependence test used in parallelising compilers based around use of the greatest common divisor.

Chapter 4: Use of Haskell to Implement Numerical Algorithms.
In this chapter, use of the purely functional language Haskell to implement numerical algorithms is described, the development of numerical algorithms in styles other than the usual array intensive definition is investigated and the impact of non-strict semantics on this area is discussed.

Chapter 5: Efficiency and Empirical Analysis
In this chapter initial conclusions are presented as to the suitability of Haskell's language features to scientific computing.

Chapter 6: Related work
Part I is concluded with a survey of the previous work in the area of functional scientific computing.

PART II: The FSC Language

Chapter 7: Design of FSC
In this chapter the necessary features for a functional language specialised to numerical analysis are discussed.

Chapter 8: Definition of FSC
A formal definition of FSC is presented.

Chapter 9: Implementation of FSC
A discussion of the pragmatic implementation details is given.

Chapter 10: Use of FSC in practice
Extended examples in FSC are presented and the efficiency of these examples discussed.

Chapter 11: Related work
A survey of work related to techniques used in Part II is presented.

PART III: Conclusions

Chapter 12: Conclusions
The work presented is summarised and assessed.

1.4 Notational Conventions

Footnotes are used to add minor points of non-essential information to the body of the text; footnotes are denoted by Arabic numerals[6]. More extensive notes are collected at the end of each chapter and are indicated by Roman numerals(i). Both footnotes and chapter notes may be ignored on a first reading.

References to other work are indicated by a series of numbers in square brackets. e.g. [1,2].

[6] A footnote.

Mathematical expressions are written in italic font: $id = \lambda x.x$.

Program code fragments are displayed in typewriter font: `x = y` and, unless specified as otherwise, larger Haskell, FSC and Miranda[96] code sections are written as *literate* scripts [60] using the Haskell offside rule, i.e. code is commented *in* rather than *out* and its column placement has meaning.

```
A variation on a familiar program in FSC

>     main :: IO ()
>     main = do IO w where
>        w = print x

            due to its column placing the
            definition below is local to
            main and not global

>        x = "hello world" ++ endl
```

1.5 Chapter Notes

i (page 4) :
Chapter notes are cross-referenced with the page to which the note refers.

Part I

Functional Implementations of Numerical Methods

Chapter 2

Functional Programming

2.1 Introduction

It has often been claimed that the use of functional programming languages for scientifically-significant computations offers the advantages over procedural languages of modularity, conciseness, clarity, ease of proof, ease of introducing problem-oriented notation and independence from hardware peculiarities [15]. However, before discussing how functional programming relates to numerical methods, we introduce some of the features found in modern functional languages and the common classification of these languages according to their function invocation disciplines. Unless stated otherwise, all examples are given in Haskell-style syntax [49, 50].

2.2 What is Functional Programming?

Functional programs are so called because at the highest level a program is considered as a function from the program input to the program output. Pure functional programs do not possess an implicit state and so preserve referential transparency. All values are static, and re-assignment is disallowed. Referential transparency is the feature of programs where *equals may replace equals* such that in the expression

$$\ldots x + x \ldots \textit{where } x = f(a)$$

the function application $f(a)$ may be substituted for any free occurrence[1] of x in the scope created by the *where* expression (such as the sub-expression $x + x$). Referential transparency implies that there is no implicit underlying context to an expression and subexpressions may be evaluated in any order. Referential transparency is important in the field of mathematical programming, it encourages equational reasoning and modular methods of proof can be utilised whereby statements about whole constructs may be proved by proving sub-theorems (or lemmas) about their constituents.

[1] free occurrence: See Appendix C

2.3 Features of Modern Functional Languages

One of the features common to many functional programming languages is strong static typing. Although strong static typing is not restricted to functional languages it has become one of the features that tends to characterise them. Each expression in a program has a type associated with it and hence has a type expression (or signature) that describes it. For example if we consider Peano's successor function [57]:

```
>          succ :: Int -> Int
```

it can be seen from the signature that the function `succ` is of type `Int -> Int` (i.e. `succ` maps one integer to another). This is very similar to the mathematical notation of a function signature. We could define *succ* as

$$succ \text{ is a function from } \mathbb{N} \text{ to } \mathbb{N} \text{ such that } succ(x) = x + 1$$

and define `succ` as

```
>          succ :: Int -> Int
>          succ x = x+1
```

Although we do not need the type definition (as the language can often infer this for us) it is useful as it improves the readability of programs and often brings sources of error to light. The advantages of statically-typed languages are well known, all type errors are detectable at compile time and a compiler is able to produce efficient code as no runtime tags are needed.

2.3.1 User-Defined Datatypes and Pattern Matching

User-defined datatypes are defined using a notation resembling BNF [5]. For instance the type `Boolean` could be written

```
>          data Boolean = TRUE | FALSE
```

and a list of elements of type `a` as[i]

```
>          data List a = Empty | Cons a (List a)
```

One of the techniques that is used extensively in functional programming is pattern matching: the writing of many different equations to define a function where in each case only one is applicable.

```
>          length :: List a      -> Int
>          length    Empty       = 0
>          length    (Cons a b)  = 1 + (length b)
```

Pattern matching is important because it supports clear, concise definitions, structural induction, and encourages equational reasoning about programs.

2.3.2 Partial Application and Sections

Partially applicable, *or Curried*, functions are arity n functions which, when applied to a single argument, return functions of arity $(n-1)$. For example addition can be thought of as a function

$$(+)_{tuple} :: (\alpha, \alpha) \rightarrow \alpha$$

which sums pairs of numbers, or as a function

$$(+)_{Curried} :: \alpha \rightarrow (\alpha \rightarrow \alpha)$$

which takes one parameter and yields a function which increments its argument by the value of that parameter. The function-space constructor $(\rightarrow)$ is assumed to be right associative so we may omit the parentheses and write

$$(+)_{Curried} :: \alpha \rightarrow \alpha \rightarrow \alpha$$

A Curried function is applied to an argument by juxtaposing the function and the argument. For example if `add` is a Curried addition function then we could define the value `three` as the sum of 1 and 2 via

```
>          three :: Int
>          three =  (add 1) 2
```

but since function application is taken to be left associative we may again omit the parentheses and write

```
>          three :: Int
>          three =  add 1 2
```

An application of Currying is differentiation:

```
>          differentiate :: Function -> EvaluationPoint -> Value
```

since `differentiate f` constructs the function f' and `differentiate f k` constructs the value $f'(k)$.

A feature closely associated with Currying is sectioning. A section is a mechanism that forms prefix functions from infix operators allowing such operators to be partially applied to their left or right arguments:

```
>          (+)  <=> f where f a b = a + b
>          (a+) <=> g where   g b = a + b
>          (+b) <=> h where   h a = a + b
```

where free occurrences of `a,b` have bindings in the surrounding environment. The use of partial application is important as it cuts down on unnecessary brackets and also increases the usefulness of higher order functions.

2.3.3 Higher Order Functions

The existence of higher order functions stems from the argument that functions are values just like any other and, as such, should be given first class status, i.e. allowed to be stored in data structures, passed as arguments and returned as results. Since functions are the main abstraction mechanism over values, allowing functions to be higher order increases this form of abstraction. Another equally valid argument for higher order functions is that recurring patterns of computation can be identified and parameterised to allow complicated functions to be expressed simply using function composition. For example a function to compute the L_2 norm of a vector $\mathbf{x}$ having elements $x_1, x_2, \ldots, x_N$

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{N} x_i^2}$$

may be written as

```
>          l2_norm  :: [Double] -> Double
>          l2_norm  =  sqrt.sum.(map (^2))
```

where `f.g` denotes function composition $f \circ g$ and `map` applies a given function f to each element in a list, so `map (+1) [1,2,3] = [2,3,4]`. The function `map` is defined using pattern matching as

```
>          map :: (a -> b) -> [a]    -> [b]
>          map    f           []     = []
>          map    f           (x:xs) = f x:(map f xs)
```

and exhibits parametric polymorphism since it acts uniformly over lists of various component types. Higher order functions are essential for mathematical programming since they allow modular specifications of functionals such as integration.

2.3.4 List Comprehensions

In functional programming there is a very common notation called the *list comprehension*. This is similar to notation in set theory. For instance we can write

$$\mathit{filterSet}\ P\ Y = \{y | (y \in Y) \wedge P(y)\}$$

to define the subset of Y satisfying some predicate P. In functional programming we have the similar notation

```
>          filter :: (a -> Bool) -> [a] -> [a]
>          filter    p              ys   = [y| y<-ys,p y]
```

which may be read as 'The list xs of elements drawn from the list ys such that for each x in xs, x satisfies the predicate p'. We may use this notation to define *map*

```
>          map :: (a -> b) -> [a] -> [b]
>          map    f           xs  =  [f x | x<-xs]
```

which may be read as 'The list defined by applying the function f to each x where x is drawn from the list xs'. A list comprehension preserves order and can be considered as the functional-programming equivalent of a for loop.

Although list comprehensions could be viewed as mere notational convenience or *syntactic sugar*, the ability to express algorithms concisely and clearly with them makes them very useful. For example 'Quicksort' could be defined [91] as

```
>     qSort [Int] -> [Int]
>     qSort []    =  []
>     qSort (a:x) =  qSort [y|y<-x,y<=a]
>                       ++ [a] ++
>                    qSort [y|y<-x,y>a]
```

2.3.5 Type classes and Ad-Hoc Polymorphism

Recently, the notion of type classes[97] has appeared in functional languages. Type classes provide a system where identifiers may be incrementally overloaded. Consider the function `min` defined as

```
>         min x y = if x < y
>                      then   x
>                      else   y
```

The most general type of `min` is

$$\text{min} :: \alpha \rightarrow \alpha \rightarrow \alpha$$

for all types α with the operator $<$ defined on them. In Haskell this is written

```
>           min::Ord(a) => a -> a -> a
```

where `Ord` is the class of types admitting the operator $<$. Type classes capture the behaviour of a collection of overloaded functions in a consistent manner. To overload (+) on both integers and floats we declare the class admitting the operator (+)

```
>           class Num a where
>            (+) :: a -> a -> a
```

and declare integers and floats as instances[2] of this class

```
>           instance Num Int where    x + y = addInts x y
>           instance Num Float where  (+)   = addFloats
```

where `addInts` and `addFloats` are primitive addition functions specialised to integers and floating point numbers. If we now define a function `sum` which sums the elements in a non-empty list

```
>          sum :: Num a => [a]    -> a
>          sum             [x]    =  x
>          sum             (x:xs) =  x + sum xs
```

[2]Note: the definition `x + y = f x y` is equivalent to `(+) = f`.

we may apply the function sum both to lists of integers and to lists of floats or indeed lists of any type which is a member of the class Num. In this way the domain of applicability of the function sum and all functions defined in terms of it may be extended at a later date via the inclusion of extra types in class Num.

2.4 Classifications of Functional Languages

Functional languages are often classified according to the method used to evaluate expressions:

Strict (call by value) languages evaluate the arguments of a function before attempting to evaluate the body of the function itself in a style analogous to applicative order reduction [21] in the lambda calculus (see Appendix C). Examples of such languages include SISAL [19] and SML [67].

Lenient (eager non strict) languages do not impose an *a priori* order of evaluation between the function body and its arguments. This means that values arising from functions applied to undefined arguments may themselves be defined. This is what is meant by a function being non-strict in an argument. The only condition on the ordering of the evaluation is that when an argument is needed, evaluation of the function body must be suspended until the required argument becomes available [93]. An example of a lenient language is Id [71].

Lazy (call by need) languages are non-strict languages in which arguments to function calls are evaluated once only, if at all, and only evaluate values which contribute to required results. This makes it possible to define and manipulate potentially infinite objects within this framework, allowing elegant solutions to certain types of problems (see Section 2.6). Examples of such languages include Haskell and Miranda.

Call by name evaluation also exists although this is extremely rare. Call by name is non-strict evaluation in which no sharing occurs. This is analogous to normal reduction in the lambda calculus (See Appendix C.).

To appreciate the difference between strict, non-strict and lenient semantics consider the function `f x y = x+x` which depends only on its first argument. In a non-strict language `f 1` $\perp$ evaluates to 2 whereas in a strict language `f 1` $\perp$ evaluates to $\perp$, where $\perp$ is the value assigned to an expression without a normal form, or to a non-terminating computation. In a lenient language `f 1` $\perp$ will evaluate to 2 as long as partial evaluation of $\perp$ does not explicitly halt the computation.

2.5 Advantages of Non-Strict Semantics

One advantage of non-strict languages is their ability to manipulate potentially infinite structures. Programs may be modularized further by decoupling the process of producing a structure from the process consuming it [52].

2.5.1 Infinite Structures

An example of an infinite structure is an infinite sequence of approximations to a value defined by an iteration. Taking square roots as an example, we can define the method of producing the next value in a Newton iteration by the function

```
>     newton :: Double -> Double -> Double
>     newton    a          x      =  ( x + a / x ) / 2
```

which, given a number a and an initial estimate x, produces a closer estimate of the square root of a. We may now use the higher-order function *iterate* which produces an infinite list of values such that *iterate* $f\ a = [a, f(a), f^2(a), \ldots]$, given the initial value a and function f:

```
>     iterate (a -> b) -> a -> [b]
>     iterate f           a =  a:iterate f (f a)
```

The function *within*, returning the approximate limit of a sequence deemed to have converged on a value to within *eps*, may be defined as

```
>     within :: Double -> [Double] -> Double
>     within eps (a:b:rest) | abs (a-b) < eps = b
>                           | otherwise = within eps (b:rest)
```

Finally the square root

```
>     sqrt :: Double -> Double -> Double -> Double
>     sqrt x eps a = within eps (iterate (newton a) x)
```

may be constructed from reusable components.

2.5.2 Recursive Structures

Another advantage of non-strict languages is the ability to consume structures before they are completely defined. This allows the recursive definition of data structures. For example, back substitution[ii] is often defined as

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}, \qquad i = 1, 2, \ldots, n$$

and may be expressed in Haskell similarly as

```
>    type Matrix = Array (Int,Int) Double
>    type Vector = Array Int Double

>    back_subst :: Matrix -> Vector -> Vector
>    back_subst u b = x
>     where x = array bnds [(b!i - f i)/u!(i,i)|i<- range bnds]
>           f i = sum[u!(i,j)*x!j|j <- range (i+1,n)]
>           bnds@(1,n) = bounds b
```

where the vector x is defined in terms of itself. This *recursive definition* is *not* allowed in a strict language as any value defined as $v = f(v)$ for some function f results in non-termination analogous to writing

```
int f(){return (!f());}
```

in the imperative language C[58].

Other forms of non-strictness occur in programs such as *functional* and *conditional* non-strictness[85] but in the context of this thesis these are of little practical benefit and, as such, will not be discussed further.

2.5.3 Simulating Non-Strict Behaviour in Strict Languages

As mentioned earlier, one of the advantages of lazy languages is the ability to manipulate potentially infinite structures. If written verbatim in a strict language such expressions would result in non-termination. It is straightforward, however, to simulate this behaviour in a strict language (SML), and a lazy list[(iii)] may be defined as follows:

```
datatype 'a lazyList = Nil
                     | Cons of 'a * (unit -> 'a lazyList);
```

Our square root function can be written in SML in a style similar to that used before:

```
fun newton a x = (x+a/x)/2.0;

fun iterate f a = Cons(a,fun()=>iterate f (f a));

fun within (eps:real) Cons(a1,as) =
     let val (Cons(a2,rest)) = as();
     in if
          (a1-a2) < eps
        then
          a2
        else
          within eps (Cons(a2,rest))
     end;

fun sqrt x eps a = within eps (iterate (newton a) x);
```

where we manually insert our own closures. In this way it can be seen that the modularity of the previous lazy program can be achieved within a strict framework. In the above example a list is represented by a value/function pair (a,as), which can be viewed as being analogous to a list element together with a *pointer* or link to the next value/function pair in an imperative language. Unfortunately, in general, this method may not be used to define recursive structures such as the vector x in a back substitution algorithm as the exements must be evaluated in a *head to tail* order.

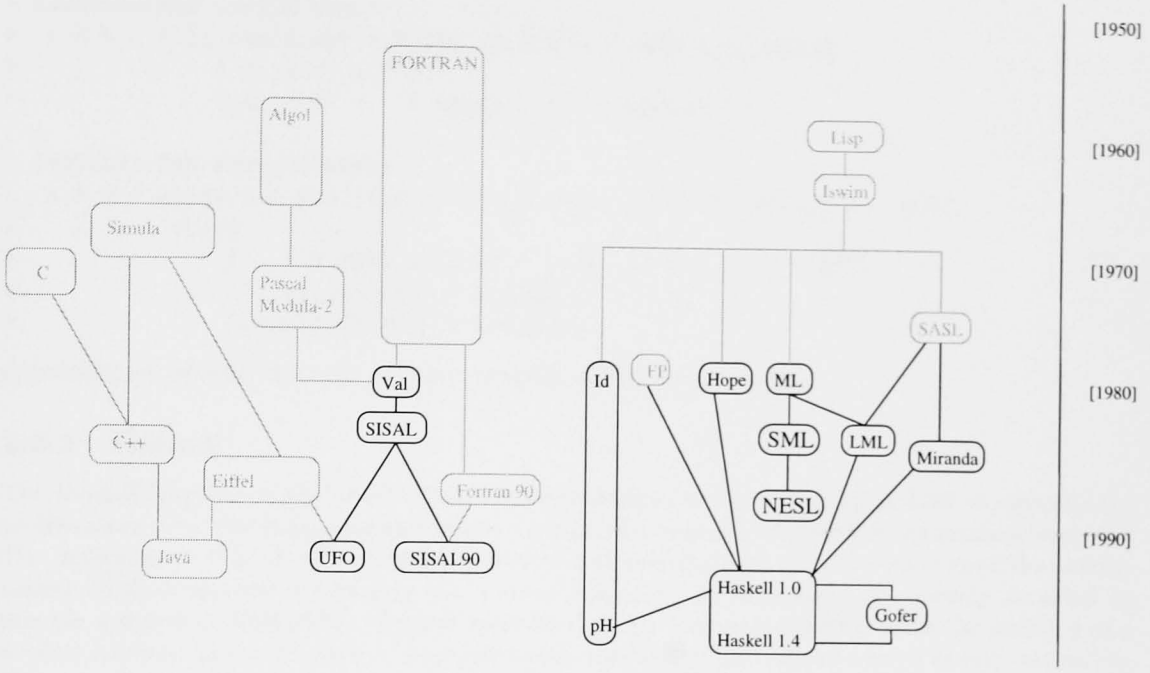


Figure 2.1: Functional language history and lineage

2.6 Functional Language Implementations

In this section common functional languages and implementations are introduced and briefly described. The lineage of these langauges is shown in Fig. 2.1, a diagram which will be extended later to include FSC.

2.6.1 GoFER

The GoFER (**Go**od **F**or **E**quational **R**easoning)[54] system provides an interpreter for a small language based closely on version 1.2 of the Haskell report [50]. In particular, GoFER supports lazy evaluation, higher-order functions, polymorphic typing, pattern-matching, and support for overloading. GoFER was the first language to provide constructor classes [54] which allow functions such as `map` to be overloaded to operate over container types such as lists and trees. The notable feature of GoFER, which distinguishes it from similar languages such as Haskell, is its multi-parameter type class mechanism which is discussed in greater depth later. Matrix multiplication in GoFER may be written as either of the following definitions[3]

[3] The GoFER definitions are not strictly permissible as synonyms are not allowed as class members. We ignore this fact as this permits a clearer presentation.

```
> instance Num List2D where
>  a * b   = [[ sum[a_ik* b_kj|(a_ik,b_kj) <- zip a_i_ b__j]
>             | b__j <- b_T ]
>             | a_i_ <- a   ] where b_T = transpose b

> instance Num Array2D where
>  a * b = array ((1,m1),(1,n2)) [(i,j):=f i j|i<-[1..m1],j<-[1..n2]]
>           where
>               f i j = sum[ a!(i,k) * b!(k ,j) | k <- [1..n1]]
>               ((1,m1),(1,n1)) = bounds a
>               ((1,m2),(1,n2)) = bounds b
```

depending on whether we view a matrix as a list of lists or as an array.

2.6.2 Haskell

The Haskell language is the result of the language-design committee set up in 1987 to alleviate the proliferation of similar languages and create a *standard* non-strict functional programming language. The definition of Haskell is frequently revised to add new features with the effect that the current version (1.4) of Haskell is certainly not a small language although there is currently an effort to provide a standard Haskell[51]. Haskell provides features similar to GoFER with the addition of a module system. From the view of expressiveness, readability and conciseness it is very attractive. However, the lack of efficiency with which it executes and the storage it currently requires are much more costly than with strict or imperative languages [50, 49]. Matrix multiplication in Haskell is exactly the same as for the GoFER example.

2.6.3 Hope

Hope [17] is a small polymorphically-typed functional language that can be considered as one of the forerunners to Haskell/SML. Hope was the first language to use pattern matching and was originally strict. To a large extent, Hope has fallen into disuse now but is worthy of mention as one of the features of Hope is its *best-fit* pattern matching whereby patterns need not be read in a top-to-bottom manner but *must* be collectively unambiguous. Matrix multiplication in Hope may be written in a very similar manner to the algorithm in GoFER.

2.6.4 Hugs

Hugs [55], the Haskell User's GoFER System, is an interpreted implementation of Haskell with an interactive development environment much like that of GoFER.

2.6.5 Id

The core of Id [71] is a non-strict functional language with implicit parallelism targeted at dataflow architectures such as the Monsoon[74]. Id supports polymorphic typing, algebraic types and definitions with clauses and pattern matching, and list comprehensions. Since the syntax of Id closely

resembles that of Haskell, Id researchers have recently switched their attention to transforming Id to pH (parallel Haskell) which will have the same semantics as Haskell (up to program termination). Id works on the basis of lenient execution (as mentioned earlier) but also allows non-functional side effects in the form of I-structures and M-structures. I-structures are structures that may be defined only once and all attempts to read an undefined element will block until it is defined. M-structures are completely mutable and allow non-determinism, the only provisos being that a process cannot write to a full cell, or read from an empty cell, and the act of reading empties a cell. Matrix multiplication in Id may be written

```
def mmult A B = {(loa1,m1),(loa2,n1) = matrix-bounds A
                 (lob1,m2),(lob2,n2) = matrix-bounds B
                  in
                  {matrix ((loa1,m1),(lob2,n2))
                  | [i,j] = sum {: A[i,k]*B[k,j] || k <- lob1 to m2}
                  || i <- loa1 to m1 & j <- lob2 to n2};
```

2.6.6 Miranda

Miranda[4] [96] is a strongly-typed, higher-order lazy functional language designed in 1985 as the successor to SASL [94] and KRC [95].

Miranda was the first widely disseminated language with non-strict semantics and polymorphic strong typing, and was one of the main influences on the later design of Haskell. Points differing from Haskell include the fact that overloading is performed over numeric types by having a single type num, and pattern matching in equations need not be linear[5]. Matrix multiplication in Miranda may be expressed in a form almost identical to that of GoFER, differing only in that Miranda does not support overloading.

2.6.7 ML

ML[6] [67] is a family of programming languages with (usually) functional control structures, strict semantics, a strict polymorphic type system, and parameterised modules [30]. The family includes Standard ML, Lazy ML, CAML, CAML Light, and various experimental languages. Matrix multiplication may be expressed as [75]

```
fun dotprod([],[]) = 0.0
  | dotprod(x::xs,y::ys) = x*y + dotprod(xs,ys);

fun rowprod(row,[]) = []
  | rowprod(row,col::cols) =
              dotprod(row,col) :: rowprod(row,cols);
```

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

[5] A non-linear pattern is a pattern which implies an equality between arguments. The function `equal x x = True` is non-linear

[6] Meta-Language.

```
fun rowlistprod([],cols) = []
  | rowlistprod(row::rows,cols) =
        rowprod(row,cols) :: rowlistprod(rows,cols);

fun matprod(A,B) = rowlistprod(A, transp B)
```

where transp transposes a matrix stored as a list of lists.

2.6.8 NESL

NESL [10] is a fine-grained, mostly-functional, nested data-parallel language based loosely on ML with implementations for workstations, the Connection Machines CM2 and CM5, the Cray Y-MP and the MasPar MP2. NESL includes a built-in parallel datatype of polymorphic sequences, strict semantics, polymorphic typing and a limited use of higher-order functions. Currently it does not have support for modules and its datatype definition is limited. The compiler is based around delayed compilation and specialised polymorphic functions, avoiding the need for uniform representation/calling convention. Matrix multiplication in NESL may be written as

```
function matrix_multiply(A,B) =
  {{sum({x*y: x in rowA; y in columnB})
    : columnB in transpose(B)}
   : rowA in A} $
```

2.6.9 SISAL

SISAL[7] [19] is a first order applicative language designed "to support clear, efficient expression of scientific programs; to free application programmers from details irrelevant to their endeavours; and to allow automatic detection and exploitation of the parallelism expressed in source programs" [30].

The SISAL language is currently implemented on several shared memory and vector systems running Unix, including the Sequent Balance and Symmetry, the Cray X/MP and Y/MP, Cray 2, and a few other less well-known machines [19, 18]. The newer SISAL90, which is still under development allows user-defined reductions, limited polymorphism via typesets, higher-order functions and array operations resembling the vector operations of FORTRAN90. SISAL90 still lacks support for datatypes, type inference, an I/O system, and many of the characteristic features of modern functional languages [31, 86]. SISAL has been shown to outperform handwritten FORTRAN and is currently benchmarked as the most efficient functional language implementation [45]. Matrix multiplication in SISAL may be written as

```
type OneDim = array[ real ];
type TwoDim = array[ OneDim ];

function Matmult( A,B:TwoDim; M,N,Linteger returns TwoDim)
```

[7] Streams and Iteration in a Single Assignment Language

```
  for i in 1, M cross j in 1, L
    S := for k in 1, N
         returns value of sum A[i,k] *B[k,j]
         end for
  returns array of S
  end for
end function
```

2.6.10 UFO (United Functions & Objects)

UFO [83, 82] is an attempt to reconcile the functional and object-oriented worlds by providing a language whose core is functional (in the style of SISAL) but which also resembles object-oriented languages such as Eiffel [66]. The motivation behind UFO is the desire to provide a general purpose parallel language for both numerical and symbolic computations which may require state. Matrix multiplication in UFO may be written

```
typevar Num <- Numeric

matmult(a, b : Array[Num]) : Array[Num] is
{
 for       i in [lower(a,1) to upper(a,1)]
     cross j in [lower(b,2) to upper(b,2)] do
  cij =
    for a_element in a[i,.. dot b_element in b[..,j] do
     c_element = a_element * b_element
    return sum(c_element)
    od
  return cij
  od
}
```

This concludes our introduction to functional programming and description of various functional programming languages. The review is not exhaustive but offers an overview of the functional language spectrum. The interested reader is referred to [48] for a more complete but less recent discussion.

2.7 Chapter Notes

i (page 8):
Although

```
>     data List a = Empty | Cons a (List a)
```

is a perfectly valid definition of a list it is more common to write `Empty` as `[]`, `Cons` as `(:)`, `List a` as `[a]` and to think of lists as being defined as

```
>     data [a] = [] | a:[a]
```

Although this is not valid Haskell syntax.

ii (page 13):
Back substitution is the solution of the linear system in n unknowns

$$\begin{array}{lllll} u_{11}x_1+ & u_{12}x_2 & +\cdots+ & u_{1n}x_n & = b_1 \\ & u_{22}x_2 & +\cdots+ & u_{2n}x_n & = b_2 \\ & & \ddots & \vdots & = \vdots \\ & & & u_{nn}x_n & = b_n \end{array}$$

usually written as $\mathbf{Ux} = \mathbf{b}$ where $\mathbf{U}$ is an upper triangular matrix. The back substitution method works by solving the final equation for x_n, substituting this value into the first $n-1$ equations, and so reducing the original system to a system with $(n-1)$ unknowns

$$\begin{array}{lllll} u_{11}x_1+ & u_{12}x_2 & +\cdots+ & u_{1,n-1}x_{n-1} & = b_1 - u_{1n}x_n \\ & u_{22}x_2 & +\cdots+ & u_{2,n-1}x_{n-1} & = b_2 - u_{2n}x_n \\ & & \ddots & = & \vdots \\ & & & u_{n-1,n-1}x_{n-1} & = b_{n-1} - u_{n-1,n}x_n \end{array}$$

The back substitution algorithm is then applied to this reduced system.

iii (page 14):
In Section 2.6.3 we referred to the datatype `lazylist` as a *lazy* list. This is actually a misnomer as no sharing occurs and repeatedly used tails are recomputed since they are explicitly represented as closures. A more accurate description would be a *call by name* list. The term *lazy list* is used since this is its title in [75] where the example originates.

Chapter 3

Numerical Methods

In this chapter we present a background to the numerical methods considered later in this thesis. For a simple background to numerical analysis see [39]. For readers unfamiliar with the notation used, Mathematical notation is explained in Appendix K.

3.1 Introduction

Numerical methods are techniques employed by engineers and scientists to solve mathematical equations. A major advantage of numerical methods over analytical methods is that a numerical answer can often be obtained even when a problem has no *analytical* solution. For example, the following integral taken from [39], which gives the length of one arc of the curve of $y = \sin(x)$, has no closed form, analytic, solution:

$$\int_0^{\pi} \sqrt{1 + \cos^2(x)}dx.$$

However, the length of this curve may be computed by standard numerical methods that apply to essentially any integrand; there is never a need to make special substitutions or to apply Greens theorem[1] to obtain a result. Moreover, the only operations required are addition, subtraction, multiplication, division and comparisons. It is this fact that makes a computer so suited to the task of *number crunching.*

Important distinctions between analytical and numerical solutions to problems are:

- A numerical solution is an approximation to the true solution which, in theory, may be made arbitrarily accurate (up to machine arithmetic).
- The behaviour and properties of numerical solutions are not as apparent as they often are in analytical solutions, precisely because they are purely numerical. Numerical methods frequently simply yield tables of values which must be analysed further, using visualisation techniques, before such properties may be determined.

[1] Greens theorem: A generalisation of integration by parts

In the remainder of this thesis we consider systems of equations, approximation of functions, numerical differentiation, numerical integration and quadrature, root finding / optimisation, differential equations, and symbolic manipulation, and so here very briefly introduce each of these areas. These topics are introduced because

- They are typical scientific computations.
- We later present the limitations of certain styles of programming in terms of these examples.
- Quantitative statistics for codes are given in terms for these topics.
- They are particularly good at demonstrating the suitability of later solutions.

Frequently used notation is given in Appendix K.

3.2 Systems of Equations

The simplest systems of equations are linear algebraic equations (LAEs). A set of n LAEs in n unknowns has the form:

$$\begin{array}{ccccccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1j}x_j & + & \cdots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2j}x_j & + & \cdots & + & a_{2n}x_n & = & b_2 \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
a_{i1}x_1 & + & a_{i2}x_2 & + & \cdots & + & a_{ij}x_j & + & \cdots & + & a_{in}x_n & = & b_i \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nj}x_j & + & \cdots & + & a_{nn}x_n & = & b_n
\end{array}$$

where the x_i are the unknowns and the a_{ij} and b_i are known. For convenience this is written in matrix notation as

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where $\mathbf{A}$ is the coefficient matrix

$$\mathbf{A} = [a_{ij}] = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

and $\mathbf{x}$, $\mathbf{b}$ are vectors

$$\mathbf{x} = [x_i] = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad \mathbf{b} = [b_i] = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}.$$

We assume that the matrix $\mathbf{A}$ is non-singular, that is $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a unique solution.

3.2.1 Direct Methods

One of the simplest methods for solving the linear system $\mathbf{Ax} = \mathbf{b}$ is *Gaussian Elimination*. This involves the systematic subtraction of multiples of one equation from others to obtain an upper triangular system

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}$$

where the u_{ij} and the β_i are linear combinations of the a_{ij} and the b_i respectively such that this reduced system is equivalent to the original. Since the last equation has only one unknown it can be solved for x_n. This value can then be substituted into the penultimate equation to determine the value of x_{n-1}. This *back-substitution* process continues until the values of all the x_i have been determined.

This is an example of a direct method since the solution is obtained in a *fixed* finite number of operations dependent only on n. One of the major drawbacks of the method is that the right-hand side vector, $\mathbf{b}$ must be altered along with $\mathbf{A}$ causing difficulties if we have multiple right-hand sides which are not available simultaneously[2]. This problem may be overcome via the use of LU decomposition.

3.2.2 LU Decomposition

Since triangular systems of equations may be solved easily using back substitution we may decompose a linear system into the product $\mathbf{LU} = \mathbf{A}$ where $\mathbf{L}$ is an $n \times n$ lower triangular matrix and $\mathbf{U}$ is an $n \times n$ upper triangular matrix. Since there is more than one possible factorisation we often take the diagonal elements of $\mathbf{L}$ to be 1 ($l_{ii} = 1$) and by omitting this diagonal store the factorisation in the same amount of space required to store $\mathbf{A}$.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{bmatrix}$$

We may now use these matrices to solve $\mathbf{Ax} = \mathbf{b}$ or $(\mathbf{LU})\mathbf{x} = \mathbf{b}$ by solving the two sets of linear equations

$$\mathbf{Ly} = \mathbf{b} \qquad \mathbf{Ux} = \mathbf{y}$$

We may solve $\mathbf{Ly} = \mathbf{b}$ using forward substitution and then solve $\mathbf{Ux} = \mathbf{y}$ using back substitution and solve the system without altering $\mathbf{b}$. In the method the u_{ij} are the original u_{ij} from Gaussian elimination and the l_{ij} are the multipliers computed at each stage.

[2] This situation arises in the case of iterative refinement.

3.2.3 Stability

Before leaving direct methods we discuss *pivoting*. In both LU factorisation and Gaussian elimination we observe that if at the k^{th} stage[3] the element a_{kk} is zero then the method breaks down, possibly because the matrix is singular but not necessarily so (as this condition may arise due to a row-wise permutation $\mathbf{P}$ of $\mathbf{A}$). For example if $\mathbf{A} = \mathbf{I}_{33}$

$$\mathbf{P}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\mathbf{x} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\mathbf{x} = \mathbf{Pb}$$

The row-wise permutation shown above would yield a system which although non-singular would cause Gaussian elimination to fail. However, as long as at least one of the sub-diagonal elements in a_{kk}'s column (*the pivotal column*) is non-zero at the k^{th} stage of elimination, the process may continue by interchanging rows. Even if this element is not identically zero, it may be so small compared to other elements in that column that the multipliers which are generated are very large, resulting in the whole process becoming unstable and highly inaccurate due to roundoff-errors being amplified.

This row-exchange process is usually performed by searching down the pivotal column for the element with largest magnitude, exchanging row k with its row and continuing as before. This is known as *partial pivoting*. An alternative is *complete pivoting* in which the whole lower submatrix is searched for the element of greatest magnitude, causing row and column exchanges to be carried out, and therefore additional book-keeping. Since partial pivoting is usually deemed sufficient, subsequent references to pivoting will refer to partial pivoting unless otherwise qualified.

Other direct methods such as Choleski decomposition, cyclic reduction or QR factorisation work in a similar manner to Gaussian elimination and LU factorisation in that they transform the system into an equivalent one which may be solved trivially. An interesting direct (but often used as an indirect) method which is not of this form is the conjugate-gradient method which is discussed later.

3.2.4 Cyclic reduction

Cyclic reduction [36] (or odd-even reduction) is a method of solving a tridiagonal system of equations which may be performed in parallel. We focus on this method as we shall use it as an extended example later. The variation of cyclic reduction we choose to implement proceeds as follows:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{bmatrix}\begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ \vdots \\ \vdots \\ y_n \end{bmatrix}$$

[3] We define the k^{th} stage as the stage in the computation which determines the value of u_{kk}.

A tridiagonal system of equations $Ax = y$ is transformed into an equivalent pentadiagonal system with zero sub- and super-diagonals.

$$\begin{bmatrix} b_1^{(1)} & 0 & c_1^{(1)} & & \\ 0 & b_2^{(1)} & \ddots & \ddots & \\ a_3^{(1)} & \ddots & \ddots & \ddots & c_{(n-2)}^{(1)} \\ & \ddots & \ddots & \ddots & 0 \\ & & a_n^{(1)} & 0 & b_n^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1^{(1)} \\ \vdots \\ \vdots \\ \vdots \\ y_n^{(1)} \end{bmatrix}$$

This system is then reordered so that it is expressed as two sub-problems of half the dimension. That is, if we assume that n is even we renumber the equations in the order $1, 3, \ldots, n-1, 2, 4, \ldots, n$ and renumber the unknowns in a similar way.

$$\left[\begin{array}{ccc|ccc} b_1^{(1)} & c_1^{(1)} & & & & \\ a_3^{(1)} & \ddots & \ddots & & & \\ & \ddots & b_{n-1}^{(1)} & & & \\ \hline & & & b_2^{(1)} & \ddots & \\ & & & \ddots & \ddots & c_{n-2}^{(1)} \\ & & & & a_n^{(l)} & b_n^{(1)} \end{array}\right] \left[\begin{array}{c} x_1 \\ \vdots \\ x_{n-1} \\ \hline x_2 \\ \vdots \\ x_n \end{array}\right] = \left[\begin{array}{c} y_1^{(1)} \\ \vdots \\ y_{n-1}^{(1)} \\ \hline y_2^{(1)} \\ \vdots \\ y_n^{(1)} \end{array}\right]$$

Since we now have two tridiagonal systems we may apply recursively and in parallel the above transformation process to the two non-zero quadrants of the matrix until we reach a set of trivial equations. The transformation step to create the pentadiagonal system described above is given below

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1} \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \beta_i a_{i+1} \\ c_i^{(1)} &= \beta_i c_{i+1} \\ y_i^{(1)} &= y_i + \alpha_i y_{i-1} + \beta_i y_{i+1} \\ \alpha_i &= -a_i / b_{i-1} \\ \beta_i &= -c_i / b_{i+1} \end{aligned}$$

After solving each final equation (in parallel) the vector of solutions may be reordered to yield the solution vector.

3.2.5 Iterative Methods

In the last section we considered the solution of linear systems in a fixed, finite number of steps. We now consider an iterative approach where we make an initial guess $\mathbf{x}^{(0)}$ at the solution and use this to form a better approximation $\mathbf{x}^{(1)}$, which in turn we use to form $\mathbf{x}^{(2)}$, etc. The aim is that the sequence of vectors $\{\mathbf{x}^{(k)}\}, k = 1, 2, \ldots$ converges to the exact solution $\mathbf{x}$ of the system, i.e. $\mathbf{x}^{(k)} \rightarrow \mathbf{A}^{-1}\mathbf{b}$ as $k \rightarrow \infty$.

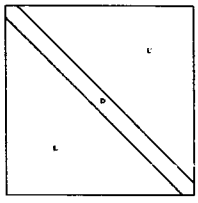


Figure 3.1: Partitioning of a matrix for the Jacobi/Gauss-Seidel methods

3.2.6 Gauss-Seidel and Jacobi Methods

If $\mathbf{A}$ is an $n \times n$ non-singular matrix and $\mathbf{Ax} = \mathbf{b}$ is the linear system to be solved, then the Jacobi iteration is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[-\sum_{j \neq i} a_{ij} x_j^{(k)} + b_i \right], \qquad i, j = 1, 2, \ldots, n$$

This may be written in matrix form by splitting $\mathbf{A}$ into three sections, the diagonal $\mathbf{D}$ and the sub/superdiagonal elements $\mathbf{L}/\mathbf{U}$ (Fig. 3.1). Our iteration may now be rewritten as

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}[-(\mathbf{L} + \mathbf{U})]\mathbf{x}^{(k)} + \mathbf{D}^{-1}\mathbf{b}.$$

Convergence to a fixed point[4] $\mathbf{x}$ satisfies

$$\mathbf{x} = \mathbf{D}^{-1}[-(\mathbf{L} + \mathbf{U})]\mathbf{x} + \mathbf{D}^{-1}\mathbf{b}$$

that is

$$\mathbf{Dx} = -(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b}$$

or

$$(\mathbf{D} + \mathbf{L} + \mathbf{U})\mathbf{x} = \mathbf{b}$$

or

$$\mathbf{Ax} = \mathbf{b}$$

and hence would solve our system of equations. With iterative methods, convergence to a fixed point is not guaranteed although, depending on the method used, there are conditions for which convergence *is* guaranteed. For example Jacobi's method converges if $\mathbf{A}$ is diagonally dominant $(\forall i, |a_{ii}| > \sum_{j \neq i} |a_{ij}|)$.

A method related to Jacobi's method is the Gauss-Seidel iteration

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[-\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + b_i \right], \qquad i = 1, 2, \ldots, n$$

[4] A fixed point $\mathbf{x}$ in the sequence $\mathbf{x}^{(0)}, T\mathbf{x}^{(0)}, T^2\mathbf{x}^{(0)}, T^3\mathbf{x}^{(0)}, \ldots$ satisfies $\mathbf{x} = T\mathbf{x}$.

or in matrix form

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\left[-\mathbf{L}\mathbf{x}^{(k+1)} - \mathbf{U}\mathbf{x}^{(k)} + \mathbf{b}\right]$$

which converges to a solution more rapidly. With iterative methods the degree of accuracy of the solution may be defined by specifying the value of the residual $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ at which the iteration is deemed to be a close enough approximation to the true solution, or by examining the difference between successive estimates. This is usually done by determining whether a particular norm is sufficiently small.

3.2.7 Non-Linear Equations

So far we have only considered systems of linear equations. However, systems of non-linear equations frequently arise such as

$$\begin{aligned} \cos(2x_1) - \cos(2x_2) &= 0.4 \\ 2(x_2 - x_1) + \sin(2x_2) - \sin(2x_1) &= 1.2 \end{aligned}$$

Generally we write these systems as

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

or more compactly as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

where $\mathbf{f}$ is the vector of functions $[f_1, f_2, \ldots, f_n]^T$ and $\mathbf{x}$ the vector of variables $[x_1, x_2, \ldots, x_n]^T$, and use root-finding or optimisation techniques to solve these systems iteratively. Such iterative methods may themselves generate and solve a linear system at each iteration. Further discussion is left to Section 3.6.

3.3 Approximation of Functions

The computational procedure used in computer software for the evaluation of a library function such as $\sin(x)$, $\cos(x)$, or e^x involves polynomial approximation. The simplest method of polynomial approximation is a truncated Taylor Series where, if we assume that f is continuously $(N+1)$-times differentiable over the interval $[a, b]$, then

$$f(x) = P_N(x) + E_N(x)$$

where $P_N(x)$ is an N^{th}-degree polynomial approximation to $f(x)$

$$f(x) \simeq P_N(x) = \sum_{k=0}^{N} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

x_0 is an arbitrary point within the inteval and $E_N(x)$ is the error term

$$E_N(x) = \frac{f^{(N+1)}(c)}{(N+1)!}(x - x_0)^{(N+1)}$$

for some c lying between x_0 and x. Hence, the function $\sin(x)$ may be approximated using the Taylor series expansion around the point ($x_0 = 0$)

$$\sin(x) \simeq x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Before continuing we briefly note that this approximation may be reduced by *economising* the power series. Economisation perturbs the coefficients in a power series such that much of the accuracy of an N^{th}-degree polynomial approximation is captured as an $(N-1)^{th}$-degree approximation, usually via the subtraction of a weighted Chebyshev polynomial (for further discussion see [64]).

3.3.1 Evaluation

The method used to evaluate a polynomial is crucial if execution efficiency is to be maintained. One method of evaluating a polynomial is *Horner's Method* where the polynomial

$$g(x) = x^N + a_{N-1}x^{N-1} + \cdots + a_1 x + a_0$$

is rewritten

$$g(x) = (((\cdots(((x + a_{N-1})x + a_{N-2})x + \cdots + a_1)x + a_0$$

requiring $(N-1)$ multiplications and N additions. Similar methods may be used to evaluate a Taylor series expansions

$$f(x) \simeq \sum_{k=0}^{N} \frac{f^k(x_0)}{k!}(x - x_0)^k = \sum_{k=0}^{N} f^k(x_0)P_k$$

where the P_i are defined as

$$\begin{aligned} P_0 &= 1 \\ P_1 &= x - x_0 \\ P_k &= \tfrac{1}{k}P_1 P_{k-1} \end{aligned}$$

and are evaluated P_1 to P_N with common terms being shared.

3.3.2 Fourier Series

Polynomials are not the only functions which may be used to approximate known functions. A *Fourier Series* approximation may be formed via a sum of sine and cosine terms:

$$f(x) \simeq \frac{A_0}{2} + \sum_{n=1}^{\infty}[A_n \cos(nx) + B_n \sin(nx)]$$

where A_n and B_n are defined as

$$A_n = \frac{1}{P}\int_{-P}^{P} f(x)\cos\left(\frac{n\pi x}{P}\right)dx, \qquad B_n = \frac{1}{P}\int_{-P}^{P} f(x)\sin\left(\frac{n\pi x}{P}\right)dx$$

creating an approximation to $f(x)$ with a period of $2P$. The act of computing the coefficients in a Fourier series is sometimes called *harmonic analysis* since if f is a time-dependent periodic function, its Fourier series represents an equivalent function of frequencies, and knowledge of the most significant coefficients in a Fourier series provides information on the fundamental frequencies of a system. This knowledge is important if phenomena such as resonance are to be avoided. Another name for this procedure is *the Fourier transform.*

3.4 Numerical Differentiation

Numerical differentiation involves the estimation of derivatives at certain specific points over the domain of a function f. To see how $\frac{df}{dx}$ may be estimated we expand $f(x)$ in a Taylor series about $(x = x_n + h)$

$$f(x_n + h) = f_{n+1} = f_n + hf_n{}' + \frac{h^2}{2!}f_n{}'' + \cdots$$

which, when truncated at the 2^{nd} term, gives:

$$hf_n' \simeq f_{n+1} - f_n = \Delta f_n \Rightarrow \boxed{f_n' \simeq \tfrac{\Delta f_n}{h}}$$

and the first derivative may be approximated by the quotient of the forward difference and step length. Similarly, expanding around $(x = x_n - h)$

$$f(x_n - h) \equiv f_{n-1} = f_n - hf_n' + \frac{h^2}{2!}f_n'' - \cdots$$

gives $\boxed{f_n' \simeq \tfrac{\nabla f_n}{h}}$. These approximations may be combined to form a more accurate approximation:

$$f_{n+1} - f_{n-1} \simeq f_n - f_n + hf_n' - (-hf_n') + \frac{h^2}{2!}f_n'' - \frac{h^2}{2!}f_n'' + \cdots \Rightarrow \boxed{f_n' \simeq \tfrac{f_{n+1} - f_{n-1}}{2h}}$$

since it involves truncation at the third term rather than the second. This is known as the *central difference* approximation to the first derivative. Adding truncated Taylor series expanded around $(x = x_n + h)$ and $(x = x_n - h)$ leads to an approximation for the second derivative $\boxed{f_n'' \simeq \tfrac{f_{n-1} - 2f_n + f_{n+1}}{h^2}}$. Further discussion of differentiation and differences is left until later sections where differential equations are introduced.

3.5 Numerical Integration and Quadrature

Just as numerical differentiation estimates values of a function's derivative evaluated at a specific point, numerical integration estimates the value of an integral over a specific interval $[a, b]$. The

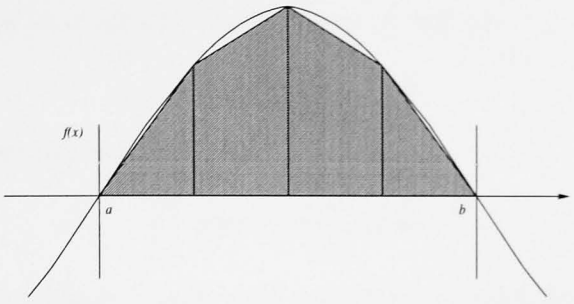


Figure 3.2: The trapezoidal rule

simplest approach to numerical integration is to sum the area under a piecewise-linear approximation to the curve as in Fig.3.2. This is called the *trapezoidal rule* and may be formulated as:

$$\int_{x_0}^{x_N} f(x)dx \simeq \frac{x_N - x_0}{2N} \sum_{i=0}^{N-1} (f_i + f_{i+1})$$

where $(x_{i+1} - x_i)$ is constant. By estimating the data using a piecewise-quadratic or a piecewise-cubic curve we obtain *Simpson's* $\frac{1}{3}$ *and* $\frac{3}{8}$ *Rules*:

$$\int_{x_0}^{x_N} f(x)dx \simeq \frac{x_N - x_0}{3N} \sum_{i=0}^{N/2-1} (f_{2i} + 4f_{2i+1} + f_{2i+2})$$

$$\int_{x_0}^{x_N} f(x)dx \simeq \frac{3(x_N - x_0)}{8N} \sum_{i=0}^{N/3-1} (f_{3i} + 3f_{3i+1} + 3f_{3i+2} + f_{3i+3})$$

These methods are all examples of *Newton-Cotes* integration formulae and, in general, formulae may be derived to approximate an integral using a piecewise-n^{th} degree polynomial, although higher order methods are not much use in general. For further information see [39].

3.5.1 Gaussian Quadrature

The previous formulae for numerical integration all require that the value of a function be evaluated at evenly-spaced $x-$values. An alternative approach is to use a method known as *Gaussian Quadrature*. Gaussian Quadrature symmetrically places N x-values around the interval midpoint and approximates the definite integral

$$I = \int_a^b f(x)dx$$

by transforming f into a function ϕ:

$$\phi(x) = f[\tfrac{1}{2}(b-a)x + \tfrac{1}{2}(b+a)]$$

allowing I to be expressed as an integral between limits -1 and $+1$:

$$I = \tfrac{1}{2}(b-a)\int_{-1}^{1} o(x)dx$$

This integral may be approximated as follows:

$$\int_{-1}^{1} \phi(x)dx = \sum_{i=1}^{N} w_i o(x_i)$$

where w_i and x_i represent tabulated values of the *weight functions* and the *abscissae* (or *integration points*) associated with the N points in the interval $(-1, 1)$. Thus the result is

$$I = \int_a^b f(x)dx \simeq \tfrac{1}{2}(b-a)\sum_{i=1}^{N} w_i\phi(x_i)$$

which, for appropriate weights and abscissae (which may be calculated), will exactly integrate a polynomial of degree $(2N-1)$ [1].

3.6 Root Finding

As mentioned earlier, root-finding techniques are often used to find a solution to a system of non-linear equations

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

often written

$$\mathbf{f(x) = 0}$$

where $\mathbf{f}$ is the vector of functions $[f_1, f_2, \ldots, f_n]^T$ and $\mathbf{x}$ is the vector of independent variables $[x_1, x_2, \ldots, x_n]^T$. In this area, no general method exists leading to a solution of the non-linear system $\mathbf{f(x) = 0}$ in a predefined number of steps, even when $N = 1$!

3.6.1 Bisection

In the case where the 'system' consists of a single non-linear equation the method of bisection may be used to find a solution to $f(x) = 0$ in the interval $[a, b]$ if f is continuous and if $f(a)f(b) < 0$. Having taken up positions on either side of a root, one of the bounds is replaced with the midpoint $\frac{b-a}{2}$ such that the root is still straddled. This process is repeated until the interval is sufficiently small, or until a zero has been discovered (Fig. 3.3). Rather than using the midpoint in a comparison, the method may be improved on by using the root of a linear approximation to the function, $\frac{f(b)(a-b)}{f(a)-f(b)}$, and the method is called the *rule of false position*. Replacing the upper bound, b, with this approximate root and replacing the lower bound a with the previous upper-bound is known as *the Secant Method*. which does not need to straddle a root although convergence to a solution is not always guaranteed.

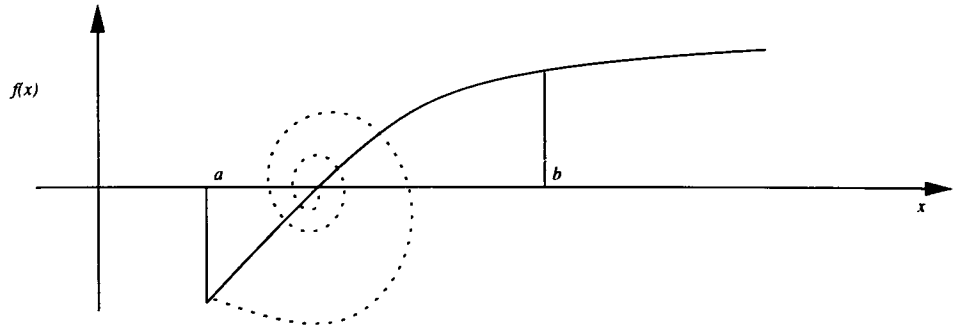


Figure 3.3: Root finding via bisection

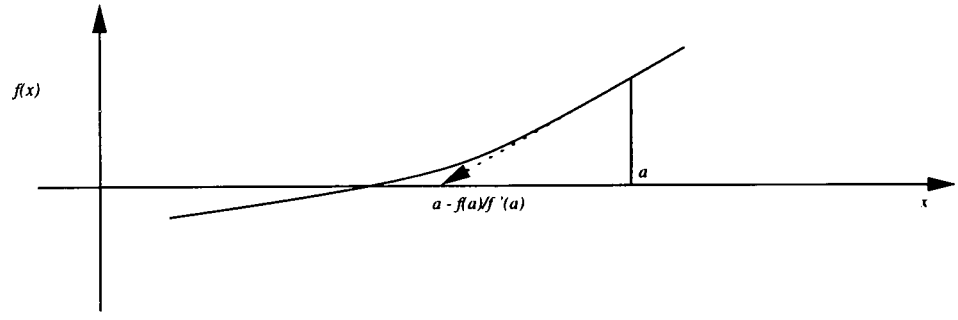


Figure 3.4: Newtons method for root finding

3.6.2 Newton's Method

By assuming an initial approximation, a, to a root, Newton's method proceeds by calculating the next approximation as the intercept of $\frac{df}{dx}\Big|_a$ with the x axis, i.e. $a - \frac{f(a)}{f'(a)}$ (Fig. 3.4). This method is often referred to as the *Newton-Raphson* method, but since we deal with its extension (Newton's method) solving systems of equations we abbreviate its title.

3.6.3 Newton's Method for Systems of Non-Linear Equations

The iteration step in the Newton-Raphson method

$$x_r = x_{r-1} - \frac{f(x_{r-1})}{f'(x_{r-1})}$$

may be readily generalised to the multivariate case, leading to the iteration

$$\mathbf{x}_r = \mathbf{x}_{r-1} - \mathbf{J}(\mathbf{x}_{r-1})^{-1}\mathbf{f}(\mathbf{x}_{r-1})$$

where $\mathbf{J}(\mathbf{x})$ is the $n \times n$ Jacobian matrix of partial derivatives evaluated at $\mathbf{x}$ having the form

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

To avoid having to compute an inverse Jacobian at each step the iteration is rewritten

$$\mathbf{J}(\mathbf{x}_{r-1})\mathbf{x}_r = \mathbf{J}(\mathbf{x}_{r-1})\mathbf{x}_{r-1} - \mathbf{f}(\mathbf{x}_{r-1})$$

That is, at each stage we solve a linear system $\mathbf{A}_r\mathbf{x}_r = \mathbf{b}_r$ where $\mathbf{A}_r = \mathbf{J}(\mathbf{x}_{r-1})$ and $\mathbf{b}_r = \mathbf{J}(\mathbf{x}_{r-1})\mathbf{x}_{r-1} - \mathbf{f}(\mathbf{x}_{r-1})$ using techniques described in Sections 3.2/3.3. In practice this system may be solved more economically via the solution of

$$\mathbf{J}(\mathbf{x}_{r-1})\delta\mathbf{x}_r = -\mathbf{f}(\mathbf{x}_{r-1})$$

where $\delta\mathbf{x}_r = \mathbf{x}_r - \mathbf{x}_{r-1}$.

3.7 Optimisation

Optimisation involves finding an $\mathbf{x}$ at which a function $\mathbf{f}(\mathbf{x})$ has its minimum value[5]. For an equation in one dimension a simple method is *the Golden Section Search* where in an interval $[a, b]$ two internal points $x_1 = b - r(b-a)$ and $x_2 = a + r(b-a)$ are calculated and at each step either a is replaced by x_1 or b is replaced by x_2, depending on which causes the updated interval to straddle the minimum value. If r satisfies $r : (1-r) = 1 : r$ (the Golden Ratio) only one new function invocation is required at each stage.

3.7.1 Method of Steepest Descent

If the first derivatives of $\mathbf{f}$ are known then an obvious choice of method is that of *Steepest Descent* in which the iteration step is

$$\mathbf{x}_r = \mathbf{x}_{r-1} - \alpha \frac{\nabla \mathbf{f}(\mathbf{x}_{r-1})}{||\nabla \mathbf{f}(\mathbf{x}_{r-1})||}$$

where ∇ is defined as $\nabla f = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j} + \frac{\partial f}{\partial z}\mathbf{k}$ and α is computed via a one-dimensional minimisation of $\phi(t) = \mathbf{f}\left(\mathbf{x}_{r-1} - t\frac{\nabla \mathbf{f}(\mathbf{x}_{r-1})}{||\nabla \mathbf{f}(\mathbf{x}_{r-1})||}\right)$. One problem with using the method of steepest descent is that a minimum along a line has a gradient which is orthogonal to that line and hence the path taken by successive estimates is full of right angles! A better choice for minimisation directions makes use of *conjugate directions* and leads to *the conjugate gradient method* which was mentioned briefly earlier.

[5] We do not consider maximising a function since a maximum value of f is a minimum value of $-f$.

3.7.2 Conjugate Gradients

Although the conjugate gradient method minimises a non-linear function, for the purpose of this thesis we only consider its use as a linear system solver. To understand how a linear system may be solved via the minimisation of a non-linear system, consider the equations

$$E_1 : 10x = 20, \qquad E_2 : y = \frac{10x^2}{2} - 20x$$

Solving E_1 is equivalent to minimising E_2 since $\frac{d(5x^2-20x)}{dx} = 10x - 20$. Similarly, solving the symmetric, positive definite[6] linear system

$$\mathbf{Ax} = \mathbf{b}$$

is equivalent to minimising the quadratic

$$Q(x) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}$$

since $\mathbf{Ax} - \mathbf{b} = \nabla\left(\frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{b}^T\mathbf{x}\right)$. Like the steepest descent method, iterations are of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k\mathbf{p}_k$$

where the $\mathbf{p}_k$ are direction vectors and the α_k are scalars specifying distance moved. The α_k are chosen so that Q is minimised in the direction of $\mathbf{p}_k$ which can be shown to be when

$$\alpha_k = \frac{\mathbf{p}_k^T(\mathbf{Ax}_k - \mathbf{b})}{\mathbf{p}_k^T\mathbf{Ap}_k}.$$

If we choose $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$ such that they are *conjugate* with respect to $\mathbf{A}$, i.e. they satisfy $\forall i \neq j$: $\mathbf{p}_i^T\mathbf{Ap}_j$ then we have a *conjugate direction method* and the iteration is guaranteed to converge in no more than n steps [40]. The most efficient way, in general, to obtain the direction vectors to be used in solving $\mathbf{Ax}=\mathbf{b}$ is via the *conjugate gradient method*, a conjugate direction method which calculates the k^{th} direction vector at the $(k-1)^{th}$ iteration. The iteration step of one formulation of the conjugate gradient method is as follows:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k - \alpha_k\mathbf{p}_k \\ \mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k\mathbf{Ap}_k \\ \mathbf{p}_{k+1} &= \mathbf{r}_{k+1} - \beta_k\mathbf{Ap}_k \end{aligned}$$

where

$$\begin{aligned} \alpha_k &= -<\mathbf{r}_k, \mathbf{r}_k> / <\mathbf{p}_k, \mathbf{Ap}_k> \\ \beta_k &= <\mathbf{r}_{k+1}, \mathbf{r}_{k+1}> / <\mathbf{r}_k, \mathbf{r}_k> \end{aligned}$$

where the $\mathbf{r}_k$ are the residuals $\mathbf{r}_k = \mathbf{Ax}_k - \mathbf{b}$, $\mathbf{p}_0 = \mathbf{r}_0$, and $<\mathbf{x}, \mathbf{y}> = \mathbf{x}^T\mathbf{y}$. An advantage of the conjugate gradient method is that the coefficient matrix, $\mathbf{A}$, need not be explicitly formed, the only requirement being that its representation must support multiplication by an arbitrary vector.

[6] $\mathbf{A}$ is positive definite: $\Rightarrow \forall \mathbf{x} : \mathbf{x}^T\mathbf{Ax} > 0$.

3.8 Differential Equations

Although we mainly concentrate on partial differential equations (PDEs) (differential equations involving more than one independent variable) we also briefly mention ordinary differential equations (ODEs) and so briefly introduce methods used to solve these. An n^{th}-order ODE has the general form

$$\frac{d^n y}{dx^n} = f\left(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \ldots, \frac{d^n y}{dx^n}\right)$$

with the useful property that an n^{th} order ODE may always be transformed into a system of n first order ODEs. ODEs of the form $dy/dx = f(x, y)$ are often solved by advancing a solution, i.e. if $y_k = y(x_k)$ is known then $y_{k+1} = y((1 + \Delta)x_k)$ may be calculated via

$$y_{k+1} = y_k + \sum_{i=0}^{N} w_i f(g_i(x_k, y_k))$$

with the parameter values being derived from a Taylor series expansion, leading to *Euler's method* and *N^{th}-order Runge-Kutta Methods* which may be straightforwardly applied to systems of ODEs and hence to higher-order ODEs. Although this introduction does not do justice to solution methods for ODEs it is a sufficient introduction for our purpose as we only mention ODE solution methods when discussing quantitative statistics derived from software developed elsewhere. We move on to solution methods for PDEs which, of course, may be applied to ODEs.

3.8.1 Partial Differential Equations

The two methods that we discuss for the solution of PDEs are the *finite difference* and *finite element* methods.

3.8.2 The Finite Difference Method

In the finite difference method, the domain of a function $u(x, y)$ is discretised (usually at regular intervals [see Fig. 3.5]) into a set of points of interest. For example, the *Poisson equation*

$$\frac{\partial^2 u}{dx^2} + \frac{\partial^2 u}{dy^2} = \rho(x, y)$$

with known values on the boundary of a region is rewritten as a difference equation, where h is defined to be the distance between adjacent mesh points, using techniques from Section 3.4:

$$\frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{h^2} \simeq \rho_{ij}$$

If the points of interest are arranged as a vector $\mathbf{x}$ then this forms a linear system which may be solved via techniques from Section 3.1, or by special *fact forms* of Gaussian elimination if the resulting system is tridiagonal. Iterative techniques such as Jacobi or Gauss-Seidel are often used on these problems as they may be easily formulated in terms of a point in the discretised domain and its local neighbours, and the matrix-vector representation need not be explicitly formed.

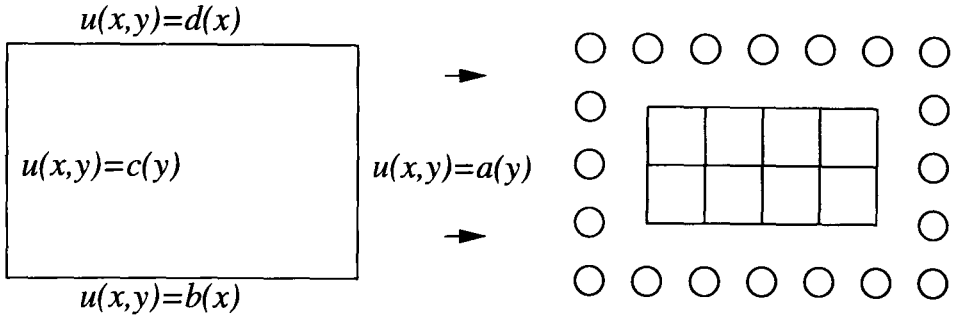


Figure 3.5: Discretisation of a 2D region

3.8.3 The Finite Element Method

The Finite Element Method (FEM) approximates the solution to a PDE over a region of interest by discretising the domain using an element mesh and computing the solution of the PDE at the node points of these *elements*. This is usually done by representing the solution of the PDE at the node points as the solution of the linear system $\mathbf{Ax} = \mathbf{b}$. The *stiffness matrix* $\mathbf{A}$ and the *force vector* $\mathbf{b}$ are constructed by performing integrations over each element and summing these *contributions* (Fig. 3.6). The solution of the PDE over the region is then approximated using interpolation functions that depend on the values at neighbouring node points. A typical FEM domain discretisation is shown in Fig. 3.7. The stiffness matrix that typically arises is symmetric, positive definite and hence the conjugate gradient method may be used for the system's solution. This very simple explanation of the FEM does not do justice to the method and is only meant to form the context for the following material, rather than an introduction to the method. For a more in-depth introduction see [69, 1].

3.9 Symbolic Manipulation

Symbolic manipulation, sometimes called computer algebra, is an area of scientific computing which is concerned with the manipulation of expressions in an analytical style. For instance, in a Computer Algebra System (CAS) differentiation of the function

$$y = \sin^2(x) + \cos^2(x) + x^2$$

would yield

$$\frac{dy}{dx} = 2x$$

The advantage of symbolic manipulation is that it allows problems to be solved which might otherwise prove troublesome. For instance, simple evaluation of the expression

$$p(x) = \frac{x}{\sin(x)}$$

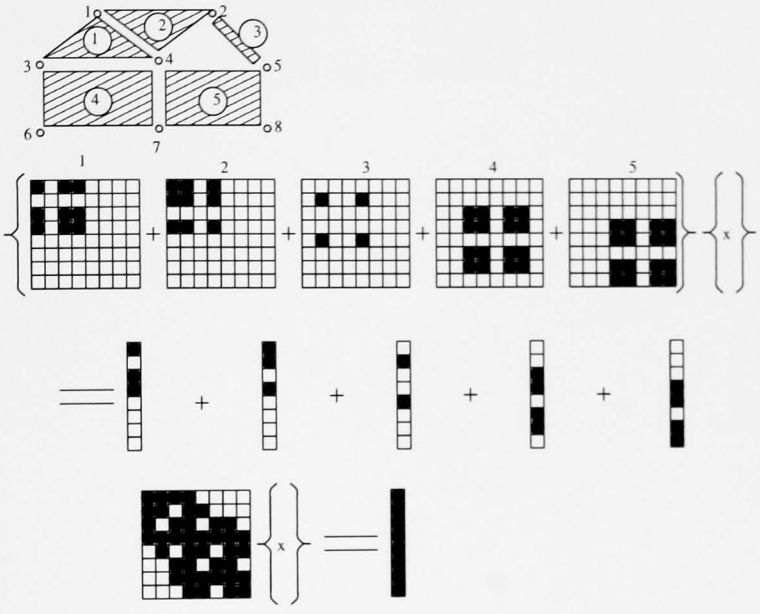


Figure 3.6: Structural finite element assembly calculations

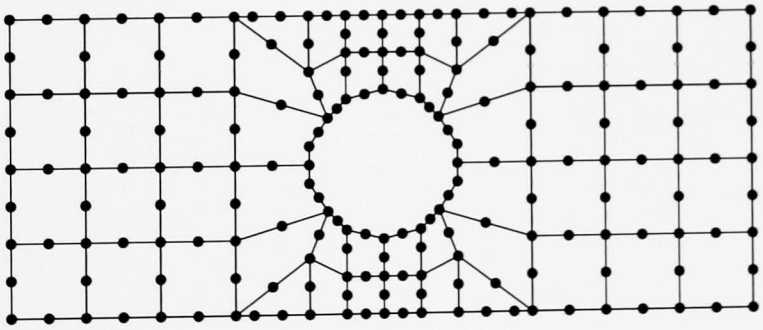


Figure 3.7: 2D finite element mesh

at the point x=0 would result in a division by zero. But we may easily work out the limit at $x = 0$ with l'Hôpitals rule[7] via symbolic manipulation and use the formula:

$$p(x) = \begin{cases} 1 & \text{if } x = 0 \\ \frac{x}{\sin(x)} & \text{otherwise} \end{cases}$$

[7] **L'HÔPITAL'S RULE** [81](p. 94):
if $\frac{f'(x)}{g'(x)} \to A$ as $x \to a$ and $g(x) \to 0$ and $f(x) \to 0$ as $x \to a$, then $\frac{f(x)}{g(x)} \to A$ as $x \to a$.

Chapter 4

Use of Haskell

In this chapter we discuss the use of purely functional languages to implement numerical algorithms. We consider how functional languages may be used to denote these algorithms concisely, the relative efficiency with which the algorithms may be executed, and identify factors within these programs which affect this efficiency.

4.1 Plan of Study

The organisation of the chapter is shown in Fig. 4.1. We initially discuss the choice of language/style and present motivating examples written without regard for execution efficiency. We use LU factorisation as a case study of some of the difficulties associated with the lack of mutable state and offer solutions to these problems in two distinct ways:

- We use alternative data structures to arrays, including quadtrees which have the advantage of providing an efficient sparse matrix representation and encouraging implicit parallel *divide and conquer*-style algorithms.
- We use alternative mathematical techniques to circumvent the need for updateable state.

We also demonstrate how these techniques can be combined. In addition, we discuss how other common numerical algorithms, such as successive over-relaxation (SOR) and cyclic reduction, may be expressed efficiently in terms of quadtrees and compare these formulations with related work.

4.2 Choice of Language

The languages which we concentrate on are non-strict and purely functional.

- Non-strict languages are more expressive than strict languages,
  i.e. $\forall e : \mathsf{EVAL}_{\text{Strict}}[\![e]\!] \preceq \mathsf{EVAL}_{\text{Non-Strict}}[\![e]\!]$

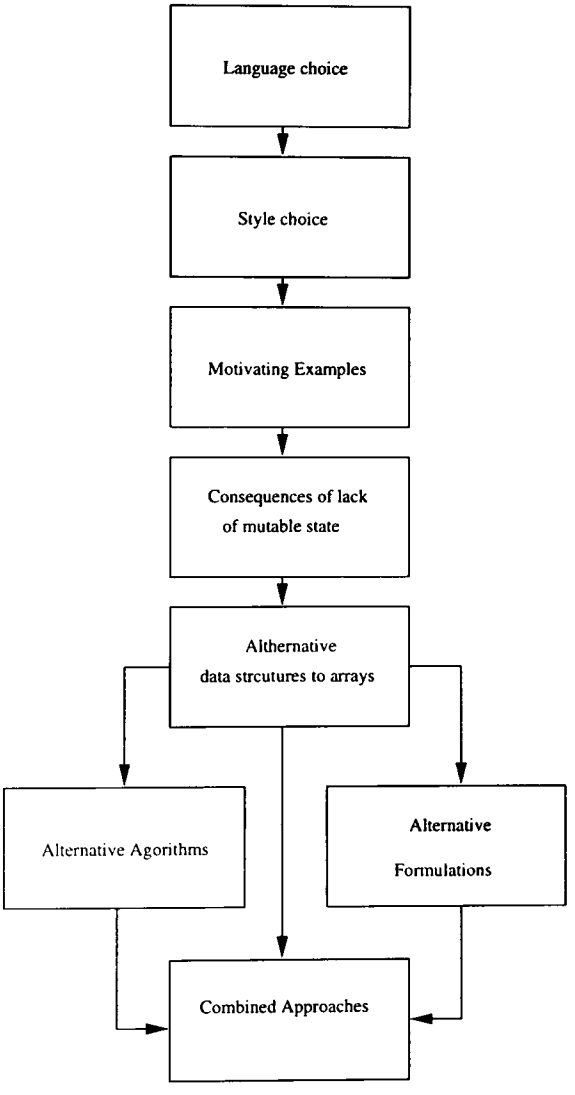


Figure 4.1: Plan of study

- Structures may be defined recursively, i.e. $x = f(x) \not\vdash \mathsf{EVAL}[\![x]\!] = \bot$. A consequence of this is that arrays may be defined recursively.
- Purely functional languages do not allow side effects of any form, and hence equational reasoning is everywhere valid.

This severely limits our possible choice of languages. From the remaining possibilities we choose Haskell (GHC 0.26) because:

- Haskell is *the* standard non-strict functional programming language.
- Haskell implementations are *much* more efficient than other non-strict functional languages [45].
- Haskell compilers exist and are freely available.

Although the current version of Haskell is 1.4, the version of Haskell we use is version 1.2 since

- This was the *de-facto* standard at the beginning of this work.
- The differences between versions 1.2 and 1.4 are concerned mainly with I/O and GoFER-style constructor classes (two features which this thesis does not rely on).
- At the time of writing, implementations of versions 1.3/1.4 are still error-prone and the greater functionality provided by the later language definition causes major inefficiencies.
- Many people have claimed that versions 1.3/1.4 are too complex to use in practice [51] and there is a move to limit some of these features as Haskell approaches standardisation to Standard Haskell.

4.3 Choice of Style

The style of functional programming we adopt is *elementary*. By elementary we mean programs where there is no hidden state. This idea of state may seem strange since functional programs are stateless, but many techniques exist for modelling state in a purely functional manner. For example, the contrived imperative example

```
w =  x;
w +=  y;
w *=  w;
...
```

represents the computation $w = (x + y) * (x + y)$. In a functional language this could either be written as

```
>     let
>             w = z * z
>             z = x + y
>     in
>             ...
```

or expressed, with appropriate functions `incBy` and `mulBy`, as

```
>     do
>             w <- return x
>             w <- incBy y
>             w <- mulBy w
>             ...
```

where great effort has been taken to model an imperative flow of control. The above fragment is a syntactically sugared version of

```
>         return x     >>= (\w ->
>         incBy y      >>= (\w ->
>         mulBy w      ))
>         ...
```

where (>>=) is a higher-order infix-combining function that models the flow of control in an imperative program, and Currying is used to carry the current state of the machine around as an extra *hidden* parameter. The advantages of this kind of *imperative* functional programming is that it can often be compiled into its imperative equivalent and imperative algorithms may be expressed in a functional language very easily. Hence FORTRAN-style array algorithms could be transliterated into functional languages given a library of combining functions which model FORTRAN features. However, we reject this style of functional programming as it suffers from the same drawback as imperative programming, i.e. it imposes a serial control flow that is difficult to reason about. The style of functional programming we adopt could be considered *applicative* functional programming as opposed to *imperative* functional programming, where we specifically avoid the use of state-modelling combinators and hidden parameters.

4.4 Denotation of Numerical Methods

In this section we discuss functional denotations of numerical methods. We use the phrase *denotations* rather than implementations to stress the fact that each script is written in terms of abstract values, rather than as a mapping onto a physical machine. Initially, we concentrate on the expressiveness of Haskell via the use of motivating examples before proceeding with a discussion of points arising from using Haskell as an implementation tool.

4.4.1 Linear Systems

A significant component of numerical problem solving is concerned with the solution of (dense) linear systems. In this area we find that algorithms such as LU factorisation can be expressed in a manner that matches, very closely, their mathematical definition. The array comprehensions of Haskell lend themselves naturally to this area.

As mentioned in Chapter 3, the LU factorisation of an $n \times n$ matrix $\mathbf{A}$ consists of two matrices $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{A}$ is expressed as the product $\mathbf{A} = \mathbf{LU}$. $\mathbf{U}$ being an $n \times n$ upper-triangular matrix and $\mathbf{L}$ being an $n \times n$ lower-triangular matrix. Since both $\mathbf{L}$ and $\mathbf{U}$ are triangular, $(\mathbf{LU})\mathbf{x} =$

$\mathbf{L(Ux) = b}$ can be solved via forward and back substitution. If we are to perform this factorisation we must decide which form of L and U to take as more than one possibility exists. If we set $l_{ii} = 1$, then by omitting explicit representation of this diagonal, we can store the factorisation in the same amount of space as A. The mathematical formulation for this can be expressed as

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) \Big/ u_{jj} \quad (i > j)$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \quad (i \leq j)$$

and a Haskell implementation as

```
>       type Matrix = Array (Int,Int) Double
>       lu :: Matrix -> Matrix
>       lu a = luf where
>        luf = array bnds [(i,j) := f i j | (i,j) <- range bnds]
>        bnds = bounds a
>        f i j | (i>j)  = (a!(i,j)-sum[luf!(i,k)*luf!(k,j)
>                                      |k<-[1..(j-1)]])/luf!(j,j)
>              | (i<=j) =  a!(i,j)-sum[luf!(i,k)*luf!(k,j)
>                                      |k<-[1..(i-1)]]
```

Because Haskell is lazy the array elements are evaluated as they are demanded, guaranteeing that the operations occur in the correct order. The 1-1 relationship between the mathematical specification and the Haskell implementation is valuable as it *allows direct correspondence of functional code with the mathematical equations, which makes the code easy to develop, write, read and modify* [24], meaning textbook matrix algebra can often be translated almost verbatim into a non-strict functional language. It is interesting to note that this definition of LU factorisation is not possible in a strict language, such as ML or SISAL, as it would lead to an infinite recursion. The formulation is referred to as a *compact* scheme since the elements in the final triangular form are obtained by accumulation, dispensing with the computation and recording of intermediate coefficients and reducing roundoff errors [101]. The formulation is known as the *Doolittle (Black)* [11, 35, 63, 34] method of LU factorisation but other compact factorisation schemes such as *Crout (Banachiewicz, General Cholesky)* [12, 27, 35, 46, 47, 63, 34]

$$u_{ii} = 1 \quad (implicit)$$

$$u_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right) \Big/ l_{jj} \quad (i < j)$$

$$l_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \quad (i \geq j)$$

or *symmetric Cholesky (Square-Root Method,Banachiewicz)* [11, 27, 35, 34]

$$l_{ii} = u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2}$$

$$l_{ji} = u_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right) \Big/ u_{ii} \quad (i < j)$$

may also be expressed in Haskell equally as easily.

4.4.2 Symbolic and Analytic Methods

In Haskell it is almost trivial to write a function to perform analytic differentiation over expressions. For example, [87] (p. 53) includes the following formulae:

13.2 $\frac{d}{dx}(c) = 0$
13.3 $\frac{d}{dx}(cx) = c$
13.4 $\frac{d}{dx}cx^n = ncx^{n-1}$
13.5 $\frac{d}{dx}(u \pm v \pm w \pm \cdots) = \frac{du}{dx} \pm \frac{dv}{dx} \pm \frac{dw}{dx} \pm \cdots$

Given an appropriate datatype this may be written as

```
> diff :: Expr -> Expr
> diff  (CON c)                          = (CON 0)
> diff ((CON c) 'MUL' VAR)               = (CON c)
> diff ((CON c) 'MUL' VAR 'POW' (CON n)) = CON (n*c) 'MUL' VAR 'POW' CON (n-1)
> diff (u 'ADD' v)                       = diff u 'ADD' diff v
> diff (u 'SUB' v)                       = diff u 'SUB' diff v
```

and again the close relationship between the textbook description and the functional specification is preserved. This form of manipulation may be extended so that the input to a differential equation solver may be written in terms of an *expression* datatype rather than as a series of constants and flags. For instance, by considering the variables u, t, x as variables U_0, U_1, U_3 the parabolic equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

may be be written as

```
D (U 0) (U 1) :=: D :^: 2 (U 0) (U 2)
```

and the first stage of the computation could dis-assemble this via pattern matching and derive an appropriate finite-difference representation. As the required solution's accuracy increases (i.e. more computational power is required) this overhead becomes negligible. With a parser and a pretty printer placed at each end we may easily construct a rudimentary computer algebra system.

4.4.3 n-point Gaussian Quadrature

Another significant component of numerical problem solving is the area of numerical quadrature, where an analytic integral is approximated via one of a variety of methods. The applicative code to carry out an algorithm such as Gaussian quadrature is also as terse, yet expressive, as the mathematical specification. The use of lists greatly increases our ability to implement very general *n-point* quadrature: simply by passing the appropriate weights and abscissae to the integration algorithm.

The method we consider estimates a definite integral

$$I = \int_a^b f(x)dx$$

by approximating f using a polynomial and integrating analytically the approximating function. The estimate is of the form

$$R = \sum_{i=1}^{n} w_i f_i$$

where $f_i = f(x_i)$, and the x_i are the *abscissae*, or *integration points*, with corresponding *weights* w_i. The formulation given below is a *symmetric* n-point Gaussian quadrature:

$$\int_a^b f(y)dy \simeq p_r \sum_{i=1}^{n} w_i(f(p_m + p_r x_i) + f(p_m - p_r x_i))$$

$$\text{where} \quad p_m = \frac{b+a}{2}$$
$$p_r = \frac{b-a}{2}$$

and a Haskell implementation is given by

```
>       type IntegrationRoutine =
>            (Double -> Double) -> (Double,Double) -> Double
>       gq :: [Double]->[Double]-> IntegrationRoutine
>       gq x w f (a,b) = pr * sum (zipWith f1 w x)
>        where f1 wi xi = wi * (f (pm+pr*xi) + f (pm-pr*xi))
>              pm = (b+a)/2.0
>              pr = (b-a)/2.0
```

where `zipWith` is a higher-order function which applies a function `f` elementwise to two lists such that, for example, `zipWith (+) [1,2] [3,4] = [4,6]`.

4.4.4 Multidimensional Integration

The use of quadrature can very easily be extended to encompass multidimensional integration using the higher-order nature of functional languages to write a general n-dimensional integration routine, passing an arbitrary 1-dimensional quadrature function as an argument to a higher-order multiple integrating function:

```
>       type Limit = Point->(Double,Double)
>       type Fn    = Point -> Double
>       type Point = [Double]

>       multipleIntegral :: IntegrationRoutine -> [Limit] -> Fn -> Double
>       multipleIntegral int limits f = mi int [] limits f
>        where
>          mi int g [ab]      f = int f1 (ab g)
>                where f1 x = f (g++[x])
>          mi int g (ab:lims) f = int f1 (ab g)
>                where f1 x = mi int (g++[x]) lims f
```

Thus, to evaluate

$$I = \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dzdydx$$

we integrate a 1-D function $H(x)$ as

$$I = \int_{x_1}^{x_2} H(x)dx$$

using our quadrature scheme, where $H(x)$ is defined as

$$H(x) = \int_{y_1(x)}^{y_2(x)} \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dzdy$$

which may be evaluated for each abscissae x_i of $\int H(x)dx$ via the above method.

The simplicity and terseness of this formulation is apparent when this is compared with a comparable but less general algorithm [79] expressed in Pascal taking 50+ lines[1]. This terseness is due directly to the ability to create functions *on the fly*, a motivating argument for functions being first order.

4.4.5 Example

As an example of this in use we can define the function `integrate` by combining our multi-dimensional integration function `multipleIntegral` and our Gaussian quadrature function `gq` with appropriate abscissae and weights.

```
>       integrate :: [Limit] -> Fn -> Double
>       integrate limits f = multipleIntegral (gq abscissae weights) limits f
>        where abscissae = ...
>              weights   = ...
```

[1] The figure 50+ relates to the fact that some Pascal procedures were not implemented, only described, in [79].

Using these functions we can very simply compute the volume of a sphere, or area of a circle, by specifying appropriate limits. The required formulae are

$$area_r = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} dydx$$

$$volume_r = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-y^2-x^2}}^{\sqrt{r^2-y^2-x^2}} dzdydx$$

and the integration limits can be expressed as

```
>       circle_lims :: Double -> [Limit]
>       circle_lims r = [lim1,lim2]
>        where lim1 _           = (-r,r)
>              lim2 (x:_)       = (-sq,sq)
>               where sq = sqrt(r*r - x*x)

>       sphere_lims :: Double -> [Limit]
>       sphere_lims r
>        = circle_lims r ++ [lim3]
>            where lim3 (x:y:_) = (-sq,sq)
>                    where sq = sqrt(r*r - y*y - x*x)
```

We can now express the final computation as tersely as we would expect it to appear in text, i.e. $\int_{circle} dydx$.

```
>       area, volume :: Double -> Double
>       area   r = integrate (circle_lims r) (const 1)
>       volume r = integrate (sphere_lims r) (const 1)
```

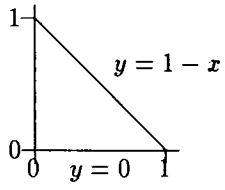
where `const` may be defined as

```
>       const :: a -> b -> a
>       const    k    x =  k
```

4.4.6 Basis Functions for Finite Element Analysis

These ideas can be extended to the specification of interpolation functions for finite element analysis, where we can very generally state facts about iso-parametric basis functions. The essence of the finite element method (FEM) is to partition the domain of a problem into non-overlapping *elements* and provide an approximate solution which has a simple form within each element. The local representations are then patched together to form a global solution. The approximate solution is

calculated using *basis*, or *interpolation*, functions dependent on the values at neighbouring node points. The simplest approach is to interpolate linearly within an element:



In Haskell we can write this as

```
>       type Element = ([Interpolation],[Limit],(Int,Int))
>       simplex_2D :: Element
>       simplex_2D    = (interpolations,limits,(dimensions,nodes))
>        where interpolations = [i1,i2,i3]
>                    where i1 (r:s:_) = 1 - r - s
>                          i2 (r:s:_) =     r
>                          i3 (r:s:_) =         s

>              limits = [l1,l2]
>                    where l1 _     = ( 0 ,  1 )
>                          l2 (x:_) = ( 0 , 1-x )

>              nodes  = 3
>              dimensions = 2
```

allowing us to model simplex elements very naturally, since the interpolation functions for a 1D simplex are

$$\begin{aligned} H_1(r) &= 1 - r \\ H_2(r) &= r \end{aligned}$$

and the interpolation functions for a 2D simplex are

$$\begin{array}{l|l} H_1(r,s) = 1 - r & -s \\ H_2(r,s) = r & \\ H_3(r,s) = & s \end{array}$$

This explains our use of `(x:y:_)` rather than `(x,y)` in that we express functions in a form that allows us to speak more generally later about the dimensionality.

Now that an element has been specified, one of the things we need to do is to integrate a function over its limits, which can also be specified very tersely via the definition of the projection `getLimits`[2].

[2] In Haskell `f.g` $\equiv f \circ g$ denotes function composition.

```
>       getLimits :: Element -> [Limit]
>       getLimits (_,limits,_) = limits

>       integrateOverElement :: Element -> Fn -> Double
>       integrateOverElement = integrate.getLimits
```

We can now integrate any function f over the element via

```
>       integrateOverElement simplex_2D f
```

4.4.7 Finite Element Analysis

The techniques described in the previous sections were brought together to form a simple finite element package implemented in Haskell. One of the requirements was that it should work in an arbitrary number of dimensions with plug-in basis functions and a plug-in linear system solver.

The notation provided by Haskell was found to be very useful in the development of this system, often making the problem description clear. For example, in the FEM we establish the element equations for each element. Generally this is done by substituting the interpolation functions into the governing integral form. Historically these matrices have been called the *element stiffness matrix* and *load* (or *force*) *vector*, respectively[3]. Once the element equations have been established the contributions from each element are summed to form the *system of equations* $\mathbf{Sx} = \mathbf{F}$. This can be expressed in Haskell as

```
>       linear_system :: Mesh -> Basis_Fn -> Linear_System

>       linear_system mesh shape = MkLinear_System sm fv
>         where sm = sum [contrib_to_sm shape e | e <- elements]
>               fv = sum [contrib_to_fv shape e | e <- elements]
>               elements = getElements mesh
```

where the element stiffness matrix and force vector depend on the interpolating function **shape** and the particular element being considered[4].

By using Haskell's type class mechanism to define instances over an *Expression* datatype (see Section 4.4.2), high level descriptions of problems can be specified very naturally.

4.4.8 Example

Consider the simple 1-D linear slider bearing shown in Fig. 4.2, which is assumed to extend to infinity out of the plane of the figure [1] (p. 207). It consists of a rigid bearing and a slider moving relative to the bearing with velocity U.

[3] This naming scheme relates to a time when the FEM was predominately used by engineers for structural analysis.

[4] The types `Mesh` and `LinearSystem` have not been defined explicitly as any representation of a finite element mesh and Matrix-Vector would do.

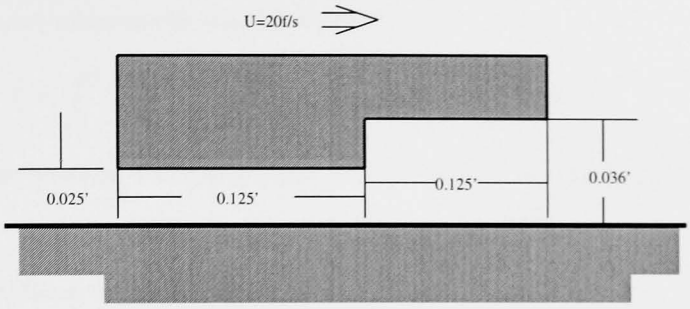


Figure 4.2: Linear slider bearing

The extremely thin gap between the bearing and the slider is filled with an incompressible lubricant having viscosity v. For the one-dimensional case the governing Reynolds equation reduces to

$$\frac{d}{dx}\left(\frac{h^3}{6v}\frac{dP}{dx}\right) = -\frac{d}{dx}(Uh)$$

where $P(x)$ denotes the pressure and $h(x)$ the distance between the slider and the bearing. The boundary conditions are that P must equal the known external pressures (usually zero) at the two ends of the bearing. Considering the governing differential equation as

$$L(P) = Q$$

we may use a weighted residual method by assuming an approximate solution

$$P^*(y) = x_1\phi_1(y) + x_2\phi_2(y) + \cdots + x_n\phi_n(y)$$

where the $\phi_i(x)$ are basis functions satisfying our boundary conditions, and define a residual error term

$$R = L(P^*) - Q$$

Although we cannot force this term to vanish we can force a weighted integral, over the solution domain, of the residual to vanish. That is, the integral over the solution domain (Ω) of the product of the residual term and some weighting function is set equal to zero:

$$I_i = \int_\Omega w_i(x) R d\Omega = 0, \qquad i = 1, 2, \ldots, n$$

In the above example we have

$$I_i = \int_0^L w_i(x)\left[\frac{d}{dx}\left(\frac{h^3}{6v}\frac{dP^*}{dx}\right) + \frac{d}{dx}(Uh)\right]dx = 0, \qquad i = 1, 2, \ldots, n$$

which, after integration by parts, gives

$$I_i = \left[w_i(x)\{\frac{h^3}{6v}\frac{dP^*}{dx} + Uh\}\right]_0^L - \int_0^L \frac{dw_i(x)}{dx}\left(\frac{h^3}{6v}\frac{dP^*}{dx} + Uh\right)dx = 0, \qquad i = 1, 2, \ldots, n.$$

Selecting $w_i(x) = \phi_i(x)$ allows us to simplify this to[5]

$$I_i = \int_0^L \frac{d\phi_i(x)}{dx}\left(\frac{h^3}{6v}\frac{dP^*}{dx} + Uh\right)dx = 0, \qquad i = 1, 2, \ldots, n$$

Writing the basis functions as a vector $[\phi_1, \phi_2, \ldots, \phi_n]^T$ and P^* as a dot product

$$P^* = [\phi_1, \phi_2, \ldots, \phi_n][x_1, x_2, \ldots, x_n]^T$$

defines our global stiffness matrix and force vector $Sx = F$:

$$\underbrace{\frac{1}{6v}\int_0^L h^3 \begin{bmatrix} \phi_1' \\ \phi_2' \\ \vdots \\ \phi_n' \end{bmatrix} [\phi_1', \phi_2', \ldots, \phi_n']dx}_{S} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}_{x} = \underbrace{-U\int_0^L h \begin{bmatrix} o_1' \\ o_2' \\ \vdots \\ o_n' \end{bmatrix} dx}_{F}$$

and defines the element stiffness matrices and force vectors as [1]

$$S^e = \frac{1}{6v}\int_{x_i}^{x_j} h^3 H_x^{eT} H_x^e dx$$

$$F^e = -U\int_{x_i}^{x_j} h H_x^{eT} dx$$

where the H_x^e are the element interpolation functions.

The element stiffness matrices and force vectors can be written in Haskell using an *Expression* datatype as[6]

```
>       element_sm_and_fv :: Bas -> Bas -> Point -> Representation
>       element_sm_and_fv f g (x:_)
>        = ((h * h * h)/(6.0 * v)) * g' * f' :=: -u * h * g'
>           where
>                 g' = D g (U 1)
>                 f' = D f (U 1)
>                 u = 20.0                -- velocity  = 20 ft/s
>                 v = 0.002               -- viscosity = 0.002 lb s/ft^2
>                 h = thickness x         -- thickness at point x
>                 thickness x | x < 0.125 = 0.0250 -- ft
>                             | x > 0.125 = 0.0360 -- ft
```

[5] Since $\forall i : \phi_i(0) = \phi_i(L) = 0$. Setting $w_i(x) = \phi_i(x)$ gives rise to the Galerkin [29] method.

[6] `D f (U n)` is taken to mean differentiate `f` with respect to the *nth* variable and $S :=: F$ is taken to mean that after integration S and F form the stiffness matrix and force vector for a general element.

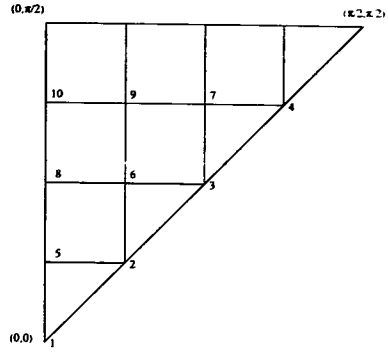


Figure 4.3: Domain of Poisson problem

where `Bas` and `Representation` are simply a suitable coding of *Expressions*. Similarly, for a 2-D Poisson problem (Fig. 4.3) taken from [69] (p.82)

$$\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} - 2 = 0$$

whose Galerkin equations become:

$$\sum_{k,l=1}^{T} U_{kl} \int\int_R \left\{ \left(\frac{\partial \varphi_{kl}}{\partial x}\right)\left(\frac{\partial \varphi_{ij}}{\partial x}\right) + \left(\frac{\partial \varphi_{kl}}{\partial y}\right)\left(\frac{\partial \varphi_{ij}}{\partial y}\right) \right\} dxdy$$

$$+ \int\int_R 2\varphi_{i,j} dxdy = 0, \qquad (i,j = 1,2,\ldots,T)$$

where the $\varphi_{ij}(x,y)$ are the basis functions, and R is the region of interest divided into $(T+1)^2$ square elements, the element stiffness matrices and force vectors can be expressed as :

```
>       element_sm_and_fv :: Bas -> Bas -> Point -> Representation
>       element_sm_and_fv f g (x:y:_)
>        = (fx * gx + fy * gy :=: -2.0 * g)
>           where
>                 gx = D g (U 1)
>                 fx = D f (U 1)
>                 gy = D g (U 2)
>                 fy = D f (U 2)
```

4.4.9 Summary

The above examples provide the motivation behind the use of functional programming languages to express numerical methods. Unfortunately, their lack of direct handling of state often makes some programs run very inefficiently, as shown in the rest of this chapter.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \mathbf{x} = \mathbf{b}$$

Figure 4.4: A non-singular matrix which causes naive LU factorisation to break down

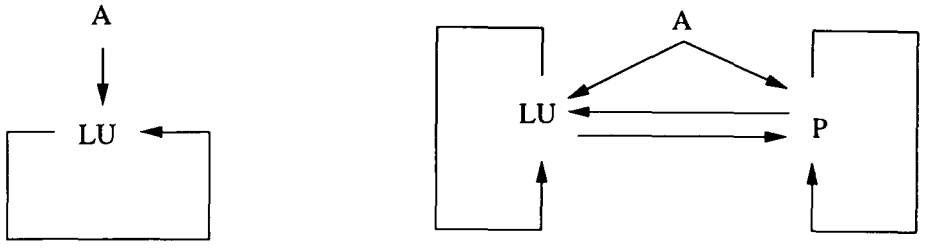


Figure 4.5: Dependencies for LU decomposition without and with partial pivoting

4.5 Case Study: LU Factorisation

In Section 4.4.1 we presented a compact formulation of LU factorisation. Unfortunately, since this formulation does not store intermediate results we cannot perform pivoting in the usual way to retain numerical stability[7]. Even if numerical stability is not an issue, a non-singular matrix such as Fig. 4.4 can still cause this method to fail if one of the pivots is zero. Since the use of compact methods encourages numerical stability it is this approach we concentrate on.

4.5.0.1 Compact Pivoting

The compact methods described earlier may admit pivoting if a permutation matrix is built at the same time as **L** and **U** and all accesses to A are performed in terms of this (Fig. 4.5). Unfortunately this involves storing $N(N+1)/2$ intermediate values, or massive recalculation, and so we reject this scheme.

4.5.0.2 Non-Compact Pivoting

To allow pivoting we must express the algorithm in terms of elimination steps:

```
> lu2 :: Matrix -> Matrix
> lu2 a = eliminate a 1 n where (_,(n,_)) = bounds a

> eliminate :: Matrix -> Int -> Int -> Matrix
> eliminate a i n
```

[7] This lack of pivoting is not as drastic as it first seems since compact methods are inherently more stable than algorithms involving the storage of intermediate results [101] (p. 10).

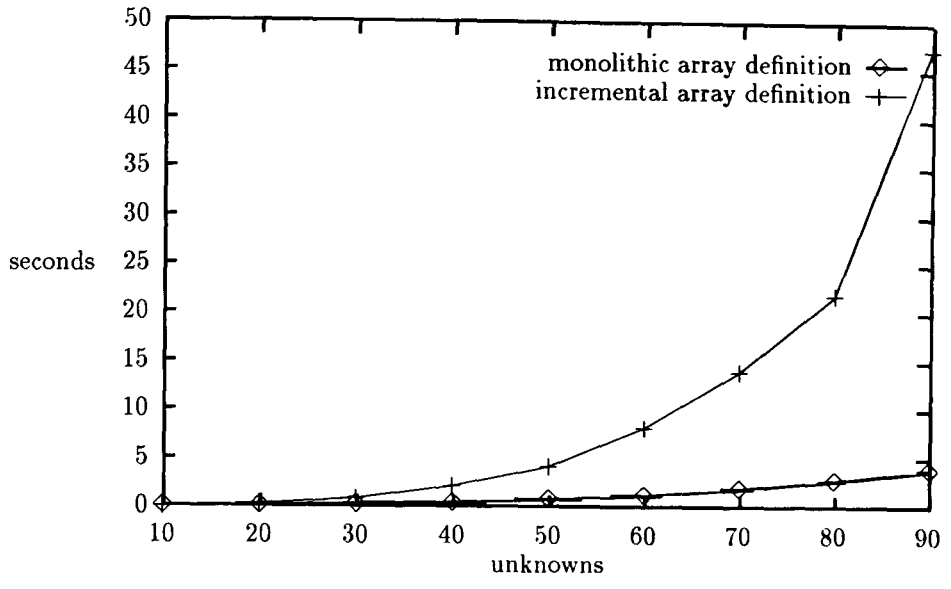


Figure 4.6: Timings(Unix time) for monolithic, incremental array versions of LU factorisation

```
>      | i >= n = a
>      | otherwise
>          = eliminate (a //  submatrix ) (i+1) n
>              where submatrix
>                      = concat[let
>                                 kmul = a!(k,i)/pivot
>                               in
>                                 ([(k,i):= kmul]
>                                     ++
>                                  [(k,j):= a!(k,j)-a!(i,j)*kmul
>                                    |j <- [i+1..n]])
>                               | k <- [i+1..n]]
>                    pivot = a!(i,i)
```

where the algorithm is obscured to ensure that the division by the pivot is shared. If this algorithm were to include pivoting we would calculate `kmul` by searching down the column `i` of `a` for the maximum element and include the row swapping in the association list. Unfortunately, when this technique is used, copying of intermediate arrays causes the program to be unacceptably slow, as can be seen from Fig. 4.6, where the previous compact scheme admits a monolithic array definition and the elimination implies an incremental array definition. This efficiency loss is due to the fact that, at each elimination step, the array update operation (`//`) causes the whole array to be copied. The reason for this copying can be explained via the following example, where `f` acts as a datatype

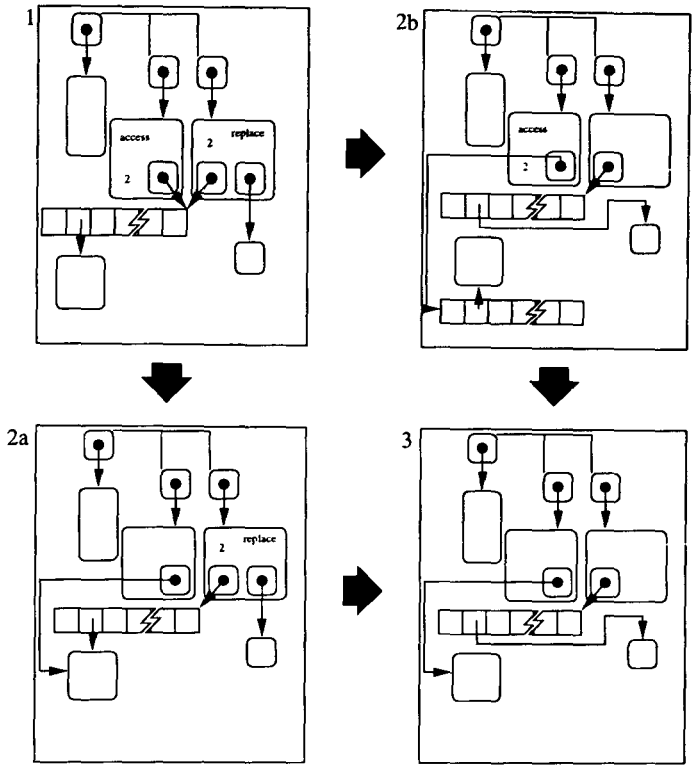


Figure 4.7: Graph transitions for f (access a i) (replace a i p)

constructor with two fields, initially neither argument to f is evaluated,

```
>       x = f (a!i) (a // [i := p])
```

and each argument holds a reference to a. An internal representation is shown in Fig. 4.7. If the left argument is evaluated first then no array copying is necessary (Fig. 4.7: path $1 \rightarrow 2a \rightarrow 3$) as its reference has been freed. However if the right argument is evaluated first (Fig. 4.7: path $1 \rightarrow 2b \rightarrow 3$) then we must copy the array as the left argument still refers to a. Since we are using non-strict languages we cannot perform all reads before writes as the computation must be demand driven to allow recursive definition.

4.5.1 Strategies

In order to avoid this efficiency loss we investigate:

1. The use of non-array datatypes which may be decomposed and their constituent parts shared, such as trees and lists.

2. Alternative mathematical methods which eschew the need for pivoting.

4.5.2 Non-Array-Based Matrix Methods

In this section we examine how we may efficiently express algorithms common to linear algebra if we no longer have efficient array access/replacement and are forced to use an alternative approach. This is the case for many functional languages as often if arrays are available they may not be efficient, or they may exhibit prohibitive access/update costs.

The structure which is most prevalent in functional programming is the list. Lists are very efficient for operations such as vector addition, matrix addition or scalar multiplication. These operations may be specified as

```
> vectoradd [Double] -> [Double] -> [Double]
> vectoradd []     []     = []
> vectoradd (x:xs) (y:ys) = (x+y) : vectoradd xs ys

> scalarmult :: Double -> [Double] -> [Double]
> scalarmult a xs = [a * x | x <- xs]

> matrixadd [[Double]] -> [[Double]] -> [[Double]]
> matrixadd []     []     = []
> matrixadd (x:xs) (y:ys) = (vectoradd x y) : matrixadd xs ys
```

or equivalently

```
> vectoradd = zipWith (+)
> matrixadd = zipWith vectoradd
```

However operations such as matrix multiplication are more difficult as, to use an analogy, *we work against the grain* of the representation, as can be seen in Fig. 4.8, and hence do not admit such an elegant algorithm and we must restructure the lists first via transpostition to work *with the grain*:

```
> mmult :: [[Double]] -> [[Double]] -> [[Double]]
> mmult a b = [[ sum [a_ik * b_kj | (a_ik,b_kj) <- zip a_i_ b__j]
>              | b__j <- b_T ]
>              | a_i_ <- a   ] where b_T = transpose b
```

This causes inefficiencies.

If an algorithm operates *with the grain* of a representation we say that it is *catamorphic*, meaning it descends the structure, and it is this property which makes many functional programs elegant. To solve a problem over a large set of data we combine solutions over subsets of that data.

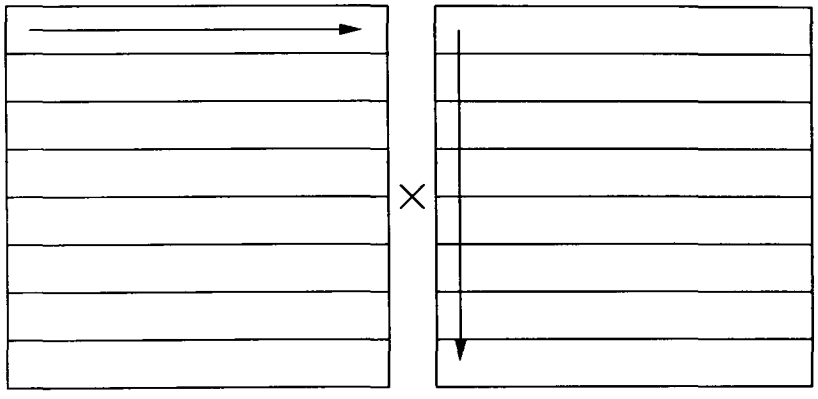


Figure 4.8: List-based matrix multiplication

4.5.2.1 LU Decomposition Over Lists of Lists

We may define LU decomposition (Doolittle) in this way using elimination:

```
> lud :: [[Double]] -> ([Double],[[Double]])
> lud [[x]] = ([],[[x]])
> lud ((pivot:row1)
>            :rows) = let
>                       (mults,submat) = eliminate pivot row1 rows
>                       (l,u)          = lud submat
>                     in
>                       (mults:l,(pivot:row1):u)
>
> eliminate :: Double -> [Double] -> [[Double]] -> ([Double],[[Double]])
> eliminate  pivot  row1 [] = ([],[])
> eliminate  pivot  row1 ((c1:row):rows)
>   = let
>       (multipliers,submatrix) = eliminate pivot row1 rows
>       m = c1 / pivot
>       updatedrow = axpy (-m) row1 row
>       axpy a x y = zipWith (\x_i y_i -> a * x_i + y_i) x y
>     in
>       (m:multipliers,updatedrow:submatrix)
```

This formulation proves to be quite efficient, as can be seen in Fig. 4.9. Unfortunately, if we wish to perform partial pivoting then we lose much of this elegant recursive definition as the columns of **L** must be dragged through the computation in case row-swapping is necessary (Fig. 4.10). This makes the problem monolithic. So although an efficient pivoting LU-decomposition is possible using lists of lists, much of the elegance that we desire is lost and we arrive at an algorithm which is very

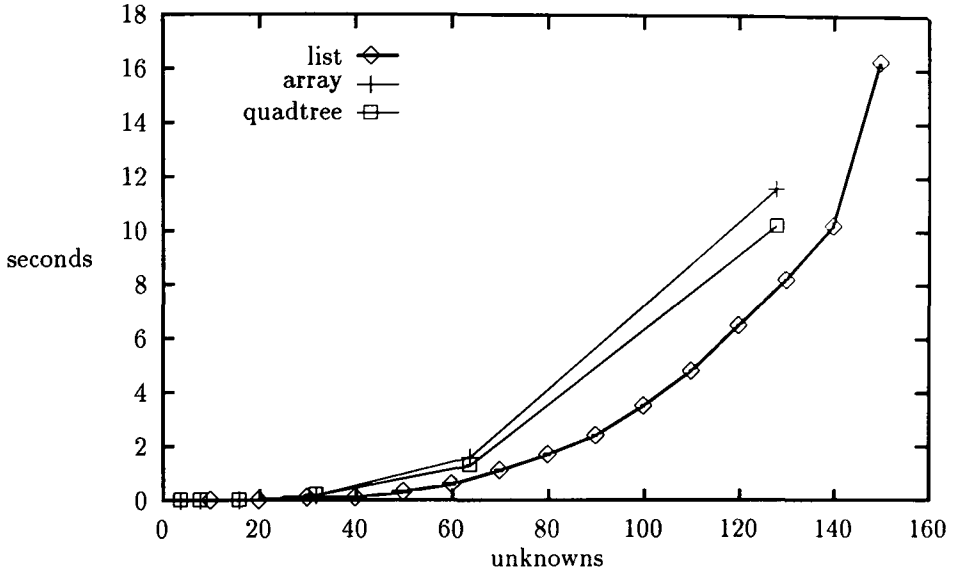


Figure 4.9: Timings (Unix time) for monolithic array, quadtree and list versions of LU factorisation

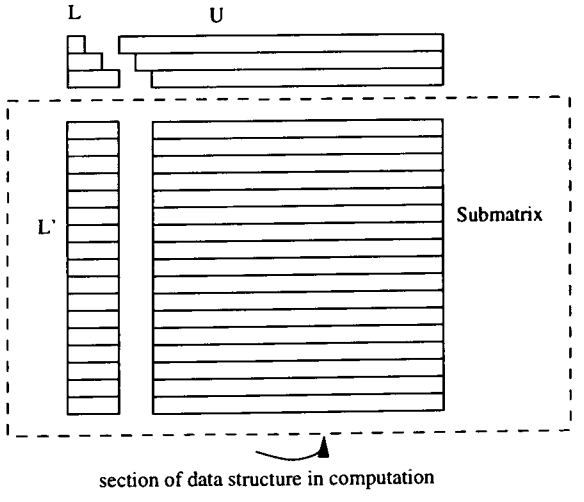


Figure 4.10: Monolithic list-based LU factorisation with partial pivoting

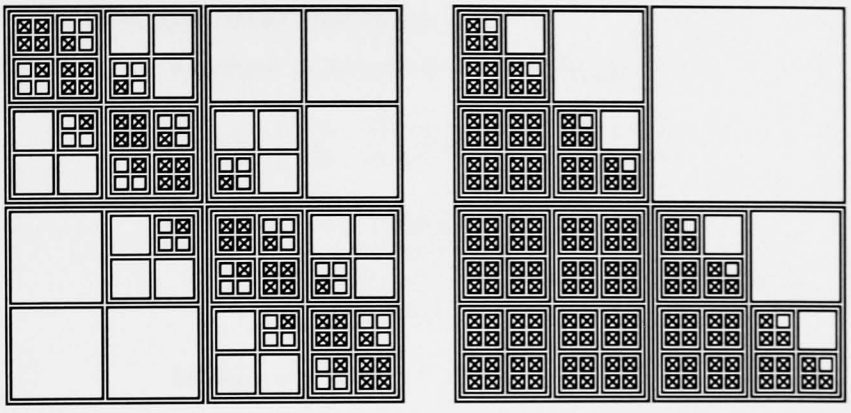


Figure 4.11: Tri-diagonal and lower-triangular matrices as quadtrees

hard to reason about.

4.5.2.2 Quadtrees

As mentioned earlier, functional languages generally shy away from the use of arrays and tend to use structures such as lists or trees instead, leading to the writing of algorithms in a very catamorphic way. Two structures that lend themselves to this area are the quadtree and binary tree, which can be used to represent matrices and vectors respectively. Examples of simple quadtree matrices are given in Fig. 4.11.

A matrix of order 2^p can be represented by a quadtree of depth p, where the tree is either:

- a zero matrix represented by 0;
- a scalar multiple aI ($a \neq 0$) of the identity matrix, represented by a;
- a 2×2 block matrix, each sub-matrix being of order 2^{p-1}.

Some of the advantages of using quadtrees are that they are useful for both sparse and dense storage patterns; their asymptotic space complexity is linear in the number of non-zeros for common matrices [106][8]; and matrix algorithms can be expressed naturally as block algorithms using quadtrees.

[8] It is possible to construct pathological matrix patterns that refute this, such as permutation matrices. However these can be stored as Ahnentafel index trees, as in [106].

4.5.2.3 LU Decomposition Over Quadtrees

An LU factorisation can be expressed using a block formulation[9] as

$$\text{blockLU} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} LU_{11} & U_{12} \\ L_{21} & LU_{22} \end{bmatrix}$$

$$\begin{aligned} \text{where} \quad & LU_{11} = \text{blockLU } A_{11} \\ & U_{12} = \text{fwdSubst } LU_{11} A_{12} \\ & L_{21} = \text{bkSubst } LU_{11} A_{21} \\ & LU_{22} = \text{blockLU } (A_{22} - L_{21} U_{12}) \end{aligned}$$

$$\text{blockLU } a = a$$

where we pattern match on the tree to determine whether decomposition is necessary. This presentation encourages inductive proof techniques and the correctness of the above algorithm may be reduced to proving the following:

$$\begin{aligned} L_{11}(\text{fwdSubst } LU_{11} A_{12}) &= A_{12} \\ (\text{bkSubst } LU_{11} A_{21}) U_{11} &= A_{21} \end{aligned}$$

where fwdSubst and bkSubst are defined as

$$\begin{aligned} (\text{fwdSubst } (L + U)\, A &= L^{-1} A \\ (\text{bkSubst } (L + U)\, A &= A U^{-1} \end{aligned}$$

4.5.2.4 Efficiency

The efficiency of this method on full matrices measures up well against array and list representations (Fig. 4.9), although the great advantage of using quadtrees to represent matrices lies in their ability to represent dense and sparse matrices uniformly with operations over quadtrees taking advantage of sparsity. In addition to this, the cost of representing common sparse matrices is favourable, as can be seen in Table 4.1, taken from [105]. The metric *Density* refers to the ratio between the space occupied by a matrix and the space occupied by a dense matrix of the same order, and the metric *Sparsity* refers to one minus the ratio between the expected time to access a random element and the expected access time within a dense matrix of the same order. Both metrics are measured on a scale from zero to one, with Density and Sparsity being accurate to within a term of $O(n^{-1})$ and $O((\lg(n))^{-2})$ meaning that some densities are recorded as zero. The expected path is the expected time to access a random element. Matrices which measure up badly are the FFT and Shuffle permutation matrices which, as we show later, may be represented more efficiently.

[9] This factorisation can be considered to be a version of the Fadeev algorithm [28].

4.5.2.5 Quadtree Cyclic Reduction

A quadtree version of *cyclic reduction* is also possible, although the derivation of this is not obvious from the usual elimination step definition:

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1} \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \beta_i a_{i+1} \\ c_i^{(1)} &= \beta_i c_{i+1} \\ y_i^{(1)} &= y_i + \alpha_i y_{i-1} + \beta_i y_{i+1} \\ \alpha_i &= -a_i/b_{i-1} \\ \beta_i &= -c_i/b_{i+1} \end{aligned}$$

The quadtree version may be expressed as

$$\text{cyclic}(a, y) = y/a \qquad \text{if } a \text{ and } y \text{ are scalars}$$

$$\text{cyclic}(A, y) = \begin{bmatrix} fv_1 \\ v_2 \end{bmatrix} \text{ where } \begin{array}{l} v_1 = \text{cyclic}(A_1, y_1) \\ v_2 = \text{cyclic}(A_2, y_2) \\ \begin{bmatrix} A_1 & - \\ - & A_2 \end{bmatrix} = K[M(\hat{L} + \hat{U}) + \hat{D}]K^T \\ \\ \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = K(My + y) \\ M = -(\hat{L} + \hat{U})\hat{D}^{-1} \\ \\ (\hat{D}, (\hat{L} + \hat{U})) = \text{takeOutDiag}(A) \end{array}$$

where the Ks are matrices of *knight's moves*of any size:

$$K = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

and A_1, A_2 are half the order of A. Obviously these permutation matrices should not be stored explicitly and we can construct functions $\text{shuffle}(x) = Kx$ and $\text{shuffle}^M(A) = KAK^T$ which serve the same purpose. The following functions are defined via pattern matching where we take some liberty[10] with syntax for clarity. We consider the binary tree in Haskell

```
data BinTree a= Leaf a | Branch (BinTree a) (BinTree a)
```

[10] In Part III of this thesis we discuss how this syntactic liberty may be realised.

as being defined as

$$\texttt{data BinTree a} = \texttt{a} \mid \begin{bmatrix} \texttt{(BinTree a)} \\ \texttt{(BinTree a)} \end{bmatrix}$$

and the quadtree

```
data Quadtree a= QLeaf a | QBranch (Quadtree a) (Quadtree a)
                                   (Quadtree a) (Quadtree a)
```

as being defined as

$$\texttt{data Quadtree a} = \texttt{a} \mid \begin{bmatrix} \texttt{(Quadtree a)} & \texttt{(Quadtree a)} \\ \texttt{(Quadtree a)} & \texttt{(Quadtree a)} \end{bmatrix}$$

$$\text{shuffle} \quad \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \text{split} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$\text{shuffle} \quad x = x$$

$$\text{split} \quad \begin{bmatrix} \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} \\ \begin{bmatrix} u_2 \\ v_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} \\ \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} \end{bmatrix} \quad \text{where} \begin{bmatrix} o_i \\ e_i \end{bmatrix} = \text{split} \begin{bmatrix} u_i \\ v_i \end{bmatrix}$$

$$\text{split} \quad x = x$$

$$\text{shuffle}^M \quad \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \text{split}^M \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$\text{shuffle}^M \quad x = x$$

$$\text{split}^M \quad \begin{bmatrix} \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} & \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} \\ \begin{bmatrix} a_3 & b_3 \\ c_3 & d_3 \end{bmatrix} & \begin{bmatrix} a_4 & b_4 \\ c_4 & d_4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} nw_1 & nw_2 \\ nw_3 & nw_4 \end{bmatrix} & \begin{bmatrix} ne_1 & ne_2 \\ ne_3 & ne_4 \end{bmatrix} \\ \begin{bmatrix} sw_1 & sw_2 \\ sw_3 & sw_4 \end{bmatrix} & \begin{bmatrix} se_1 & se_2 \\ se_3 & se_4 \end{bmatrix} \end{bmatrix}$$

$$\text{where} \begin{bmatrix} nw_i & ne_i \\ sw_i & se_i \end{bmatrix} = \text{split}^M \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix}$$

$$\text{split}^M \quad x = x$$

4.5.2.6 Algorithms Over Quadtrees

It has been suggested [42, 100] that quadtrees be used as an implementation for matrices in applicative languages. However, using this as an abstract datatype is prohibitive as we incur $O(\lg n)$ access times for element access. If the algorithms employ a *divide and conquer* style approach acting *catamorphically* over the quadtree then the situation is not so severe. For example, the addition

	Space	Density	Expected Path	Sparsity
Dense	$\frac{4}{3}(n^2 - \frac{1}{4})$	1	$\lg n + 1$	0
Symmetric	$\frac{2}{3}(n+2)(n-\frac{1}{2})$	$\frac{1}{2}$	$\lg n + 1$	0
Triangular	$\frac{2}{3}(n+2)(n-\frac{1}{2})$	$\frac{1}{2}$	$\frac{\lg n}{2} + \frac{3}{2} - \frac{1}{2n}$	$\frac{1}{2} - \frac{1}{\lg n}$
FFT permutation	$\frac{n \lg n}{2} + \frac{4n}{3} - \frac{1}{3}$	$\frac{3}{8}\frac{\lg n}{n}$	$\frac{\lg n}{2} + \frac{4}{3} - \frac{1}{3n}$	$\frac{1}{2} - \frac{5}{6 \lg n}$
Tridiagonal	$6n - 2\lg n - 5$	0	$\frac{10}{3} - \frac{3}{n} + \frac{2}{3n^2}$	$1 - \frac{10}{3 \lg n}$
Pentadiagonal	$8n - 2\lg n - 9$	0	$\frac{10}{3} - \frac{1}{n} - \frac{10}{3n^2}$	$1 - \frac{10}{3 \lg n}$
Heptadiagonal	$11n - 2\lg n - 19$	0	$\frac{10}{3} - \frac{5}{n} - \frac{76}{3n^2}$	$1 - \frac{10}{3 \lg n}$
Enneadiagonal	$13n - 2\lg n - 26$	0	$\frac{10}{3} - \frac{7}{n} - \frac{100}{3n^2}$	$1 - \frac{10}{3 \lg n}$
Shuffle permutation	$3(n-1)$	0	$3(1 - \frac{1}{n})$	$1 - \frac{3}{\lg n}$
Identity	1	0	1	$1 - \frac{1}{\lg n}$

Table 4.1: Average costs of quadtree representation

of quadtrees may be performed element-wise using an *element selector function*, or by recursively descending the structure. The time taken to add two quadtree matrices of order N catamorphically may be defined as follows:

$$\begin{aligned} T(N) &= 4\left[T\left(\tfrac{N}{2}\right)\right] + C_1 \\ T(1) &= C_2 \end{aligned}$$

where C_1 is associated with constructing a 4-element node and C_2 is associated with element addition. The complexity of the recursive scheme is as follows:

$$\begin{aligned} T(N) = T(2^k) &= 4\left[T\left(\tfrac{N}{2}\right)\right] + C_1 \\ &= 4\left[4\left[T\left(\tfrac{N}{4}\right)\right] + C_1\right] + C_1 \\ &\vdots \\ &= 4^k C_2 + C_1 \sum_{i=0}^{k} 4^i \\ &= 4^k C_2 + 4^{k+1} C_1 - C_1 \\ &= C_2 N^2 + 4 C_1 N^2 - C_1 \\ &= O(N^2) \end{aligned}$$

This complexity tends towards $O(N)$ as one or more of the matrices tends towards a diagonal matrix with no explicit description of the structure. If the structure is accessed element-wise then the complexity is $O(N^2 \lg N)$ since the cost of element access is $O(\lg N)$.

4.5.2.7 Quadtree SOR

This access-cost was largely ignored by Wainwright and Sexton in their study of sparse matrix representations for the solving of linear systems in functional languages [100]. In their study they conclude (as expected) that a quadtree representation is more suited to techniques such as the conjugate gradient method and less suited to methods such as SOR. We agree with these findings only to a certain extent as Wainwright and Sexton's formulations of each of these methods are very

different. Their conjugate gradient method concentrates on level-2 BLAS (Matrix-vector) operations and hence suits the quadtree representation. However, their SOR formulation concentrates on level-1 BLAS operations, updates and accesses, stamping a row-wise formulation on the quadtree. It is, however, straightforward to create a version of SOR suitable to quadtrees starting with the definition in [101] (p. 60):

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \left(\omega^{-1}\mathbf{D}+\mathbf{L}\right)^{-1}\left[\mathbf{b}-\mathbf{U}\mathbf{x}^{(k)}-\left(1-\omega^{-1}\right)\mathbf{D}\mathbf{x}^{(k)}\right] \\ &= \left[\mathbf{I}-\left(\omega^{-1}\mathbf{D}+\mathbf{L}\right)^{-1}\mathbf{A}\right]\mathbf{x}^{(k)}+\left(\omega^{-1}\mathbf{D}+\mathbf{L}\right)^{-1}\mathbf{b} \\ &= \mathbf{x}^{(k)}+\left(\omega^{-1}\mathbf{D}+\mathbf{L}\right)^{-1}\left[\mathbf{b}-\mathbf{A}\mathbf{x}^{(k)}\right]\end{aligned}$$

Although both methods are $O(N^2)$ (full matrix), imposition of a row-wise decomposition causes the row-oriented version to be very expensive, suggesting that it is the lack of regard for the data structure, rather than the inherent unsuitability of the quadtree to SOR. For a tightly-banded matrix, such as tridiagonal, this *mining* of rows and $O(\lg N)$ element access/replacement causes the row-wise algorithm to exhibit $O(N \lg N)$ behaviour rather than $O(N)$.

A quadtree version of SOR whose behaviour tends towards $O(N)$ as the bandwidth of A tends towards zero is given below[11]:

$$\begin{aligned}&\text{quadStep } A\ x\ b = x+\epsilon \\ &\quad \textit{where } \epsilon = \textit{SORsolve}\ \omega\ A\ r \\ &\qquad r = b - Ax \\ &\qquad \omega = 1.66\end{aligned}$$

where *SORSolve* is defined as

$$\begin{aligned}&\textit{SORsolve}\ \omega \begin{bmatrix} A_{11} & \\ A_{21} & A_{22}\end{bmatrix} \begin{bmatrix} b_1 \\ b_2\end{bmatrix} = \begin{bmatrix} \textit{SORsolve}\ \omega\ A_{11}\ b_1 \\ \textit{SORsolve}\ \omega\ A_{22}\ (b_2 - A_{21}x_1)\end{bmatrix} \\ &\textit{SORsolve}\ \omega\ a\ b = \omega b/a\end{aligned}$$

This algorithm was tested against the row-oriented SOR of Wainwright and Sexton under Hugs(reduction+cells used) and GHC (unix time), the results being summarised in Table 4.2. The results show that this catamorphic formulation executes at between 25% and 300% faster than that of Wainwright and Sexton (a motivating argument for computing along the grain). This signifys the importance of respecting the datastructure's underlying representation.

[11] A value of $\omega = 1.66$ is shown although any value in the range $[1, 2]$ is permissable.

N	Matrix Type	row-oriented method	block method
512 (HUGS)	Full	4,554,169(10,049,620)	3,598,500(8,256,506)
1024 (HUGS)	Full	18,000,057(39,719,793)	14,362,276(32,940,531)
512 (HUGS)	Tridiagonal	259,485(575,447)	51,820(130,030)
1024 (HUGS)	Tridiagonal	573,328(1,267,839)	104,008(259,996)
2048 (HUGS)	Tridiagonal	1,255,301(2,769,508)	208,428(520,010)
4096 (HUGS)	Tridiagonal	(heap exhausted)	417,280(1,040,120)
8192 (HUGS)	Tridiagonal	(heap exhausted)	835,044(2,080,422)
8192 (GHC -O)	Tridiagonal	7.3	1.8

Table 4.2: Row and block oriented SOR results

4.5.2.8 Sparse Strassen Multiplication

By using a quadtree it is trivial to implement a version of Strassen's algorithm [89] for matrix multiplication which automatically respects the sparseness of the matrices.

$$
\begin{array}{rcl}
0 & B & = 0 \\
A & 0 & = 0 \\
\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} & \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} & = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}
\end{array}
$$

where

$$
\begin{array}{rcl}
C_{11} & = & P_1 + P_4 - P_5 + P_7 \\
C_{12} & = & P_3 + P_5 \\
C_{21} & = & P_2 + P_4 \\
C_{22} & = & P_1 + P_3 - P_2 + P_6 \\
P_1 & = & (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 & = & (A_{21} + A_{22})B_{11} \\
P_3 & = & A_{11}(B_{12} - B_{22}) \\
P_4 & = & A_{22}(B_{21} - B_{11}) \\
P_5 & = & (A_{11} + A_{12})B_{22} \\
P_6 & = & (A_{21} - A_{11})(B_{11} + B_{12}) \\
P_7 & = & (A_{12} - A_{22})(B_{21} + B_{22})
\end{array}
$$

$$
\begin{array}{rcl}
a & \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} & = \begin{bmatrix} aB_{11} & aB_{12} \\ aB_{21} & aB_{22} \end{bmatrix} \\
\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} & b & = \begin{bmatrix} A_{11}b & A_{12}b \\ A_{21}b & A_{22}b \end{bmatrix} \\
a & b & = ab
\end{array}
$$

4.5.2.9 Pivoting

As with nearly all quadtree algorithms, the formulations above automatically take advantage of sparsity, decompose naturally into a few subproblems, highlight implicit parallelism inherent in the

method[12], and make reasoning and proving programs correct much easier. However, since they do not decompose naturally (or efficiently) into rows or columns, additions to algorithms such as pivoting are hard (if not impossible) to implement efficiently[13].

4.5.2.10 Summary

It can be seen from the algorithms above that quadtrees have many useful applications, such as sparse matrix representations and tools for divide and conquer algorithms. In writing quadtree methods it is clear that very similar subfunctions regularly occur. These functions are variations on the BLAS and therefore producing a library of quadtree BLAS would be very beneficial to this area. Unfortunately, many non-block algorithms do not admit an elegant (or efficient) quadtree formulation and so arrays are still very desirable.

4.5.3 Alternatives to Pivoting

As mentioned earlier, the big disadvantage of using a quadtree LU factorisation scheme, or a compact scheme, is that it cannot deal with the need for pivoting, i.e. it breaks down on problems such as

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \mathbf{x} = \mathbf{b}$$

In this section we examine what occurs during pivoting and attempt to create an alternative factorisation algorithm which eschews pivoting and which will not break down on problems such as the above. For the compact scheme

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \Big/ u_{jj} \quad (i > j)$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad (i \leq j)$$

method breakdown is avoided if each leading submatrix of A is nonsingular. If a factorisation involves pivoting we do not factor **A** into **LU** but instead factor **PAQ** into **LU** such that

$$\mathbf{PAQ}(\mathbf{Q}^T\mathbf{x}) = \mathbf{Pb}$$

where P and Q are permutation matrices generated and implicitly applied during the method execution using intermediate values. One possibility would be to permute the matrix A in advance but this would involve as much computation as pivoting and so we reject this. Another possibility is artificially to impose non-singularity on each leading sub-matrix and deal with the resulting consequences later. In both the compact and Fadeev (quadtree) methods a singularity in the i^{th}

[12] In our quadtree LU decomposition the North-East and South-West quadrants of the matrix may be built in parallel and many of the linear algebra building blocks are implicitly parallel.

[13] An LU-style factorisation over quadtrees has been developed by D.S. Wise based on *undulant block pivoting* [106], although this formulation is very involved and not easily reasoned about.

leading submatrix is discovered at the point where the element u_{ii} is determined. If the value of u_{ii} is zero then the i^{th} leading submatrix is singular, otherwise the leading submatrix is nonsingular. The method we suggest is that on discovering a singularity at the i^{th} stage we take the value of u_{ii} as one, rather than zero, and continue. This ensures each leading submatrix is nonsingular.

4.5.3.1 Proof

- For the case $n = 1$ any nonzero constant is a nonsingular 1×1 matrix.
- For the inductive case (assume true for $n-1$): The first $n-1$ rows of U_n are linearly independent (from hypothesis) and so the first $n-1$ columns may be triangularised via Gaussian elimination. If U_n is nonsingular we are done, otherwise we add the vector $[0, 0, \ldots, 0, 1]^T$ to the final column of U_n. As this forms a triangular matrix with non-zero diagonal elements it is necessarily nonsingular.

Here U_i refers to the i^{th} leading submatrix. We thus have the factorisation

$$\mathbf{LU} = \mathbf{A} + \mathbf{D}$$

where $\mathbf{D}$ is a diagonal matrix of ones and zeros. To obtain the solution to $\mathbf{Ax} = \mathbf{b}$ we need a method of solving $(\mathbf{LU} - \mathbf{D})\mathbf{x} = \mathbf{b}$. A corollary of the above proof is that if $\mathbf{A}$ is nonsingular then so is $\mathbf{A} + \mathbf{D}$.

4.5.3.2 Rank Annihilation

We present a method of solving $\mathbf{Ax} = \mathbf{b}$ based around *Rank Annihilation* [102, 103]. From the *Sherman-Morrison* formula [79] (p. 77) which gives the inverse of $\mathbf{A} + \mathbf{u} \otimes \mathbf{v}$ for arbitrary vectors $\mathbf{u}, \mathbf{v}$:

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1} = \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1}\mathbf{u}) \otimes (\mathbf{v}^T\mathbf{A}^{-1})}{1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}}$$

we may derive the solution of

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})\mathbf{x} = \mathbf{b}$$

as

$$\mathbf{x} = \mathbf{y} - \left(\frac{\mathbf{v}^T\mathbf{y}}{1 + \mathbf{v}^T\mathbf{z}}\right)\mathbf{z}$$

where

$$\mathbf{Ay} = \mathbf{b} \qquad \mathbf{Az} = \mathbf{u}$$

and from this derive a formulation of LU factorisation.

4.5.3.3 Rank annihilating LU factorisation

Consider the linear system

$$\mathbf{Ax} = \mathbf{b}$$

This may be rewritten as

$$((\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) - \mathbf{u} \otimes \mathbf{v})\mathbf{x} = \mathbf{b}$$

and may be solved via

$$\mathbf{x} = \mathbf{y} + \left(\frac{\mathbf{v}^T\mathbf{y}}{1 - \mathbf{v}^T\mathbf{z}}\right)\mathbf{z}$$

where

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})\mathbf{y} = \mathbf{b} \qquad (\mathbf{A} + \mathbf{u} \otimes \mathbf{v})\mathbf{z} = \mathbf{u}$$

More interestingly, if we write $\mathbf{Ax} = \mathbf{b}$ as

$$\left(\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{u}_k\right) - \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{u}_k\right)\mathbf{x} = \mathbf{b}$$

we may solve $\mathbf{Ax} = \mathbf{b}$ via

$$\mathbf{x} = \mathbf{y} + \mathbf{Zw}$$

where

$$\begin{aligned}
\mathbf{w} &= \mathbf{H}^{-1}\mathbf{v} \\
\mathbf{v} &= \mathbf{U}^T\mathbf{y} \\
\mathbf{H} &= [\mathbf{I} - \mathbf{U}^T\mathbf{Z}] \\
\mathbf{y} &= \left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{u}_k\right)^{-1} \mathbf{b} \\
\mathbf{z}_i &= \left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{u}_k\right)^{-1} \mathbf{u}_i \\
\mathbf{Z} &\equiv [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_P] \\
\mathbf{U} &\equiv [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_P]
\end{aligned}$$

$\mathbf{Z}$ and $\mathbf{U}$ being matrices with columns $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_P$ and $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_P$. Therefore we may avoid method breakdown in LU factorisation without pivoting by slightly altering our normal *Doolittle (Black)* formulation

$$\begin{aligned}
l_{ij} &= \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) \Big/ u_{jj} \qquad (i > j) \\
u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \qquad (i \leq j)
\end{aligned}$$

To encompass this change we have

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \Big/ (\text{if } u_{jj} = 0 \text{ then } 1 \text{ else } u_{jj}) \quad (i > j)$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad (i \leq j)$$

which defines

$$(\mathbf{L} - \mathbf{I}) + (\mathbf{U} - \mathbf{D})$$

where

$$\mathbf{LU} = \mathbf{A} + \mathbf{D}$$

and D is a diagonal matrix with d_{ii} being either 1 or 0:

$$d_{ii} = \begin{cases} 1 & \text{if } u_{ii} = 0 \\ 0 & \text{otherwise} \end{cases}$$

From this we may easily solve $\mathbf{Ax} = (\mathbf{LU} - \mathbf{D})\mathbf{x} = \mathbf{b}$ via application of the *Sherman-Morrison-Woodbury* method.

4.5.3.4 Convergence

As it stands this method may not converge as there is no guarantee that the set of vectors $\{\mathbf{u}_i\}$ that the $(N \times N)$ matrix A produces will have cardinality $< N$, although this will be extremely rare. A solution to this problem would be initially to find a row with a non-zero first element and consider this as the first row in the matrix performing the decomposition under a row-wise permutation of which we have *a-priori* knowledge.

$$l_{ij} = \left(a_{p(i)j} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \Big/ (\text{if } u_{jj} = 0 \text{ then } 1 \text{ else } u_{jj}) \quad (i > j)$$

$$u_{ij} = a_{p(i)j} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad (i \leq j)$$

$$p(i) = i \quad \text{if } a_{11} \neq 0$$
$$p(i) = 1 \quad \text{if } i = piv$$
$$p(i) = piv \quad \text{if } i = 1$$
$$p(i) = i \quad \text{otherwise}$$
$$piv = \text{head}[j | a_{j1} \neq 0]$$

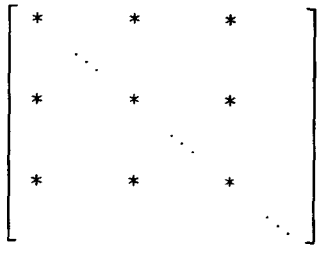


Figure 4.12: Patterning of matrix H under quadtree-Sherman-Morrison-Woodbury method

4.5.4 Combination Approaches

This method of rank annihilation may also be used with quadtrees where suitable changes are made to `bkSubst`, i.e. a division by zero implies division by 1.

$$\text{blockLU} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} LU_{11} & U_{12} \\ L_{21} & LU_{22} \end{bmatrix}$$

$$\begin{aligned} \text{where} \quad & LU_{11} = \text{blockLU } A_{11} \\ & U_{12} = \text{fwdSubst } LU_{11} A_{12} \\ & L_{21} = \text{bkSubst } LU_{11} A_{21} \\ & LU_{22} = \text{blockLU } (A_{22} - L_{21} U_{12}) \end{aligned}$$

$$\text{blockLU } a = a$$

However, since the previous formulation (Section 4.5.3.) constructed matrices in a row-wise manner, it does not suit the quadtree representation particularly well. If quadtrees are to be used with this method, it is suggested that square matrices are used, relying on the performance of the quadtree as a sparse matrix data structure. That is, the diagonal matrix **D** is explicitly formed as a sparse matrix and block methods are used in the Sherman-Morrison-Woodbury method. A consequence of this is that we solve an $N \times N$ linear system at each stage, although the patterning of the matrices (Fig. 4.12) is such that no *fill-in* occurs under LU-factorisation and therefore, due to the advantage quadtrees take of sparsity, the amount of extra work which needs to be performed is low. To gain the full benefit from the quadtree representation it is advisable to add a specific multiplicative identity I to avoid unnecessary multiplications.

4.6 Summary

The methods presented above demonstrate that in many cases functional specifications of numerical algorithms are very elegant. Unfortunately, it is often the case that the speed of execution of these

algorithms is unacceptably slow[14]. In array-based algorithms this speed loss can often be attributed to the copying of data during updates. To side-step this problem we presented two strategies:

1. Non-array based methods
2. Alternative mathematical formulations

and demonstrated via the SOR method that if non-array-based methods were to be used then they should be used catamorphically. In this area it is the quadtree representation for which the most elegant, catamorphic algorithms exist. These formulations almost always expose any coarse-grain parallelism and respect sparsity implicitly. However, many common row/column-style algorithms are either impossible or too costly in practice to implement in terms of these structures.

Alternative mathematical formulations were used to attempt to eschew the need for pivoting. This produced a method which suits functional languages but which is still inefficient, in terms of the amount of necessary work, compared to an imperative pivoting array version. However, this process did produce a method which could be used with sparse matrices, does not introduce fill-in, and is more parallelisable than pivoting.

Unfortunately these methods did not produce an efficient, more aesthetically pleasing or more easily reasoned about method than the incremental update elimination version, suggesting we turn our attention towards exploring what (if anything) may be altered so that such definitions may be efficiently executed.

4.6.1 Array Semantics

Although we have shown that non-array-based methods are indeed possible, the most natural way to express a method, such as a pivoting factorisation, is often by incrementally updating array elements, necessitating strict semantics (Section 4.5). Unfortunately, the main advantage of non-strict semantics is the ability to consume structures before they are complete, and hence define data structures in terms of themselves. For example, back substitution can be defined as

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}, \qquad i = 1, 2, \ldots, n$$

and expressed in Haskell as

```
>    back_subst :: Matrix -> Vector -> Vector
>    back_subst u b = x
>     where x = array bnds [(b!i - f i)/u!(i,i)|i<- range bnds]
>           f i = sum[u!(i,j)*x!j|j <- range (i+1,n)]
>           bnds@(1,n) = bounds b
```

[14] By "unacceptably slow" we mean that it exhibits a worse asymptotic time complexity than its imperative counterpart.

where the vector x is defined in terms of itself.

However, we may define back substitution very elegantly using a quadtree approach. i.e.

$$\text{bkSub} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{aligned} \text{where} \quad x_2 &= \text{bkSub}\, U_{22} b_2 \\ x_1 &= \text{bkSub}\, U_{11}(b_1 - U_{12} x_2) \end{aligned}$$

$$\text{bkSub}\, ub = b/u$$

which is pictorially expressive, terminates under eager evaluation, and is more efficient. Therefore, before decisions on possible language alterations are to be considered the amount to which language features are used, and the areas which contribute to efficiency loss, should be examined, and hence it is these areas which are the subject of the next chapter.

Chapter 5

Efficiency and Empirical Analysis

In the previous chapter we used Haskell to implement numerical methods and concentrated on the relative efficiency and elegance of various techniques and formulations rather than the absolute efficiency of the resulting executables compared with FORTRAN or C.

5.1 Chapter Overview

In this chapter we take the most efficient methods from Chapter 4 and benchmark these against their imperative cousins.

5.1.0.1 White Box Analysis

Chapter 4 can be considered as treating Haskell implementations in a *black-box* style, attempting to gain the best performance with little regard for what was occurring *under the hood*. To continue the use of testing terminology we analyse the implementation in a *white-box* style to highlight the area where efficiency is lost.

5.1.0.2 Empirical Analysis

After highlighting the major areas of efficiency-loss we study how these language features, or idioms, are used in practice. This follows the simple idea that launched the RISC revolution in computer architecture: to obtain high performance, one must measure the behaviour of "real" programs, and make sure that the most common operations are performed at blinding speed - even if less common operations go a bit slower as a result [77]. This quantitative analysis is very important as a designer's "seat of the pants" instinct about what counts is often wrong [59].

5.1.0.3 Collation of Results

After empirically studying the extent and manner to which language features are used we collate this information to lay out a set of requirements that a functional language tailored to scientific computing

should satisfy, and suggest directions of investigation with the view to improving efficiency.

5.2 Efficiency Issues

Lazy evaluation in languages such as Haskell has advantages in areas such as heuristic searching, allowing elegant, recursive specifications, and β-reduction is always valid. Unfortunately lazy languages also have the property of slowing down the execution of work which is necessary by delaying it if the compiler fails to determine that the work *is* necessary. This has a side-effect of creating a situation where the order of evaluation has little to do with the *loop-like* higher order functions. and closures persist longer than necessary.

5.2.0.4 Benchmarks

As a simple benchmark of raw power we use the following arbitrary computation

$$\text{value} = \sum_{i=1}^{10,000} \left(\sum_{j=1}^{10,000} ij \right)$$

which we express in Haskell$^{(i)}$ as

```
> value :: Double
> value = sum[sum[fromInt i * fromInt j|j <-[1..10000]]|i<- [1..10000]]
```

We convert from integers since this more closely models common numerical computations and avoids overflow; in all versions we perform similar castings. We express this in C as

```
register int i,j;
register double value=0.0;
for (i=1;i<=10000;i++)
     {
      for (j=1;j<=10000;j++)
           {
            value += (double) i * (double) j;
           }
            /* end of for loop index = j */
     }
     /* end of for loop index = i */
```

and denote this as RC (readable C). We also optimise this and denote this as OC (obfuscated C) leaving it up to the compiler (gcc -O4) to perform more obvious optimisations such as loop unrolling and invariant removal[1]. In OC we "hack" the RC to remove relational operators and rely on the fact that a predecrement is generally faster than post decrement ([13] p.154).

[1] Unrolling and restructuring RC did not improve the performance of the compiler-optimised code.

```
for (i=10001;--i;)
            {
            for (j=10001;--j;)
                 {
                   value += (double) i * (double) j;
                 }
                 /* end of for loop index = j */
            }
            /* end of for loop index = i */
```

5.2.0.5 Timings

To print *value* the times were as shown in Table 5.1. From these timings we see that the Haskell

Code	RC(in code timing)	OC(in code timing)	Haskell(unix time)
Time (Seconds)	8.32	7.55	71.43

Table 5.1: Benchmark timings for computation

version executes 9.46 times slower than OC and 8.59 times slower than RC! If functional languages are to be used in this area then we must at least produce codes comparable to the efficiency of RC. That is, we must increase their efficiency by an order of magnitude.

5.2.1 Functional Programming and Arithmetic

When non-strict languages such as Haskell are compiled, the resulting code usually manipulates heap-allocated *box* numbers [78]. Thus, in a naive implementation, numbers are always represented as a pointer to a heap-allocated object which may be an unevaluated closure, or a *box*, containing the number's value which has overwritten the closure. Consequently, simple arithmetic operations which require a single machine instruction in a strict language require the following steps:

- fetch operands from their respective boxes
- perform the operation
- allocate a new box
- place the result in the box

Clearly it is more efficient to work with the bit patterns that reside in the boxes (*the unboxed values*) than with the boxes and contents (*the boxed values*). *With reference to large-scale scientific computing, retaining this efficiency is not only desirable but essential.*

Some compilers such as GHC allow the programmer access to these unboxed representations but at the expense of losing common higher-order functions and polymorphism! This effect is reduced, however, by the use of specialisation pragmas in the code. Other features, such as strictness annotations, also exist in functional languages (Haskell 1.3 definition) and, up to a point, these

features help to claw back efficiency. For instance, the performance of Level 1 BLAS operations are improved eight-fold when written in unboxed core Haskell.

5.2.1.1 Unboxed Haskell

In Glasgow Haskell (ghc), unboxed values can be accessed using hash-notation by considering the ground types (integers, doubles, floats, etc.) as being algebraic types with unary constructors, i.e. integers, doubles and floats can be thought of as being defined as

```
> data Int    = MkInt    Int#
> data Float  = MkFloat  Float#
> data Double = MkDouble Double#
```

For example `Int#` represents the unboxed integer (`int` in C). Unfortunately we cannot easily create structures of unboxed values since, when we use them, we lose polymorphism. This means we must redefine a monomorphic structure for each ground type.

5.2.1.2 Example: Boxed and Unboxed Lists

To find the lengths of a list of boxed integers and a list of boxed reals we could write

```
> data List a = Null | Cons a (List a)

> length :: List a -> Int
> length Null = 0
> length (Cons a as) = 1 + length as

> rlength = length rlist
> ilength = length ilist
```

for suitable lists, `ilist` and `rlist`. However, for unboxed lists we would need to write

```
> data FloatList = NullF | FCons Float# FloatList
> data IntList   = NullI | ICons Int#   IntList

> intlength :: IntList -> Int
> intlength xs = case xs of
>                    NullI       -> 0
>                    ICons x# xs -> 1 + (intlength xs)

> floatLength :: FloatList -> Int
> floatlength xs = case xs of
>                    NullI       -> 0
>                    ICons x# xs -> 1 + (floatlength xs)

> ilength = intlength iUnboxedList
```

```
> rlength = floatlength rUnboxedlist
```

To *really* improve performance however, we may split the function into a *worker* and *wrapper*[2]. unbox further and use tail recursion.

```
> intlength :: IntList -> Int
> intlength xs = case worker xs 0# of i# -> MkInt i#
>  where worker xs acc#
>     = case xs of
>                NullI -> 0#
>                ICons i# rest -> case plusInt# 1# acc# of
>                                        t# -> worker rest t#
```

only re-boxing into a boxed integer at the end of the function. This is the sort of optimisation that strictness analysis should allow. However, optimising code in the above fashion is usually performed by hand as automatic optimisation often fails to claw back this lost efficiency. It is felt that this form of source code obfuscation is really only acceptable in a library.

5.2.1.3 Scalar Functions

For scalar-valued functions the situation is not so bleak since, following [78], Haskell compilers can automatically lower the number of closures being built by using unboxed values. Unfortunately the case of non-scalar valued functions is not as simple.

5.2.1.4 Complex Numbers

Consider the complex numbers:

$$\begin{aligned} c_1 &= (re_1, im_1) \\ c_2 &= (re_2, im_2) \end{aligned}$$

Performing the addition $c_1 + c_2$ will cause closures to be built which, because of the non-strict semantics, will not be resolved until the real and imaginary parts of the tuple are demanded. This can be resolved by making the real and imaginary parts of the complex number strict so that no closures are built inside the tuple. That is, we regard complex numbers as atomic. Unfortunately, this approach cannot be extended since, if we consider a complex number as a point in the complex plane (a two-dimensional space) how should we represent points in three-, four- or n-dimensional space? If we take the above approach of strictifying each of the components we are in the position of being able to perform arithmetic more efficiently at these points but we lose the ability to consume part of the data structure before the complete structure is built and therefore cannot recursively define data structures. Later in this chapter we examine existing Haskell code for numerical methods to investigate whether this recursive definition is frequently used in practice.

[2]This worker/wrapper idiom is common in functional programming.

5.2.2 Efficiency Via Unboxing

To structure the use of unboxing we extend the Haskell 1.2 class hierarchy (Fig. 5.1) to admit a lower level. Unfortunately, this hierarchy does not allow efficient overloaded methods over structures

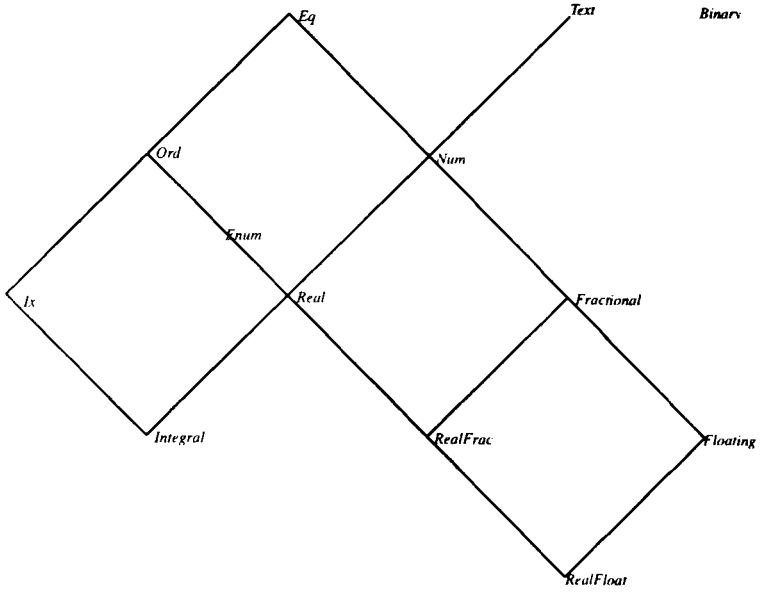


Figure 5.1: Existing Haskell 1.2 class hierarchy

such as vectors or matrices, and operations such as transposition do not fit into the hierarchy. With this in mind we extend the hierarchy to Fig. 5.2, where BLAS operations reside at the intersection of the classes whose member operations they depend on. (For instance a dot product would reside in Num-Structure as it operates over a structure and relies on the members of the class Num [+ and ×].) The advantage of operating in this manner is that we may provide efficient medium-to-large sized BLAS functions which have been internally unboxed with which to build algorithms. The Structure class contains methods which operate over structures such as transposition.

5.2.3 Performance

In practice this method of unboxing library internals dramatically improves performance.

To demonstrate this, codes taking a command line argument were written in naive boxed Haskell, unboxed Haskell using this class hierarchy, and in readable C (RC). For an argument n, the codes output the value of $x^T x$, where x is the vector $[1, 2, \ldots, n]^T$ of single-precision floating point numbers. The results in Fig. 5.3 exhibit approximately an eightfold increase in efficiency over the boxed representation although the amount of heap and stack space required for the boxed version suggests that it suffers from a major space leak as closures accumulate (seen as a departure from linear around 60,000). The boxed and unboxed Haskell versions were compared up to vector length 10^5. as

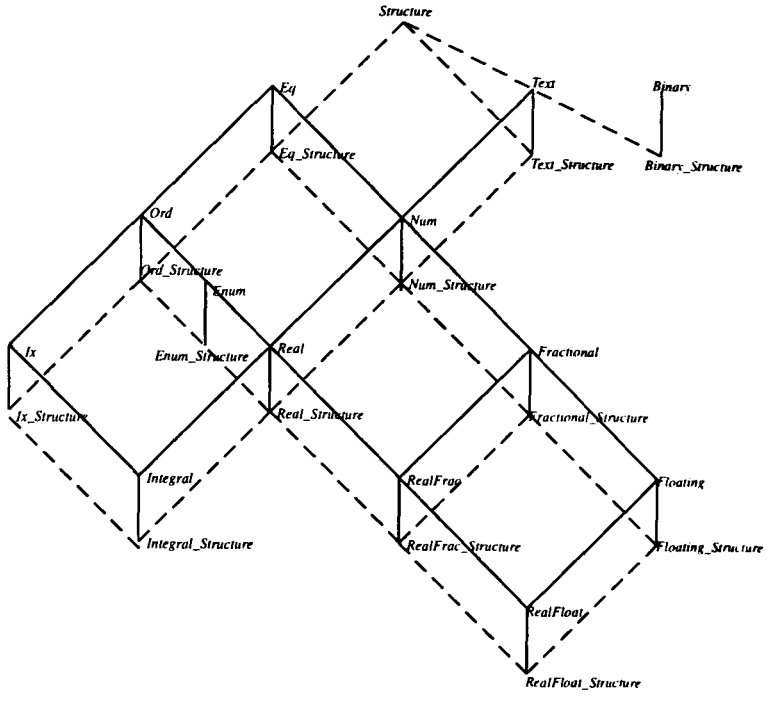


Figure 5.2: Extended class hierarchy

beyond this point the naive Haskell implementation consumes excessive heap and stack resources. (For vectors of length greater than 10^5 the boxed implementation fails with a 16Mb heap and a 16Mb stack!) The unboxed version was tested against an RC implementation up to a vector length of 10^6 (Fig 5.4) and ran at 1/4 of the speed of the corresponding RC implementation.

5.2.4 Larger Algorithms

A question which must be answered at this point is *does this method scale up and the benefit accrue, or does the efficiency diminish as the complexity of the algorithm increases?*

To test this, a conjugate gradient solver was constructed, using this extended class hierarchy, which solved a tridiagonal linear system arising from a finite difference solution to a one-dimensional Poisson equation. The number of unknowns was given as a command line argument and the implementations first constructed then solved the system. Heap profiles were examined to check that the solution of the system dominated the computation and that system assembly was insignificant compared to this.

For each Haskell version (Boxed and unboxed) the conjugate gradient method was coded as

```
>       iterativelySolve :: ([b] -> c) -> (a -> b) -> a -> c
>       iterativelySolve convergenceCondition nextIteration start
```

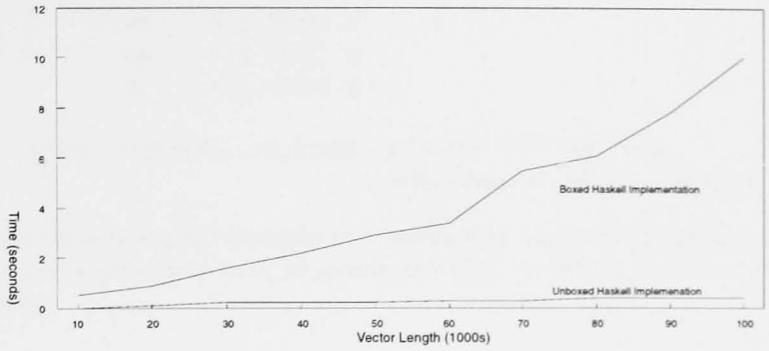


Figure 5.3: Times(unix) for boxed and unboxed dot products for vectors of length $10^4 - 10^5$

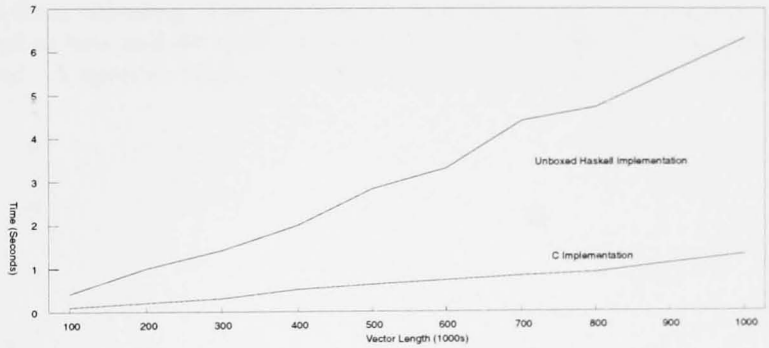


Figure 5.4: Times(unix) for unboxed and RC dot products of vectors of length $10^5 - 10^6$

```
>       = convergenceCondition (iterate nextIteration start)

>     conj_grad x0 eps a_times b
>      = iterativelySolve cgConverge cgIteration (x0,p0,r0)
>       where
>            p0 = b - a_times x0
>            r0 = p0

>            cgIteration (x,p,r) = (x',p',r')
>             where x'    = x  + alpha * p
>                   p'    = r' + beta  * p
>                   r'    = r  - alpha * q
>                   alpha = rr / pq
>                   beta  = (alpha * qq)/pq-1
>                   pq    = p `dot` q
```

```
>                   rr    = r `dot` r
>                   qq    = q `dot` q
>                   q     = a_times p

>             cgConverge((x,_,r):rest)| r `dot` r <= eps = x
>                                     | otherwise = cgConverge rest
```

with the extended class hierarchy enabling the overloading and multiplications. Note that this is *exactly* the code that appeared in both programs and that all boxing issues are relegated to inside instance definitions.

5.2.4.1 Results

It was found that under a binary tree/quadtree representation of vectors/matrices we could produce no improvement from unboxing. This is not surprising as the unary `Leaf` constructor acts as a box. For vectors stored as lists and the tridiagonal matrices stored as three lists the results shown in Fig. 5.5 were observed. A speedup factor of around 1.7 due to this unboxing was observed which was

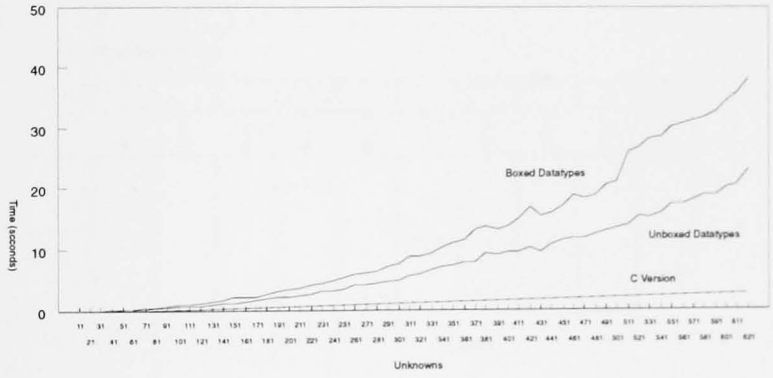


Figure 5.5: Times(unix) for unboxed/boxed Haskell and RC

disappointing in light of the earlier, simpler, dot product speedup.

5.2.4.2 Conclusion

The use of unboxing in BLAS subroutines improves the performance of algorithms. Unfortunately, as the complexity of these algorithms increases, the overhead of non-strict semantics creeps in and the efficiency is amortized against this. In view of this fact, and in view of the amount that unboxing obfuscates previously elegant Haskell, we reject this method as it does not satisfy our goal of elegance and only partially satisfies our goal of efficiency (we are still nowhere near the speed of RC or OC). The approach we choose is more holistic in that we aim to alter the language so that it efficiently

supports programming in the style of Chapter 4. We begin this process by highlighting areas of efficiency-loss as prime candidates for alteration.

5.3 Highlighting Areas of Inefficiency

Since the greatest proportion of imperative numerical algorithms involve array traversals, and since we argued that efficient array manipulation is required in a functional language specialised to numerical methods (Chapter 4), we begin by examining how $O(1)$ access arrays may be represented in functional languages.

5.3.0.3 Arrays in Functional Languages

Fig. 5.6 shows different storage schemes implied by different classes of language. These schemes are

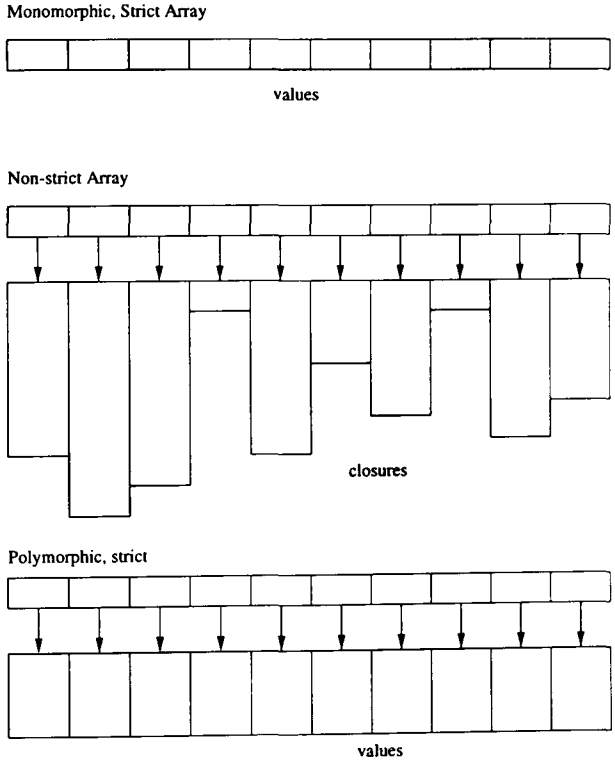


Figure 5.6: Array storage schemes

as follows (in order of decreasing efficiency):

- **Monomorphic, strict** arrays can be stored as a contiguous block of memory and hence map directly to arrays in FORTRAN or C. This is possible as the array is monomorphic and hence we have *a-priori* knowledge of the size of each element. However, since these arrays *are* monomorphic we may not apply useful higher-order polymorphic functions to them, such as *map* or *fold*, but must specialise these higher-order functions over each element type. Also, since these arrays are strict we may not define them recursively.

- **Polymorphic, strict** arrays require a uniform representation which may be used over all types. This implies that they are represented as arrays of pointers to heap-allocated elements. This scheme allows us to use higher-order uniformly but still does not allow recursive array definition.

- **Polymorphic, non-strict** arrays require that an array of closures be stored and, as these are represented as pointers to heap-allocated boxes, can be polymorphic at no extra cost. These arrays may be defined recursively but exhibit the efficiency loss associated with the manipulation of closures and boxed data. This is the variety of arry which Haskell supports.

Of these schemes, it is only monomorphic, strict which stands a chance of approaching the raw speed of optimised C or FORTRAN.

5.3.0.4 Polymorphism

If functions are polymorphic then they must either have a uniform calling convention so they can operate parametrically, regardless of type, or be tagged with their type as run-time data. Whilst being a great boon to the programmer, the uniformity exhibited in parametric polymorphism causes inefficiencies as we could generate more efficient code if monomorphic functions were used. The major advantage of monomorphic code is that a compiler can produce optimum code using optimum representations of data. That is, the code can be tailored to the type of the specialisation, producing the required runtime speed and low space overhead. The disadvantage of this technique is that the number of monomorphic specialisations can explode exponentially. We aim to investigate whether "real programs" in the area of numerical software explode in this manner or whether monomorphic specialisation may be vindicated.

5.3.0.5 Overloading

The charge of causing exponential code-growth levelled against the specialisation of polymorphism is also levelled against ad-hoc polymorphism (overloading) in languages such as Haskell. To avoid this exponential growth, Haskell implements instances of its type classes via implicit parameters. For instance, if we define addition over pairs of objects element-wise as

```
> instance (Num x,Num y) => Num (x,y) where
>   (a,b) + (c,d) = (a+c,b+d)
```

then Haskell translates this to

```
>   addPair :: (x->x->x) -> (y->y->y)->(x,y)->(x,y)->(x,y)
>   addPair plus_x plus_y (a,b) (c,d) = (plus_x a c,plus_y b d)
```

which, because of the cost associated with higher order functions, is much less efficient than if a monomorphic specialisation were constructed. Therefore, it would be beneficial to investigate whether monomorphic specialisation of overloaded functions would cause (numerical) program size to grow intractably.

5.3.0.6 Non-Strictness

As mentioned earlier in this chapter, non-strictness has a nontrivial implementation cost associated with it and therefore we should also examine the way non-strictness is used in practice.

5.3.0.7 Expressiveness

Fig. 5.7 shows the expressiveness, efficiency of evaluation and typing systems. By "more expressiveness" we mean that the translation of a program which uses non-strictness into a strict equivalent may require global re-organisation of the entire program.

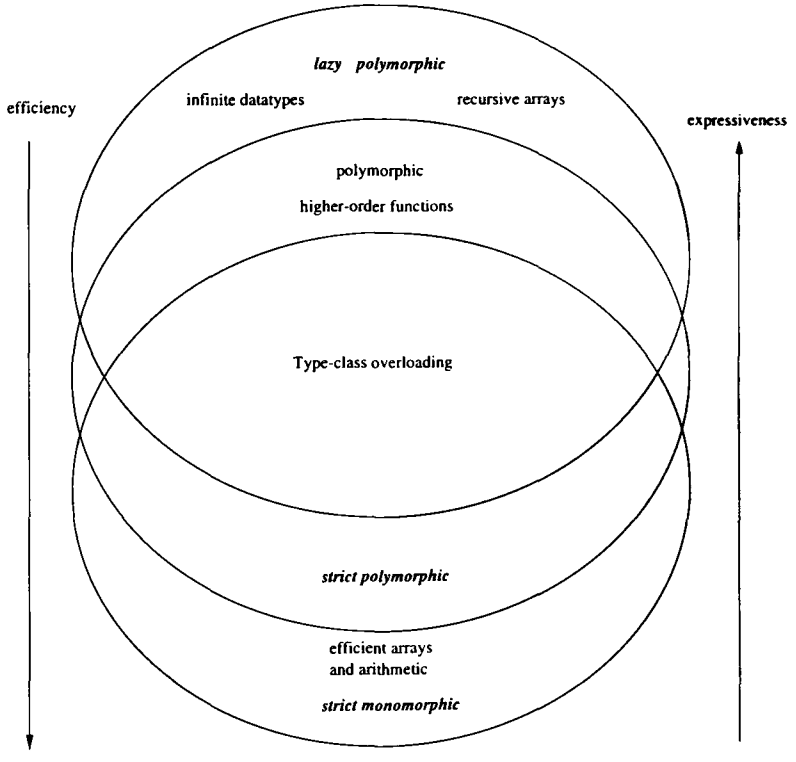


Figure 5.7: Intersection of desirable features

5.3.1 Forms of Non-Strictness

In this section we identify different forms of non-strictness and present examples to illustrate them. The forms and examples we use are taken directly from Schauser and Goldstein [85] so that their scientific computing results may be included in our results. In their study of non-strictness they only consider programs which do not manipulate potentially infinite data. We extend their results to cover infinite data also.

5.3.1.1 Functional Non-Strictness

Functional non-strictness arises from feedback dependencies from the result of a function invocation to its arguments. To illustrate this form of non-strictness and the need for dynamic scheduling, we use the following simple, contrived, example taken from [84]:

```
>   two :: Int -> Int -> (Int,Int)
>   two x y = (x*x,y+y)
>   g,h  :: Int -> Int
>   g z = let (a,b) = two z a in b
>   h z = let (a,b) = two b z in a
```

In this example, the function *two* takes two arguments, x and y, and returns two results $x * x$ and $y + y$. Inside the function there is no dependence between the multiplication and the addition. Thus code to evaluate the two halves of the pair can be put in either order when compiling the function under eager evaluation. This is not true under non-strict evaluation. In our example, the function *two* is used in two different contexts which require non-strictness. In the function g the argument z is given as the first argument to the function *two*, while the second argument to *two* is taken from its first result. This requires that *two* first computes $z * z$, returns the result, and then computes $(z * z) + (z * z) = 2z^2$. We see that in this case the multiplication is executed before the addition. In the function h the opposite occurs. The second result of the function *two* is fed back as the first argument. Here $z + z$ is computed first and then $(z + z) * (z + z) = 4z^2$. Now the addition is executed before the multiplication. Thus the multiplication and the addition have to be scheduled independently. Note that the scheduling is independent of the data values of the arguments; it depends only on the context in which the function is used and how results are fed back in as arguments. Larger examples of this form of non-strictness, used for less trivial computations, may be found in [9].

5.3.1.2 Conditional Non-Strictness

Non-strictness and the requirement for dynamic scheduling not only occur across function calls, but can also appear within conditionals. The following example, taken from [92], illustrates this.

```
>  kt :: Bool -> Int -> Int
>  kt p z = let (a,b,c) = if p then (y,z,x) else (z,x,y)
>               x = a+a
>               y = b*b
>           in c
```

```
>
> g,h :: Int -> Int
> g z = kt True  z
> h z = kt False z
```

In this example a single conditional steers the evaluation of three variables, a, b and c. If the predicate is true than $b \mapsto z$, $y \mapsto z * z$ and $a \mapsto y$, and the result c becomes $z * z + z * z$. In this case the multiplication is executed before the addition. If the predicate is false, the evaluations occur in a different order. First $a \mapsto z$ then $x\&b \mapsto (z + z)$, and finally y and the result c evaluate to $(z + z) * (z + z)$. Now the addition is performed before the multiplication. Again, we see that both the addition and the multiplication have to be scheduled dynamically. Although the operations appear outside the scope of the conditional, the conditional affects the order in which the values a, b and c are available.

Unlike the previous example, it may seem that the scheduling is at least data-dependent, since it is influenced by the conditional and therefore depends on the value of the predicate. While this observation is correct, we can obtain precisely the same behaviour without conditionals as is shown in the next example.

```
> f1 :: a -> b -> c -> (b,c,a)
> f1 x y z = (y,z,x)
> f2 :: a -> b -> c -> (c,a,b)
> f2 x y z = (z,x,y)
> kt2 :: (Int -> Int -> Int -> (Int,Int,Int)) -> Int -> Int
> kt2 func z = let (a,b,c) = func x y z
>                  x = a + a
>                  y = b * b
>              in c
>
> g,h :: Int -> Int
> g z = kt2 f1 z
> h z = ht2 f2 z
```

Here the conditional is replaced with a call to a function taking three arguments and returning three results. The argument *func* determines which function is called; it is f_1 in the case of g and f_2 in the case of h. The two functions f_1 and f_2 do not perform any computation; they merely shuffle the results around and thereby affect the order in which the addition and multiplication in the caller get executed. This example shows that in addition to the caller affecting the order in which operations get executed in the callee, the callee can also affect the order in the caller. In general, it is the whole context, i.e. the whole call tree, in which a function appears which determines the order.

5.3.1.3 Data Structure Non-Strictness

In non-strict languages, data structure constructors exhibit the same form of non-strictness as function calls, i.e. the result may be required before all the elements are defined. This gives the programmer the ability to define circular data structures, infinite data structures, or recursively

define some of the elements in terms of other elements. For example, the recursive binding `a = 1:a` denotes a simple cyclic list. This power of non-strictness also extends to list comprehensions and array comprehensions. Schauser and Goldstein [85] define the following hierarchy of non-strictness in data structures.

- **Functionally strict:** All elements must be evaluated before the data structure is created.
- **Circular:** A pointer to a data structure may be stored in one of its elements.
- **Recursive:** An element of a data structure may be defined in terms of other elements. For this case we can distinguish two sub-cases which describe the schedule used to fill the elements.
  1. **Static:** A static schedule can be found for the program which fills in the elements in the correct order.
  2. **Dynamic:** A dynamic schedule is required.

Frequently-cited examples of recursively defined data structures are wavefront [4] and the following function computing the first n Fibonacci numbers.

```
> fib_array :: Int -> Array Int Int
> fib_array n = a where
>  a = array (0,n) ([0 := 1,1 := 1]
>                   ++ [i := a!(i-1) + a!(i-2) |i <- [2..n]])
```

Both examples can be scheduled statically, i.e. a static scheduling of the program can be found which obeys the data dependencies among the array elements (left to right in the case of the Fibonacci series). With a static schedule it would be possible to implement the arrays as standard arrays in imperative languages (modulo polymorphism). However, this may not always be the case. Consider the array

```
> dyn_array :: Array Int Int -> Int -> Array Int Int
> dyn_array b n = a where
>  a = array (0,n) [i := if
>                          (i == b!0)
>                        then
>                          1
>                        else
>                          a!(mod (i-1) (n+1)) + 1
>                    | i <- [0..n]]
```

The schedule depends on the vector **b** which may not be available at compile time.

5.3.1.4 Infinite Iteration Lists

One function which appears repeatedly in functional scientific programs is the higher order function `iterate`

```
> iterate (a -> a) -> a -> [a]
> iterate f a = a : iterate f (f a)
```

which (lazily) constructs an infinite list of approximations to the value $f^{\infty}(a)$.

$$[a, f(a), f(f(a)), \ldots$$

This list is usually truncated and the value prior to truncation used. For example, a square root may be written as

```
> newton :: Double -> Double -> Double
> newton a x = (x + a/x) / 2
> sqrt :: Double -> Double -> Double -> Double
> sqrt init eps a = within eps (iterate (newton a) init)
```

where within looks along the list for a sufficiently accurate approximation to the true solution. However, this could be written without laziness as

```
> sqrt :: Double -> Double -> Double -> Double
> sqrt init eps a = iterateWithin eps (newton a) init

> iterateWithin :: Double -> (Double->Double) -> Double -> Double
> iterateWithin eps f x | abs(f x - x) < eps = f x
>                       | otherwise = iterateWithin eps f (f x)
```

or

```
> sqrt2 :: Double -> Double -> Double -> Double
> sqrt2 init eps a = NewtonIteration eps a init

> newtonInteration :: Double -> Double -> Double -> Double
> newtonIteration eps a current
>       | abs(new - current) < eps = new
>       | otherwise = newtonIteration eps a new
>                     where new = newton a current
```

or by inventing some special syntax

$$sqrt(val_{\text{init}}, \epsilon, a) = (val \mid val \leftarrow newton(a, val) \text{until}(\mid val - val_{\text{prev}} \mid < \epsilon) \mid val = val_{\text{init}})$$

Because this specific form of non-strictness is used so frequently we denote this as *iterative non-strictness*, since a program which depends on it could be trivially rewritten so that it no longer depended on non-strictness. Another reason for encouraging alternative styles of expression is that the code produced by the strict version is more efficient that the code produced by the lazy version.

Author code	Author
1	Kostas Ksickis [61]
2	Various (results taken from [85])
3	Stephen Bevan [8]
4	Sexton + Wainwright [100]
5	Chris Angus
6	Liu, Kelly & Cox [62]
7	NAS benchmark Fourier Transform [43]
8	John McCrory [65]

Table 5.2: Authors of functional software

5.3.2 Experimental Results

5.3.2.1 Programs

We use the benchmark programs shown in Tables 5.3+5.4. The programs chosen cover the gamut of numerical methods presented in Chapter 3. In this section we assess the degree of non-strictness required by a large set of functional numerical software (Tables 5.3+5.4), most of which are from sources other than the author (Table 5.2).

5.3.2.2 Categories

Tables 5.6+5.7 and 5.8+5.9 show the results of analysing the programs as detailed in Table 5.5[3]. The figures presented may be overly pessimistic as they ignore simple in-lining. For example, an application of

```
> sum xs = foldl (+) 0
```

often can, and will, be in-lined, eschewing the need to specialise the function **sum**. Within the specialisation categories of Table 5.5 *True* relates to the number of truly polymorphic non-function objects which are not given a monomorphic type. The other categories refer to the number of extra copies of functions/type definitions which would need to be generated for the code to become completely monomorphic.

5.3.2.3 Notes

† (Table 5.6): illustrates that the infinite structure in AnDiff is `[1..]` used to number variables. This can be trivially rewritten. FinElem imports AnDiff and so this result is carried over.
† (Table 5.8): shows an integer specialisation of a list which may be trivially converted to a loop in a strict language, i.e. `[lo..hi]` or `range (lo,hi)`, etc.
‡ (Table 5.8): refers to a specialisation which exists purely through an application of `iterate`. For example, an iterative linear system solver defined using `iterate` would require a list of vectors. Each of the columns denotes the number of *extra* specialisations which would be needed, rather than the

[3] We do not include I/O functions in the results.

Program	Brief Description	Author
ABA	Adams-Bashforth methods for 2^{nd}-order differential equations	3
ADAMS4	Adams 4^{th}-order predictor-corrector	1
AnDiff	Analytic differentiation	5
APCA	Adams-predictor-corrector methods for solving second-order differential equations.	3
Bisection	Bisection method for finding roots of $f(x) = 0$	3
CholeskiA	Choleski's algorithm for deriving the lower triangular matrix	3
CholeskiB	Choleski factorisation in Miranda	6
Choleski1	Choleski factorisation (Array)	5
Choleski2	Choleski factorisation (Quadtree)	5
ConjGrad1	Conjugate gradient solver	4
ConjGrad2	Conjugate gradient iteration	5
Crout	Crout reduction for tridiagonal linear systems	3
Cubic	Cubic spline interpolants	3
CYCR1	Cyclic-reduction (List)	8
CYCR2	Cyclic-reduction (Quadtrees)	5
DFP	Davidon-Fletcher-Powell (Optimisation)	5
Eigen3	Eigen problems	2
EULER1	Euler's method	1
EULER2	Modified euler	1
FEHLBERG	Runge-Kutta-Fehlberg	1
FinElem	Simple finite element code for n-dimensional p.d.e.	5
FixedPoint	Fixed-point method for finding roots of $f(x) = 0$	3
FDCN	Finite difference (Crank-Nicholson) for parabolic p.d.e.	5
FDE	Finite difference (explicit) for parabolic p.d.e.	5
FFT	Fast Fourier transform (NAS)	7
Gauss1	Gaussian elimination (Max Row pivot) + back substitution	5
Gauss2	Gaussian elimination with naive pivoting/back substitution	1
Gauss3	Gaussian elimination with maxima column pivoting/back substitution	1
GaussSeidel1	Gauss-Seidel	1
GaussSeidel2	Perform Gauss-Seidel iteration to solve $\mathbf{Ax} = \mathbf{b}$	3
Golden	Golden section	5
Hermite	Calculates the value of the Hermite polynomial	3
HEUN	Heun's method	1
Housholder	Eigen-solver	2
Horner	Horner's algorithm for evaluating polynomials $P(x), P'(x)$	3

Table 5.3: Table of numerical methods

Program	Brief Description	Author
Jacobi1	Jacobi's iterative method	1
Jacobi2	Perform Jacobi iteration to solve $\mathbf{Ax} = \mathbf{b}$	3
Jacobi3	Jacobi eigen solver	2
Jacobi4	Jacobi-iteration (Quadtree)	5
Jacobi_group	Jacobi eigen solver (group rotations)	2
LeastSquares	Calculate the gradient and intersection for a least squares	3
LDL	$\mathbf{LDL}^T$ method in Miranda	6
LU1	LU decomposition (Array)	5
LU2	LU decomposition (Quadtree)	5
LU3	LU decomposition (Miranda)	6
Matrix1	misc. operations on matrices	3
MM	Matrix multiply	2
MMT44	Blocked matrix multiply test	2
MCNP	Monte carlo photon transport	2
MDQUAD	n-dimensional multiple integrals	5
MPF	Multiplier-penalty-function (Optimisation)	5
Neville	Neville's iterated interpolation algorithm	3
Newton	Newton's method for non-linear equations	1
NewtonCotes	Newton-Cotes formulae for approximating an integral	3
NewtonDivided	Newton's interpolatory divided-difference formula	3
NewtonRaphson	Newton-Raphson method for finding roots of $f(x) = 0$	3
PCG	Preconditioned conjugate gradient method (Miranda)	6
QUAD	Numerical quadrature	5
QTrees	Matrix/vector quadtree operations	5
RungeKutta1	Runge-Kutta 4^{th} order	1
RungeKutta2	Runge-Kutta methods for 2^{nd}-order differential equations.	3
Secant	Secant method of finding roots of $f(x) = 0$	3
Simple	Hydrodynamics and heat conduction	2
Simplex	Simplex method	1
SOR1	Perform SOR iteration to solve $\mathbf{Ax} = \mathbf{b}$	3
SOR2	SOR method in Miranda	4
SOR3	SOR iteration (Quadtree)	5
Steffensen	Steffensen's method for finding roots of $f(x) = 0$	3
Strassen	Strassen multiplication	5
Taylor	Taylor methods for solving 2^{nd}-order differential equations	3
Vector	Misc. operations on vectors	3
Wavefront	Simple wavefront SOR	2

Table 5.4: Table of numerical methods (continued)

Category	Description	Values
Func.	Does the program require functional non-strictness?	Yes/No
Cond.	Does the program require conditional non-strictness?	Yes/No
Circ.	Does the program require circular non-strictness?	Yes/No
Iter.	Excluding I/O, does the program manipulate potentially infinite data, or would *only* iterative non-strictness suffice?	No/Yes/Iter.
Recu.	Does the program exhibit recursive non-strictness?	Yes/No
Dyn.	Does the program exhibit dynamic non-strictness?	Yes/No
Intrinsic	No. of extra specialisations needed for *map, filter, fold, append*	Integer
Other	No. of extra specialisations needed for user-defined Polymorphic functions	Integer
True	No. of true polymorphic objects	Integer
Overload	No. of extra specialisations needed for overloaded identifiers	Integer
List/Array	No. of extra specialisations of lists/arrays	Integer
Other Data	No. of extra specialisations of other data types	Integer

Table 5.5: Category descriptions

number of specialisations *per se*. Thus the optimal value for each column is zero. Specialisations of each function are denoted as x, y, meaning there are x extra specialisation of one function and y of another.

5.3.3 Discussion of Results

5.3.3.1 Non-Strictness

Table 5.6 shows that in the field of numerical methods the greatest use of non-strictness is in the recursive definition of arrays using schemes which admit a static schedule, functional, conditional and circular varieties being hardly used. The only program using any of these is a cubic spline interpolant generator which uses a technique called circular programming [9] in an attempt to increase the efficiency of a smaller, clearer incrementally updating program. Because any efficiency gain is amortised against the cost of the system necessary to support the technique (closures), we do not believe functional non-strictness is necessary to this area. The use of infinite data is restricted to iterations with the exception of the list `[1..]` being used as a numbering mechanism. Since we showed earlier that instances of `iterate` are trivial to replace we conclude that the only *necessary* incarnation of non-strictness is recursive data-structure non-strictness.

5.3.3.2 Polymorphism and Overloading

Table 5.8 shows that the greatest use of polymorphism/overloading is for sequence data structures, such as arrays or lists, and even for these there is very little. All the programs analysed defined polymorphic functions for ease of definition and called them under a single context. Most surprisingly, the programs analysed did not require *any* extra specialisation of overloadings. The only area where function specialisation is used in more than one context is functions such as *map, filter, length, concat*, etc. i.e. the tiny core of standard functions which make up all functional language preludes.

Program	Func.	Cond.	Circ.	Iter.	Recu.	Dyn.
ABA	No	No	No	Iter	No	No
ADAMS4	No	No	No	No	No	No
AnDiff	No	No	No	Yes†	No	No
APCA	No	No	No	Iter	No	No
Bisection	No	No	No	No	No	No
CholeskiA	No	No	No	No	Yes	No
CholeskiB	No	No	No	No	Yes	No
Choleski1	No	No	No	No	Yes	No
Choleski2	No	No	No	No	No	No
ConjGrad1	No	No	No	No	No	No
ConjGrad2	No	No	No	No	No	No
Crout	No	No	No	No	Yes	No
Cubic	Yes	No	No	No	Yes	No
CYCR1	No	No	No	No	No	No
CYCR2	No	No	No	No	No	No
DFP	No	No	No	No	No	No
Eigen3	No	No	No	No	No	No
EULER1	No	No	No	No	No	No
EULER2	No	No	No	No	No	No
FEHLBERG	No	No	No	No	No	No
FinElem	No	No	No	Yes†	No	No
FixedPoint	No	No	No	Iter	No	No
FDCN	No	No	No	No	Yes	No
FDE	No	No	No	No	No	No
FFT	No	No	No	No	Yes	No
Gauss1	No	No	No	No	No	No
Gauss2	No	No	No	No	Yes	No
Gauss3	No	No	No	No	Yes	No
GaussSeidel1	No	No	No	No	Yes	No
GaussSeidel2	No	No	No	Iter	Yes	No
Golden	No	No	No	No	No	No
Hermite	No	No	No	No	Yes	No
HEUN	No	No	No	No	No	No
Housholder	No	No	No	No	No	No
Horner	No	No	No	No	No	No

Table 5.6: Use of non-strictness in practice

Program	Func.	Cond.	Circ.	Iter.	Recu.	Dyn.
Jacobi1	No	No	No	No	No	No
Jacobi2	No	No	No	Iter	No	No
Jacobi3	No	No	No	No	No	No
Jacobi4	No	No	No	No	No	No
Jacobi_group	No	No	No	No	No	No
LeastSquares	No	No	No	No	No	No
LDL	No	No	No	No	Yes	No
LU1	No	No	No	No	Yes	No
LU2	No	No	No	No	No	No
LU3	No	No	No	No	Yes	No
Matrix1	No	No	No	No	No	No
MM	No	No	No	No	No	No
MMT44	No	No	No	No	No	No
MCNP	No	No	No	No	No	No
MDQUAD	No	No	No	No	No	No
MPF	No	No	No	No	No	No
Neville	No	No	No	No	Yes	No
Newton	No	No	No	No	Yes	No
NewtonCotes	No	No	No	No	No	No
NewtonDivided	No	No	No	No	Yes	No
NewtonRaphson	No	No	No	Iter	No	No
PCG	No	No	No	No	No	No
QUAD	No	No	No	No	No	No
QTrees	No	No	No	No	No	No
RungeKutta1	No	No	No	No	No	No
RungeKutta2	No	No	No	Iter	No	No
Secant	No	No	No	No	No	No
Simple	No	No	No	No	Yes	No
Simplex	No	No	No	No	Yes	No
SOR1	No	No	No	Iter	Yes	No
SOR2	No	No	No	No	No	No
SOR3	No	No	No	No	No	No
Steffensen	No	No	No	Iter	No	No
Strassen	No	No	No	No	No	No
Taylor	No	No	No	Iter	No	No
Vector	No	No	No	No	No	No
Wavefront	No	No	No	No	Yes	No

Table 5.7: Use of non-strictness in practice (continued)

Program	Intrinsic	Other	True	Overload	List/Array	Other Data
ABA	0	0	0	0	0	0
ADAMS4	0	0	0	0	0	0
AnDiff	0	0	0	0	1†	0
APCA	0	0	0	0	0	0
Bisection	0	0	0	0	0	0
CholeskiA	0	0	0	0	1†	0
CholeskiB	0	0	0	0	1,1†	0
Choleski1	0	0	0	0	1†	0
Choleski2	0	0	0	0	0	0
ConjGrad1	0	0	0	0	0	0
ConjGrad2	0	0	0	0	0	0
Crout	0	0	0	0	1†	0
Cubic	0	0	0	0	0	0
CYCR1	0	0	0	0	1†	0
CYCR2	0	0	0	0	0	0
DFP	0	0	0	0	1†	0
EULER1	0	0	0	0	0	0
EULER2	0	0	0	0	0	0
FEHLBERG	0	0	0	0	0	0
FinElem	1,3	0	0	0	3,1‡	0
FixedPoint	0	0	0	0	0	0
FDCN	0	0	0	0	2,1†	0
FDE	0	0	0	0	1†	0
Gauss1	0	0	0	0	1	0
Gauss2	0	0	0	0	1†	0
Gauss3	0	0	0	0	1†	0
GaussSeidel1	0	0	0	0	1†	0
GaussSeidel2	0	0	0	0	1†,1‡	0
Golden	0	0	0	0	0	0
Hermite	1	0	0	0	1†	0
HEUN	0	0	0	0	0	0

Table 5.8: Use of polymorphism and overloading in practice

Program	Intrinsic	Other	True	Overload	List/Array	Other Data
Horner	0	0	0	0	0	0
Jacobi1	0	0	0	0	1†	0
Jacobi2	0	0	0	0	1†,1‡	0
Jacobi4	0	0	0	0	0	0
LeastSquares	0	0	0	0	1†	0
LDL	0	0	0	0	1,1†	0
LU1	0	0	0	0	1†	0
LU2	0	0	0	0	0	0
LU3	0	0	0	0	1,1†	0
Matrix1	0	0	0	0	1†	0
MDQUAD	0	0	0	0	1	0
MPF	0	0	0	0	1†	0
Neville	0	0	0	0	1†	0
Newton	0	0	0	0	1†	0
NewtonCotes	0	0	0	0	0	0
NewtonDivided	0	0	0	0	1†	0
NewtonRaphson	0	0	0	0	0	0
PCG	0	0	0	0	1,1†	0
QUAD	0	0	0	0	0	0
QTrees	0	0	0	0	0	0
RungeKutta1	0	0	0	0	0	0
RungeKutta2	0	0	0	0	0	0
Secant	0	0	0	0	0	0
Simplex	0	0	0	0	2	0
SOR1	0	0	0	0	1†,1‡	0
SOR2	0	0	0	0	1†	0
SOR3	0	0	0	0	0	0
Steffensen	0	0	0	0	0	0
Strassen	0	0	0	0	0	0
Taylor	0	0	0	0	0	0
Vector	0	0	0	0	1†	0

Table 5.9: Use of polymorphism and overloading in practice (continued)

We could find no values for which a monomorphic type is not known at runtime. With this in mind we conclude that the only support needed in a functional language specialised to this area is an efficient sequence data type over all types and efficient support for a few intrinsic functions, all other features being *specialised out.*

5.4 Lessons Learned

Since the performance of Haskell is so drastically improved by unboxing it is believed that success in using functional programming languages for scientific computing relies on the use of strict languages. When we unbox a value we impose a level of strictness. The benefit of carrying out unboxing decreases as the complexity of the problem increases, and the act of unboxing a datatype removes the ability to use higher-order functions such as *map*, makes code monomorphic, and obscures the algorithm. As mentioned in previous sections, it is possible to simulate the manipulation of infinite structures in a strict language and it is also possible to express recursively-defined array problems in terms of matrix/vector operations. Because of these observations it is proposed that a strict language with recursive arrays should be used for this type of application, where either lazy behaviour is explicitly coerced in the style of Section 2.6, or specialised language constructs are used for common patterns of computation. This approach has the benefits of not having to unbox arithmetic expressions, an improved execution, and an improved ability to reason about the runtime behaviour of programs.

This view is echoed by experience with the SISAL [19] language, where efficiency comparable with hand-written C/FORTRAN for serial and parallel machines (mostly shared memory) has been achieved. Although a functional language, SISAL is quite primitive by modern language standards and is perhaps more accurately described as a single-assignment language with Pascal-like syntax. SISAL is monomorphic, explicitly typed but lacks many of the modern features of functional languages. The newer SISAL 90 [31] addresses some of these shortcomings, such as the first order nature of SISAL 1.2, but concentrates mainly on improving array operations (a feature which SISAL 1.2 is strong in anyway).

5.4.1 Non-Strict Semantics and Program Manipulation

We further vindicate our move towards strict semantics by identifying difficulties with proofs under non-strict semantics. We desire a language which is easily reasoned about and, because β-reduction is everywhere valid, initially non-strict languages seem the most promising. However non-strictness can easily lead to false reasoning. Consider the following axioms concerning the functions

```
> (++)::[a] -> [a] -> [a]    -- list append  ([1] ++ [2] = [1,2])
> elem ::p -> [p] -> Bool     -- element predicate (elem 1 [1,2] = True
> (||):: Bool -> Bool -> Bool -- logical OR

        (1) elem e (x ++ y) <=> elem  e x || elem e y
        (2) a || b          <=> b || a
```

Now consider the expression

```
elem 1 [1..]
```

which evaluates to `True` under non-strict semantics. If we apply our axioms we may derive an "equivalent" expression.

```
  elem 1 [1..]
= elem 1 ([1] ++ [2..])
= elem 1 [1] || elem 1 [2..] (by 1)
= elem 1 [2..] || elem 1 [1] (by 2)
= elem 1 ([2..] ++ [1])      (by 1)
```

which never terminates! The mistake that was made was in the definition of the second axiom. It is only valid when both arguments are either $\perp$ or both are $\neq \perp$. These difficulties are not insurmountable but require any reasoning to be carried out in terms of domain theory [88] (see Appendix I) rather than set theory. It is the author's opinion that applied mathematicians and engineers would much rather reason in terms of sets and induction rather than more abstract models.

5.4.2 Summary

With these issues in mind, a functional language, *Functional Scientific Computing* (FSC), has being developed specifically for the purpose of numerical programming in the style of Haskell, together with features found in other functional languages. It is the design, implementation and demonstration of this language which is the subject of Part II of this thesis.

5.5 Chapter Notes

i (page 74) Earlier we presented benchmarks for the simple Haskell program:

```
> value = sum[sum[ fromInt i * fromInt j|j<-[1..10000]]|i<-[1..10000]]
```

However, the program which was presented to the Haskell compiler was

```
> value :: Double
> value = (sum :: [Double] -> Double) [
>         (sum :: [Double] -> Double) [
>         ((*) :: Double -> Double -> Double)
>         ((fromInt :: Int -> Double) (i::Int))
>         ((fromInt :: Int -> Double) (j::Int))
>         | j <- [(1::Int)..(10000::Int)]]
>         | i <- [(1::Int)..(10000::Int)]]
```

to encourage GHC to perform as much optimisation as possible. The program's export list was also restricted to `main` to encourage in-lining.

Chapter 6

Related Work

6.1 Use of Functional Languages

In this section we conclude Part I by introducing some of the previous work in this area and relating this to the material presented.

There have been many attempts to apply functional programming to numerical methods over the last few years.

- D. S. Wise [105, 104, 106] uses quadtrees to define matrix/vector algorithms in a 'divide and conquer' fashion. The quadtree is used in the same manner as it is in [25], where it is called a hypermatrix. Wise presents algorithms for matrix inversion and the fast Fourier transform in the style of [76].
- The Quadtree approach is also used in the work of Wainwright & Sexton [100] where conjugate gradient and SOR methods are coded in Miranda for various sparse matrix representations. In their report they conclude that the quadtree is more suitable for CG-type algorithms than SOR, although, as mentioned in Chapter 4, their row-based formulation of SOR rather than a block-based formulation, would explain this.
- Page & Moe [73] define an executable specification of a reservoir model as a Miranda program embedded in a 200 page LaTeX document. Although executable, this code is not meant to be production quality and was later hand-coded in an imperative language. One of the interesting points arising from this study is the fact that, compared with previous software, the amount of documentation in the form of commentary approximately doubled, and the number of flagged lines of code approximately halved when a literate style of programming was employed. It is because of this that one of the areas this thesis later explores is a more literate style of programming.
- As part of the FLARE[1] [41] project a finite element program simulating fluid dynamics was written in Haskell. This concurs with [100] in that quadtree-type data structures were found to

[1] Functional Languages Applied to Real world Exemplars.

be faster than list based structures for sparse linear system solvers. The study concludes that compared to procedural programming, functional programming is more expressive and more easily maintained, with most errors being detected at the compilation stage. Lazy evaluation did not prove advantageous, with evaluation having to be forced for efficiency.

- Other very simple FE codes have been written in functional languages such as Miranda [62] (simple beam joint code), and SASL [26] (problem-specific elementary 1D PDE code), with emphasis mainly on the elegance of the implementation.

- Boyle, Fitzpatrick, Clint & Harmer [15, 14] use program transformations to translate code expressed as pure LISP and ML to FORTRAN. The approach proposed in this thesis differs from their work in that it aims to bring together many of the features found in various languages to support the construction of large scale numerical software, rather than writing transformational compilers for existing languages, although we do build on their work in Part II.

- Diaz & Shenoi [24] use Miranda to investigate decoupling systems of well equations in a Schur complement approach to domain decomposition using the number of reductions to measure work complexity. In the study they comment that *"functional syntax allows direct correspondence of functional code with the mathematical equations, which makes the code easy to develop, write, read and modify"*.

- Hartel & Vree [44] present a study of the use of arrays in functional languages using the fast Fourier transform [22] as a case study. They conclude " *...that algorithms published in the literature for imperative languages cannot always be translated directly into a* [lazy] *functional language, because efficiency considerations are of a different nature*" and that "*... an efficient implementation of arrays contributes significantly to the performance of functional languages in some areas. However a clear distinction should be made between array construction and array subscription*". For their Fourier transform they could not gain efficiency by using array construction, other than for storing precomputed data, like the input.

- Hammes, Sur & Böhm [43] investigate the effectiveness of Id and Haskell language features when writing scientific codes by implementing the NAS Fourier transform benchmark three-dimensional heat equation solver in both languages and comparing their resulting performance with a FORTRAN implementation. They conclude that these languages still have inefficient implementations with their largest executable problem (32^3) running at 15 times slower than FORTRAN using 3 times as much space!

- Sullivan & Zorn [90] compare various languages for suitability for implementing numerical methods including SML, C++ and Haskell although they approach the coding of a sparse Gaussian elimination benchmark with mutable state clearly in mind. As a result they are not able to offer a benchmark for Haskell as it suffers the same space and time efficiency problems as the incremental array LU factorisation presented in Chapter 4 (Copying arrays on update). They conclude that the features offered by Haskell make the code terse and readable but its current implementation causes it to be unusable on any but the smallest problems. "*In benchmarks ... we found the Haskell implementations to be 50 to 10,000 times slower than*

C++ or FORTRAN in array manipulations. Additionally, current Haskell implementations tend to die when handling arrays larger than 20 by 20' ([90] *Note 1* p.292). However this *dying* is more likely to be due to the multiple copying of arrays as in Chapter 4, rather than a bad implementation.

6.2 Summary

The preceding chapters have all concentrated on the use of the functional language Haskell to implement numerical methods and discussed its advantages/failings as a vehicle for the expression of these methods. In addition to this we have presented quantitative statistics concerning the level to which its specific language features are actually used in practice. This information is used in the following chapters to provide the motivation for the design and implementation details of a functional language specialised to this area (Fig. 6.1).

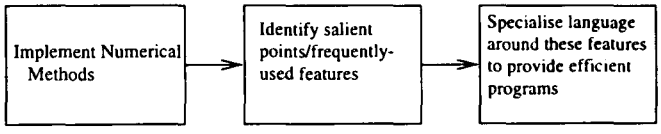


Figure 6.1: Development plan

This idea of using quantitative analysis to improve performance is not new, as it was exactly this thinking which led to the RISC [59] revolution in microprocessors. In our case this analysis leads directly to the design and implementation of a purely functional language tentatively named FSC (Functional Scientific Computing). This language is the focus of the second part of this thesis.

Part II

The FSC Language

Chapter 7

Language Design

In this chapter we discuss the principles which should underpin a functional, numerical programming language. We focus on (implementation independent) drawbacks and advantages of Haskell and discuss how its weak points may be circumvented, and its strong points mimicked, in an efficient manner.

7.1 Introduction

Before presenting a definition of FSC, we discuss its required features and how they may be implemented efficiently.

7.1.1 Characteristic Features

From experiences reported in Chapters 4-6, and from exposure to the other functional programming languages presented in Chapter 2, we conclude that the features which would prove useful in FSC are as follows:

- Specialisation *à la* NESL [10]
- Real arrays *à la* SISAL with array comprehensions
- Multi-parameter type classes
- Recursive arrays
- Simple I/O
- User-defined transformations
- Mixed-mode arithmetic
- 'Where' constructs and λ expressions

- An offside rule and terse syntax *à la* Haskell
- Guarded expressions
- Support for ADTs and pattern matching
- Parametric polymorphism (*textual*)
- Allowing uppercase characters to denote matrix identifiers
- A literate programming style *à la* Miranda/Haskell
- Strict semantics
- Implicit typing
- Partial application and sections
- Iterations *à la* SISAL

However, before discussing how these features may be collectively implemented we explain the rationale behind some of them:

- **Specialisation:**

  Experience reported in Chapters 4-6 suggests that the flavour of polymorphism supported should be *textual* and implemented via specialisation in the style of NESL [10]. This allows optimal code to be written using optimal representation of data, therefore allowing for more efficient code. Experiments reported in Chapter 5 show that the fear of an exponential explosion in code length (the main argument against this technique [97]) is not justified in our problem domain.

- **Real Arrays:**

  From the material presented in Chapters 4-6 we conclude that the language should be strict, with true arrays being readily available. The language should be based around arrays rather than lists. As such, array comprehensions should be provided, together with techniques for performing pattern matching on arrays. The ability to pattern match over an array, and consequently to regard an array as a free algebraic datatype, allows reasoning about array algorithms to be simplified. As such the language need not have built-in support for lists.

- **Multi-Parameter Type Classes:**

  Although Haskell's typeclass mechanism is a great improvement over standard Hindley-Milner polymorphism, it is felt that its insistence on type classes being single parameter is a major drawback to a scientific programming language as many common overloadings cannot be expressed in terms of a single parameter[1]. We discuss mechanisms for implementing multi-parameter type classes suited to this area, together with algorithms detailing how these may be integrated into the Hindley-Milner type system.

[1] Recently members of the Haskell community have also commented on this limitation which originated due to difficulties in compiling muli-parameter type classes[51].

- **Recursive Arrays:**

  Chapter 5 showed that recursive arrays are often used in the definition of matrix problems[2]. We investigate how recursively-defined arrays may be elegantly expressed in a strict language and discuss how these recursive arrays may be efficiently implemented eagerly.

- **Simple I/O:**

  Although powerful, functional I/O can often be unwieldy. The SISAL language presents a very simple, and easy to use, I/O system but, as a result, is constrained to simple batch-style programs. Haskell provides a monadic I/O system which allows interactive I/O but has the disadvantage that if all that is required is a batch-style operation then the program still looks unwieldy. We aim to integrate these two models of I/O as batch operation is common in numerical programming.

- **Program Transformation:**

  Boyle, Harmer, Clint and Fitzpatrick [15, 14] demonstrate the power and usefulness of user-defined transformations for expressing domain specific knowledge. We investigate how a transformation sub-language may be included in a Haskell-style language and discuss its semantics and implementation details.

Other features which we regard as worthwhile include:

- The ability to use upper case characters to denote identifiers.
- Iteration constructs *à la* SISAL, including *for* and *while* constructs, and a method of integrating these constructs with array comprehensions.
- Many of the features of Haskell including:
  - Partial application/sectioning
  - Algebraic types
  - Guarded expressions
  - λ-expressions
  - *where* constructs
  - Pattern matching
  - Implicit polymorphic typing.

  These features are preserved as we wish to retain the essence of Haskell, but improve its usefulness and ease of application to the numerical domain.

7.2 Simple I/O

In this section, suitable I/O mechanisms for numerical functional programming are discussed. However, before discussing these mechanisms we investigate existing methods of I/O, namely those of Haskell and SISAL.

[2] Recursive arrays were shown to be one of the few places where non-strictness is actually needed.

7.2.1 Haskell I/O

Over the last few years the model of I/O for Haskell has changed considerably[50]. However, Haskell 1.3 bases I/O around the notion of a *monad* [3, 98]. In this system the type of `main` is constant across programs, with

```
>       main :: IO ()
```

denoting that `main` takes no arguments and returns a value of type `IO ()`. This type represents a computation which yields the value `()`. We use the term *computation* rather than *function* to differentiate the fact that a side effect may occur within a computation. However, these *side-effects* are rigorously controlled and, by making the datatype `IO` abstract, all properties of equational reasoning are preserved although restrictions are imposed by the structure of the equations.

The interface to `IO` includes a *binding* function, `(>>=)`, whose type is

```
>       (>>=) :: IO a -> (a -> IO b) -> IO b
```

and a function `return` whose type is

```
>       return :: a -> IO a
```

The purpose of `return` is to transform a value into the trivial computation yielding that value, and the purpose of `(>>=)` is to sequence two existing computations. The reason that equational reasoning is valid is that once a value has been raised to the status of computation there is no way of returning it to a simple value (because the datatype is abstract), so side-effecting expressions, such as

```
>       let
>               x = readIntFromStdin() + readIntFromStdin()
>       in
              ...
```

which would normally cause a common subexpression elimination

```
>       let
>               z = readIntFromStdIn()
>               x = z + z
>       in
              ...
```

to be invalid, cannot occur since a side-effecting function of type

```
>       readIntFromStdIn :: () -> Int
```

can never be constructed.

Using Monadic I/O, a Haskell program which receives two strings from the user parses them as integers using a suitable function `readInt` and prints their sum may be written

```
>       main :: IO ()
>       main = getLine >>= \x ->
>              getLine >>= \y ->
>              print (readInt x + readInt y)
```

which cannot be invalidated by common subexpression elimination[3]. One disadvantage of this method is that data must be explicitly described by the user.

7.2.2 SISAL I/O

SISAL does not have any I/O system to speak of, other than the fact that its entry point may have any type (its arguments being the values supplied by the user and its result being the value printed by the program). This simplicity is attractive as it removes the burden of parsing datatypes from the user, although it is not suitable for programs which require interaction. The previous program could be written in SISAL's Pascal-like syntax as

```
DEFINE main

FUNCTION main(x,y:INTEGER RETURNS INTEGER)
        x + y
END FUNCTION
```

whose charm is its simplicity.

7.2.3 FSC I/O

FSC combines the Haskell and SISAL I/O models as follows. The main function is of the following type:

$$\text{main} :: \tau_0 \rightarrow \tau_1 \rightarrow \ldots - \tau_n \rightarrow \text{IO}(\phi_1, \phi_2, \ldots, \phi_m) \qquad m \geq 0, n \geq 0$$

i.e. the type of main has the syntax shown in Fig. 7.1.

The addition example from Subsection 7.2.1 may be written in FSC either as:

```
>       main :: INT -> INT -> IO ()
>       main x y = print (x + y)
```

or as

```
>       main :: INT -> INT -> IO INT
>       main x y = return (x + y)
```

which retains the simplicity of the SISAL version.

[3]This program relies on the existence of two further intrinsic functions, `getLine::IO String` and `print::Show a => a -> IO ()`, being in the interface of the abstract datatype. These are computations which return a line of input and display a *printable* value, respectively.

Type-syntax			
Main type	ϕ	::=	IO()
		|	IO(ψ)
		|	$\tau \rightarrow o$
Return type	ψ	::=	τ
		|	τ, ψ
type	τ		All types

Figure 7.1: Formal type-syntax of FSC entry-points

7.2.4 Combining I/O Actions

To combine I/O actions we borrow the *do* syntax of GoFER to sugar monad operations. In GoFER syntax our original Haskell example is written

```
>       main = do
>                x <- getLine
>                y <- getLine
>                print (readInt x + readInt y)
```

which is a simple sugaring and translates directly into the original Haskell example.

7.2.5 Translation

These examples may be translated into the simpler language SISAL by regarding `IO` to be a function which takes an integer and returns a pair which includes an integer:

```
>        datatype IO a = INT -> (INT,a)
```

This being the case, `return` can be considered as an arity-2 function and `(>>=)` as an arity-3 function with the following, continuation-passing style, SISAL translations:

```
        (>>=) m k io = LET
                           new_io,result = m(io)
                       IN
                           k(result,new_io)
                       END LET

        return a io  = io,a
```

Hence the FSC function

```
>              println :: String -> IO ()
>              println s =
>                        do IO
```

```
>                          print s
>                          putchar '\n'
```

may be translated to the intermediate code

```
>             println :: String -> INT -> (INT,())
>             println s init
>              = (>>=) (\io -> print s io) (\res io -> putchar '\n' io) init
```

and then to a SISAL equivalent

```
FUNCTION PRINTLN(STR:ARRAY[CHARACTER]; IO_INIT:INTEGER
                        RETURNS IO(VOID))
       LET
           IO_1,RES_1 := PRINT(STR,IO_INIT);
       IN
           PUTCHAR('\n',IO_1);
       END LET
END FUNCTION
```

We may generate interactive SISAL code by making use of SISAL's foreign language interface to C in a style analogous to GHC's implementation of I/O, although these details are inside the PRINT and PUTCHAR functions provided in a standard prelude. The use of the IO keyword is tells the compiler it is dealing with the IO-monad, thus simpilifying compilation and error reporting.

7.3 Array sugaring

Pattern matching is very useful for writing compact and readable programs. Unfortunately knowledge of the concrete representation of an object is necessary before pattern matching can be invoked.

To apply pattern matching to arrays we initially regard an array as an abstract data type with the operations shown in Table 7.1[4] .

And then sugar these operations as follows

```
>      (i <: x)     = setliml i x

>      (i :> x)     = setlimh i x

>      (a <+ A)     = appendLow A a

>      (A +> a)     = appendHigh A a
```

These four operators can be hard-coded into the language and pattern matching should be allowed over them.

[4]The names of these functions have been chosen to be familiar to Haskell and SISAL programmers.

Prototype	Description
`liml::Array a->INT`	`liml A` returns the lower limit of `A`.
`limh::Array a->INT`	`limh A` returns the upper limit of `A`.
`setliml::INT->Array a->Array a`	`setliml n A` sets the lower limit of `A` to n.
`setlimh::INT->Array a->Array a`	`setlimh n A` sets the upper limit of `A` to n.
`empty::Array a->BOOL`	The `empty A` predicate is True if `A` has no elements.
`appendLow::Array a->a->Array a`	`appendLow A a` extends the array `A` by element a at its lower bound.
`appendHigh::Array a->a->Array a`	`appendHigh A a` extends the array `A` by element a at its upper bound.
`init::Array a->Array a`	`init A` removes the upper-most element from `A`.
`tail::Array a->Array a`	`tail A` removes the lower-most element from `A`.
`[ ]::Array a->INT->a`	`A[i]` selects the element associated with subscript i.
`[]::Array a`	`[]` is the array with no elements.

Table 7.1: Operations over arrays

7.3.1 Comment

Pattern matching is important as it allows us to provide elegant specifications of highly-efficient built-in functions such as *map*. Pattern matching also allows us to define specifications which may be refined if necessary and simplifies inductive proofs, as an inductive proof[5] is simply a statement that a fact is valid over all the constructors of its datatype.

7.4 Recursively Defined Arrays

As mentioned in Chapters 4-6, functional languages such as Haskell allow the definition of *non-strict monolithic arrays* using *so-called* "array comprehensions", although the fact that these are bolted on top of list comprehension syntax makes them slightly inelegant. *Non-strict* arrays may contain undefined (i.e. $\perp$) elements and still be well-defined overall; this is in contrast to *strict* arrays which are completely undefined if any one element is undefined. Non-strict arrays are more in the spirit of lazy, or call-by-need, evaluation, whereas strict arrays capture the essence of call-by-value computation [2]. Functional arrays can be categorised further depending on their method of definition:

- In *monolithic* arrays, elements are defined when the array is created.
- In *incremental* arrays, elements are defined incrementally.

[5] In a strict language.

Although both types of array are functional, monolithic arrays are more in the spirit of functional programming and incremental arrays resemble the imperative array model.

7.4.1 Strict Versus Non-Strict Arrays

We begin by defining the difference between non-strict and strict arrays.

> **Definition : Let $A[i]$ be the value at subscript i of array A. An array is *strict* if for any i within the bounds of A. $A[i] = \perp$ implies that $A = \perp$[2].**

From this it is easy to show that if a strict array is recursively defined the entire array must evaluate to $\perp$. However, in the field of scientific computing, recurrence relations are frequently used and hence we prefer a non-strict array constructor for this expressiveness. Unfortunately, in general, non-strict arrays must represent the delayed computation of array elements as closures (thunks) which incur prohibitive runtime costs.

7.4.2 Strict Contexts

In most scientific programs the programmer knows that an array is used in a context that involves all of its elements. Hence we may be able to treat a recursively defined array like a strict array. To do this, we need to know:

- that the non-strict array is used in a strict context, and
- a safe (partial) order of evaluation of the elements such that no element is evaluated until *after* every element on which it depends.

Since FSC is a strict language we choose to take the approach that all recursive arrays are used in strict contexts, together with the ability to offer a (partial) ordering and an array which should be overwritten by the new array during the computation. This is in direct agreement with the needs that were identified in Chapter 5.

7.4.3 Wavefronts

Consider the wavefront example

$$a_{i,j} = \begin{cases} 1 & \text{if } i = 1, \quad 1 \leq j \leq n \\ 1 & \text{if } j = 1, \quad 2 \leq i \leq n \\ a_{i-1,j} + a_{i-1,j-1} + a_{i,j-1} & 2 \leq i \leq n, \quad 2 \leq j \leq n \end{cases}$$

which would fill in a two-dimensional array in a south-easterly direction. In a lazy language such as Haskell this could be written as:

```
>       a :: Array (Int,Int) Int
>       a = array ((1,1),(n,n)) ( [(1,j) := 1  | j <- [1..n] ] ++
>                                 [(i,1) := 1  | i <- [2..n] ] ++
>                                 [(i,j) := a!(i-1,j)
```

```
>                                          + a!(i-1,j-1)
>                                          + a!( i ,j-1)
>                                               | i <- [2..n],
>                                                 j <- [2..n] ] )
```

However, the style of notation we suggest for FSC is:

```
>    A :: [[INT]]
>    array A[(1,1)..(N,N)] where
>                        A[1,j] = 1
>                        A[i,1] = 1
>                        A[i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1]
```

i.e. a pattern matching function defines the array. At this point no ordering is assumed and dynamic code can be generated which, at runtime, ensures that the data-dependancies are respected (see later). However, efficiency may be improved by specifying an ordering on the data in the style of [37, 38]:

```
>    A :: [[INT]]
>    array A[(1,1)..(N,N)]
>         ordered
>                   [(i,1)| i in [1..N]] and  [(1,j)|j in [2..N]]
>         then by k in [4..2*N]
>                   [(i,j) when (i+j==k)|i in [2..N];j in [2..N]]
>         where
>                  A[1,j] = 1
>                  A[i,1] = 1
>                  A[i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1]
```

Efficiency may also be regained by overwriting the original array, so that in the example of SOR iteration

```
>    U :: [[DOUBLE]]
>    array U[(1,1)..(N,N)] where
>        U[i,j] | (i,j) is on_boundary = A[i,j]
>               | otherwise
>                           = U[i,j-1]
>                 + U[i-1,j]          + A[i+1,j]
>                           + A[i,j+1]

>        on_boundary (i,j) = i==1 || i==N || j==1 || j==N
```

We could also specify (dangerously) that an attempt be made to overwrite **A** with the values of **U** if we knew this could not affect the semantics of the expression as is the case in this example. Also, if this was not the last use of **A** it would not be overwritten. However we may be able to coerce this into being the *last use* of **A** since FSC semantics are strict.

```
>     U :: [[DOUBLE]]
>     array U[(1,1)..(N,N)] overwrites A where
>         U[i,j] | (i,j) is on_boundary = A[i,j]
>                | otherwise
>                             = U[i,j-1]
>                  + U[i-1,j]           + A[i+1,j]
>                             + A[i,j+1]

>         on_boundary (i,j) = i==1 || i==N || j==1 || j==N
```

7.4.4 Implementation

In this SOR example each element is defined via a *star* which accesses its neighbours, as in Fig. 7.2. The dependencies which must be respected by the compiler can be seen in Fig. 7.3, where a dotted arrow represents a *basic* dependence and a continuous arrow a *recursive* dependence. With the basic dependencies we are free simply to return the values, but with the recursive dependencies we must evaluate the elements in order. This order is maintained by generating *tag arrays* for representing recursive arrays, that is an array of the same size which holds tags regarding the evaluation extent of each element in the array.

7.4.4.1 Example

Consider the 4×4 array shown in Fig. 7.5 At the start of the computation all of these tags are set to *undefined* and the elements of the data array are undefined (Fig. 7.4). (If we are overwriting **A** then this data array would be **A** itself.) We then pick an arbitrary path through the array from start to finish (for the sake of explanation we choose $15 \ldots 0$, as $0 \ldots 15$ is not very interesting). Elements 15 to 11 are basic dependencies so they require no more work than writing their values and setting their tags to *defined* (Fig. 7.6[A]). It is when element 10 is reached that we come across a recursive dependence which is handled by locking the tag and making a recursive call to the function which evaluates the array. This call evaluates element 6 (element 10's first recursive dependent). A *side effect* of evaluating U_6 is that U_1, U_2, U_4 and U_5 are also evaluated (Fig. 7.6[B]). Once this is done the evaluation of U_{10} proceeds to its next dependent U_9 and evaluates this (a side effect of which is that U_8 is evaluated (Fig. 7.6[C])). Once all its recursive dependents have been checked U_{10} is built and its tag set to *defined* (Fig. 7.6[D]). U_8 and U_9 are skipped since their tags are set and U_7, U_3 and U_0 are defined via basic dependencies. The tag array is of no use now and is discarded, leaving the evaluated array. The initial SOR code from Section 7.4.3 would be automatically translated to

```
>     on_boundary :: Int -> (Int,Int) -> Bool
>     on_boundary N (i,j) = i == 1 || i == N || j == 1 || j == N


>     U::(Int,Int) -> Int -> [[Double]] -> [[TAG]]
>                  -> ([[Double]],[[TAG]])
```

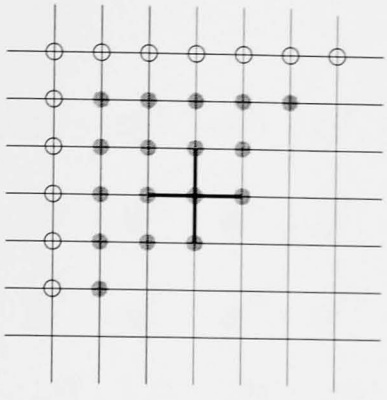


Figure 7.2: SOR star

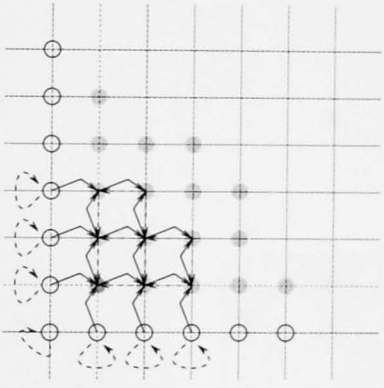


Figure 7.3: SOR dependencies

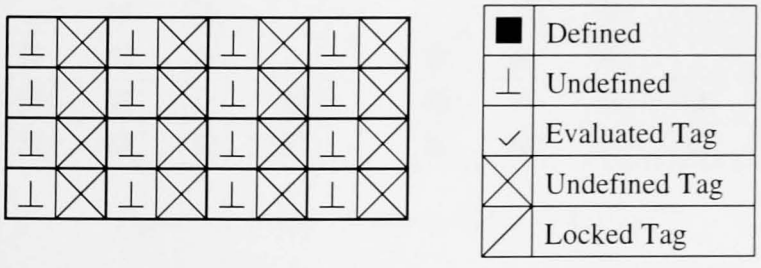


Figure 7.4: Initial data in recursive SOR implementation

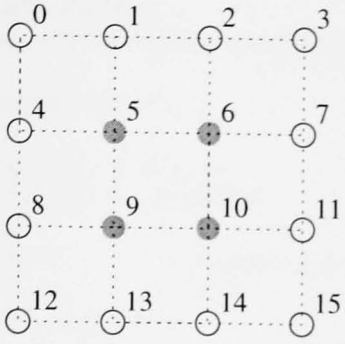


Figure 7.5: SOR grid

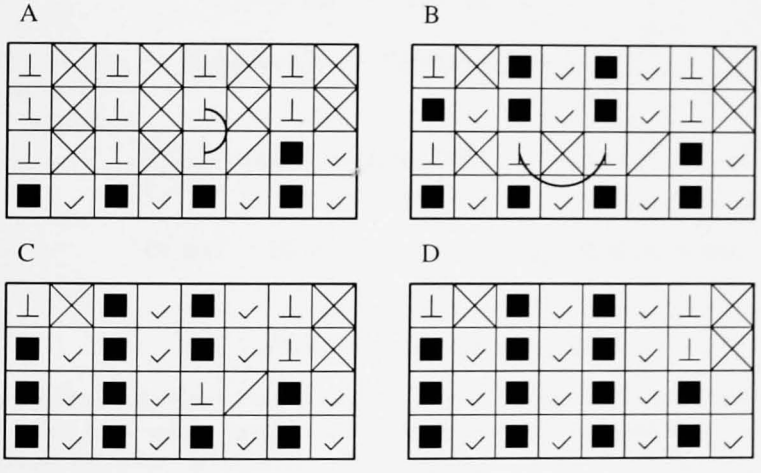


Figure 7.6: Stages of recursive SOR evaluation

```
>     U (i,j) N U_00 T_00 =
>      case(T_00[i,j]) of
>        DEFINED -> (U_00,T_00)
>        LOCKED  -> error "BLACK HOLED \
>                          on evaluating recursive array U"
>        UNDEF   -> if
>                        on_boundary N (i,j)
>                   then
>                       (U_00[(i,j) -> A[i,j],
>                        T_00[(i,j) -> DEFINED])
>                   else
>                        let T_01        = tags[(i,j)->LOCKED]
>                            U_01        = U_00
>                            (U_02,T_02)
>                               = U (i,j-1) N U_01 T_01
>                            (U_03,T_03)
>                               = U (i-1,j) N U_02 T_02
>                            result = U_03[i,(j-1)]
>                                   + U_03[(i-1),j]
>                                   + A[i,j+1]
>                                   + A[i+1,j]
>                        in
>                           (U_03[(i,j) -> result],
>                            T_03[(i,j) -> DEFINED])

>     U_TRAVERSE :: Int -> [[Double]] -> [[Double]]
>     U_TRAVERSE N A
>              = let
>                      U_00 = build_initial_U N
>                      T_00 = build_initial_T N
>                in
>                      (U_01|   (U_01,T_01) <- U (i,j) N U_01 T_01;
>                                 i in [1..N];
>                                 j in [1..N]
>                           |   (U_01,T_01) = (U_00,T_00))
```

where U is a recursive function which mimics the structure of the array U, the syntax `A[i->v]` is read as *replace the i^{th} value of A with v* and the function U_TRAVERSE evaluates each of the elements of the array U in a single-threaded, arbitrarily ordered, manner before returning the evaluated array.

7.4.5 Preliminary Results

Various schemes were tested for the following recursively defined array

$$g_n = \sum_{k=1}^{n} a_k, \qquad \text{where } \begin{aligned} a_1 &= 0.0 \\ a_2 &= 1.0 \\ a_i &= a_{i-1} + a_{i-2} + 1.0 \end{aligned}$$

which was used to compute

$$\sum_{k=999900}^{1000000} (\text{if } g_k = 0.0 \text{ then } 1.0 \text{ else } 2.0)$$

This example was chosen as the computation is suitably large, involves floating-point arithmetic, and a small amount of time is spent on I/O. Comparative results are shown in Table 7.2, while the

Version	In-code Time (Seconds)
gcc	208.96
gcc -O4	174.66
Array Scheme 1	351
Array Scheme 1(a)	311.17
Array Scheme 1(b)	284
Ordered Array Scheme 2	192
Ordered Array Scheme 2(a)	182
Haskell	(Exhausted 32Mb heap)
Haskell on problem 1/10 of size	15 mins (unix time)

Table 7.2: Recursive arrays versus Haskell and C

schemes are detailed in Table 7.3. The scheme used in 7.4.4 was Array scheme 1.

Scheme	Explanation
Array Scheme 1	Simplest scheme using tristate tags.
Array Scheme 1(a)	Scheme using Boolean tags.
Array Scheme 1(b)	As 1(a) except undefined data is not zeroed.
Array Scheme 2	Order is made explicit and hence tags are not needed.
Array Scheme 2(a)	As 2 except undefined data is not zeroed.

Table 7.3: Compilation scheme key

7.4.6 Initial Conclusions

The results shown in Table 7.2 are most promising as the essence of Haskell's recursive arrays have been captured in an economical manner. The method allows algorithms to be prototyped with no ordering and an ordering applied at a later date. The increase in speed over Haskell (> 25) is

extremely encouraging as the method provides all the power we need to express the computations investigated in Chapters 4-6 at a fraction of the cost. In the above example, the worst case where no information on ordering is given runs at 50% of the speed of optimised C, but 250% faster than Haskell on a problem an order of magnitude smaller[6]. However the best case benchmarks at 96% of optimised C.

7.4.7 Notes

Much of the syntax for recursive arrays builds on the work of Gao *et al* [37, 38] although their work does not detail implementation issues. Although the specification of an ordering is left to the user, many checks may be made at compile time[37, 38]. As regards efficiency, the previous example involved only floating-point addition. As the computation increases the overhead of managing a tag array reduces its dominance on the cost of the computation.

7.5 Support for partial application

A key area in the design of a numerical functional language is array and function support. In curried languages such as Haskell or ML a function can be written

```
>       f :: Int -> Int -> Int -> Int
>       f x y z = x + y + z
```

This is removed from the idea of functions in languages such as C where a similar function could be written as

```
int f(int x,int y,int z)
{
  return (x + y + z);
}
```

The advantage of writing the above function in its curried form is that it can be partially applied (e.g. `g = f 1 2` is a function which adds 3 to its argument). This syntax is clumsy as it is only the last argument which may be applied later. With this in mind, the following syntax is introduced

```
>       f{x,y,z} = f x y z
```

the advantage being that we may partially apply this function via the use of underscores

```
>       f{x,_,z} =  \y -> f x y z
```

This idea of under-scoring can be carried over to array definitions, where, if we had a two-dimensional array **A**, we can only access one dimension easily. This is resolved similarly.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

[6] In the Haskell version, the range of k was (99990,100000) rather than (999900,1000000) as GHC ran out of resources at the original range.

If we consider the above array then

```
A[1,1] = 1
A[1,2] = 2
A[1]   = [1,2,3]
A[1,_] = [1,2,3]
A[_,1] = [1,4,7]
```

and we may define matrix multiplication as

```
>       A * B = C where array C[i,j] = A[i,_] * B[_,j]
```

if there exists an appropriate dot product version of *.

Similarly, matrix-vector multiplication could be written as

```
>       A * v = b where array b[i] = A[i] * v
```

This syntax allows us to slice arrays in a style syntactically similar to that used in functions. The above examples exhibit implicit bounds definition, i.e. if an array has not defined its extent explicitly then it should be taken to be as large as possible. To declare bounds implicitly then all subscripts should be simple linear expressions. If no finite bounds can be determined then a compilation error should be returned.

7.5.1 Example

The following code

```
>       a:: [DOUBLE]
>       array a[i] = b[C1 * i + C2]
```

is translated to

```
>       a :: [DOUBLE]
>       a = let
>               (lo <: _ :> hi) = b
>               low = (lo - C2) div C1
>               high= (hi - C2+extra C1) div C1
>               extra 1 = 0
>               extra n = 1
>           in
>               low<:[b[C1*i + C2] | i in [low..high]]
```

If more than one array is used then the convention which seems intuitive (and is also used in texts such as [36]) is that expressions combined with addition should have their range defined as the union of the ranges of the left and right operands and all other combining functions should cause the range to be defined as the intersection of the ranges of each argument. Out of range expressions should be zero-ed for addition purposes and the union and intersection of array ranges are defined as

$$[a_1..b_1] \bigcup [a_2..b_2] = [\min(a_1, a_2)..\max(b_1, b_2)]$$
$$[a_1..b_1] \bigcap [a_2..b_2] = [\max(a_1, a_2)..\min(b_1, b_2)]$$

Note: since this is a syntactic convention it is neither compatible with α-conversion nor γ-equality, and as such, use of the identifiers $\{+,-\}$[7] within an array definition should be considered distinct to the use of these identifiers elsewhere. That is, in the following definitions array `X` may have a greater number of elements than `Y` since `+` causes the union of ranges and `'plus'` the intersection.

```
>       array X[i] = b[i] + c[i]
>       array Y[i] = b[i] 'plus' c[i] where plus = (+)
```

7.6 Example: Cyclic Reduction

Using ideas from the previous section, the computational section of cyclic reduction [36] for a tridiagonal matrix

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ \vdots \\ \vdots \\ y_n \end{bmatrix}$$

may be expressed as follows:

```
>    a',b',c',y' alpha,beta :: [DOUBLE]
>    array a'[i]     =  alpha[i]*a[i-1]
>    array b'[i]     =  b[i] + alpha[i]*c[i-1]+beta[i]*a[i+1]
>    array c'[i]     =  beta[i] * c[i+1]
>    array y'[i]     =  y[i] + alpha[i]*y[i-1] + beta[i]*y[i+1]
>    array alpha[i] = ~a[i]/b[i-1]
>    array beta[i]  = ~c[i]/b[i+1]
```

which compares favourably with the textbook definition taken from [36]

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1} \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \beta_i a_{i+1} \\ c_i^{(1)} &= \beta_i c_{i+1} \\ y_i^{(1)} &= y_i + \alpha_i y_{i-1} + \beta_i y_{i+1} \\ \alpha_i &= -a_i/b_{i-1} \\ \beta_i &= -c_i/b_{i+1} \end{aligned}$$

The only disadvantage is that substantial bounds checking occurs. A solution is to allow extra information relating the bounds of the arrays:

[7] Minus is included as $a - b \equiv a + (-b)$.

```
>     a::{2..N};
>     b,y::{1..N};
>     c::{1..N-1};
```

where `N` is a value already in scope. Like the recursive array examples presented earlier, this is another example of a textbook-style definition followed by extra information to increase efficiency.

7.7 Array Stripping

Often an expression is more easily understood in terms of an intermediate value using either a `let` expression or a `where` clause:

```
>       let
>               temp = f x1 ... xn
>       in
>               g y_1 ... y_p temp y_q ... y_n
```

This use of local values may be easily in-lined via β-reduction and no allocation is needed for a temporary structure. However, if there are multiple instances of the temporary value then β-reduction may cause extra work to be done:

```
>       let
>               temp2 = f x1 ... xn
>       in
>               g y_1 ... y_p temp2 temp2 y_q ... y_n
```

i.e. in-lining will double the number of invocations of the function `f`. If the local value is not an array then checking the number of references to it is sufficient to allow in-lining. However, if the structure is an array then the decision whether or not to inline is more complex. We call the process of removing intermediate arrays, *array stripping*.

7.7.0.1 Example

The simplest example of an array strip is a simple β-reduction:

```
>       A,B :: [INT]
>       array A[i] = F x[i]
>       array B[i] = F A[i]
>       ...
> {-  No further references to A -}
```

This expression becomes

```
>       B :: [INT]
>       array B[i] = F (F X[i])
```

However, other situations exist where there is more than one reference to the array yet array stripping would still prove beneficial, such as the following example:

```
>       A,C,D,E :: [INT]
>       array A[i] = B[i]
>       array C[i] = A[2*i]
>       array D[i] = A[2*i+1]
>       array E[i] = D[i] + C[i]

>       -- No further dependency on A, C or D
```

Here the references to A are mutually exclusive, i.e. each element of the array is accessed at most once only and hence may be in-lined. From this we conclude that the necessary condition for stripping without fear of causing extra computation is that each element of an array is accessed at most once only, with a view to transforming the above example into:

```
>       E :: [INT]
>       array E[i] = B[2*i+1] + B[2*i]
```

However, in the following code, array C should not be stripped as this would increase the number of invocations of f1.

```
>       let array C[i] = f1 A[i+1]
>           array B[i] = f2 C[i]
>           array D[i] = f3 C[i+1]
>       in
>           f4 B D;
```

The rest of this section discusses tests which facilitate a general strip decision for the above examples. The method we suggest for evaluating whether a strip can be performed safely is as follows:

- The iterations containing the accesses are normalised. If this cannot be done then the value of any identifiers used in an access is taken to mean *all* possible integers.
- An $N \times N$ matrix of Booleans **B** is built, where N is the number of references to the structure in question, and b_{ij} is True if and only if the i^{th} and j^{th} references to the array in question are mutually exclusive.
- A linear Diophantine[8] equation is constructed for each non-diagonal element of the matrix which is tested using the *greatest common divisor* (GCD) test (given below). If all tests show independence then the array is not built and the accesses are in-lined.

[8] A *Diophantine* equation is a linear algebraic equation of the form $\sum_{i=1}^{N} c_i x_i = c_0$, where both the x_i and the c_i are integers.

7.7.1 The GCD Test

Testing whether n array accesses are mutually exclusive is exactly the same as testing whether they are pairwise-parallelisable for all possible pairs. A simple test for parallelisability is the GCD test [108]. To apply this test, two array accesses are chosen and a Diophantine equation is constructed from their index expressions. To construct this equation the index expressions are set equal to each other with their variables being kept distinct.

7.7.1.1 Example

The accesses

```
>        A[2I + 2J]
        ...
>        A[4I - 6J +3]
```

would cause the equation

$$2x_1 + 2x_2 = 4y_1 - 6y_2 + 3$$

to be constructed. Collecting all variables on the left yields

$$2x_1 + 2x_2 - 4y_1 + 6y_2 = 3$$

which is the Diophantine equation which we wish to examine for solvability. This may done by finding the *greatest common divisor* of the coefficients in the Diophantine equation. That is, for a Diophantine equation.

$$\sum_{i=1}^{N} c_i x_i = c_0$$

we compute the greatest common divisor of the set $\{c_i | i \in \{1, 2, .., N\}\}$ as follows

$$\begin{array}{lcl} \gcd(\{0,0\}) & = & \bot \\ \gcd(\{a,0\}) & = & |a| \\ \gcd(\{a,b\}) & = & \gcd(\{|b|, \mathrm{mod}(|a|,|b|)\}) \\ \gcd(\{\}) & = & \bot \\ \gcd(\{a_1, a2, \ldots, a_N\}) & = & \gcd(\{a_1, \gcd(\{a_2, \ldots, a_N\})\}) \end{array}$$

and test whether $g = \gcd(\{c_1, c_2, \ldots, c_N\})$ divides c_0.

If $g \not| c_0$ then no dependence exists between the two array accesses and they are pairwise parallelisable.

As mentioned earlier, applying this test to all possible pairs yields a test for strippability applicable to our original example.

7.7.1.2 Comment

This section is intended to be a discussion of how the GCD test may be used to provide an array-stripping decision, rather than an introduction to data dependence analysis in super-compilers. The interested reader is encouraged to investigate [108] for a more in-depth treatment.

7.8 Summary of FSC arrays

Since arrays form an integral part of FSC we provide examples in this section.

7.8.1 Array access

FSC arrays are accessed using [] notation:

```
>       A[1]
```

Arrays of arrays may be accessed similarly:

```
>       B[i,j] = B[i][j] = (B[i])[j]
```

Array access may be partial using the underbar:

```
>       C[i,_] = C[i]
>       D[_,j] = \i -> D[i,j]
```

Arrays may also be pattern-matched against using the +> and <+ operators:

```
>       (aHead <+ aTail) = A
>       (aInit +> aLast) = A

>       reverse (l <+ m +> r) = r <+ reverse m +> l
>       reverse other         = other
```

7.8.2 Array Bounds Interrogation

The bounds of an array may be determined via pattern matching using the :> and <: operators:

```
>       determineBounds (low <: A :> high) = (low,high)
```

7.8.3 Array Modification

Arrays are modified using the [->] syntax:

```
>       A = [1,2,3]
>       A' = A[1 -> 0] = [0,2,3]
```

7.8.4 Literal Array Definition

Literal arrays may be defined as follows:

```
>       emptyArray = []
>       intArray   = [1,2,3]
>       charArray  = ['a','b','c'] = "abc"
>       twoDarray  = [emptyArray,[4,5,6],intArray]
>       intArrayWithLowerBoundOf2 = 2 <: [1,2,3]
```

7.8.5 Arrays as Strides and Sequences

An array may be defined as an arithmetic sequence or as a Stride:

```
>       digits = [0..9]         = [0:9] = [0,1,2,3,4,5,6,7,8,9]
>       revdig = [9..0]         = [9,8,7,6,5,4,3,2,1,0]
>       odddig = [1:9:2]        = [1,3,5,7,9]
>       revevendig = [8:0:-2]   = [8,6,4,2,0]
```

7.8.6 Simple Array Comprehensions

Comprehensions similar to Haskell list comprehensions are provided in FSC. A comprehension may use the `in` keyword to define a range:

```
>       [ x | x in A] = A
```

And filter elements using `when` and `unless`

```
>       odds  = [ x when x is odd     | x in [0..10]] = [1,3,5,7,9]
>       evens = [ x unless x is odd   | x in [0..10]] = [0,2,4,6,8]
```

Multiple arrays may be built via the use of commas

```
>       (odds,evens)= [x when x is odd,x unless x is odd| x in [0..10]]
>                   = ([1,3,5,7,9],[0,2,4,6,8])
```

Functions may be included when commas are used via the `of` keyword:

```
>       sum[x      | x in [1..10]] = [sum of x     | x in [1..10]] = 55
>       product[x | x in [1..10]] = [product of x| x in [1..10]] = 3628800
>       [sum of x,product of x | x in [1..10]] = (55,3628800)
```

The index of an iteration may be found using `at`:

```
>       [ y,A[i] | y in A at i] = (A,A)
```

Semicolons may be used to create arrays in blocks:

```
>       [ 10*x;x | x in [1..4]] = [10,1,20,2,30,3,40,4]
```

Loop ranges may be combined using `dot`,`cross` and ;

```
>       [ x + y | x in [1..4] dot y in [11..14]] = [1+11,2+12,3+13,4+14]

>       [ x + y | x in [1..4] cross y in [11..14]]
>          = [[1+11,1+12,1+13,1+14],
>          =  [2+11,2+12,2+13,2+14],
>          =  [3+11,3+12,3+13,3+14],
>          =  [4+11,4+12,4+13,4+14]]
```

```
>       [ x + y | x in [1..4] ; y in [11..14]]
>          = [1+11,1+12,1+13,1+14,
>          =  2+11,2+12,2+13,2+14,
>          =  3+11,3+12,3+13,3+14,
>          =  4+11,4+12,4+13,4+14]
```

7.8.7 Iterative Array comprehensions

Array comprehensions may also be defined iteratively from left to right using the `while/until` keywords and local bindings:

```
>       [ count | count <- count+1 until count == 10 | count = 0]
>         = [0,1,2,3,4,5,6,7,8,9]
>       [ count | count <- count+1 while count != 10 | count = 0]
>         = [0,1,2,3,4,5,6,7,8,9]
>       [ count | until count == 10 count <- count+1 | count = 0]
>         = [0,1,2,3,4,5,6,7,8,9,10]
>       [ count | while count != 10 count <- count+1 | count = 0]
>         = [0,1,2,3,4,5,6,7,8,9,10]
```

Multiple updates may be combined using semicolons:

```
>      [count1,count2 | count1 <- count1+1;
>                       count2 <- count2+2
>        until count1 == 10 | count1 = 0;
>                             count2 = 0]
```

This is equivalent to:

```
>      [count1,count2 | (count1,count2) <- (count1+1,count2+1)
>        until count1 == 10 | (count1,count2) = (0,0)]
```

7.8.8 Array declarations

Arrays may also be defined using the `array` keyword as follows:

```
>      array A[1..10] where A[i] = 10 - i
>          -- A  = [9,8,7,6,5,4,3,2,1,0]
>
>      array B[i] = A[i]
>          -- B  = [9,8,7,6,5,4,3,2,1,0]
>
>      array C[1..10] where
>                    C[1] = 10
>                    C[i] = C[i-1] - 1
>                 -- C = [10,9,8,7,6,5,4,3,2,1]
```

```
>      array D[1..10] where
>                    C[i] | i is even = 'e'
>                         | otherwise = 'o'
>                  -- D = ['o','e','o','e','o','e','o','e','o','e',]
>                       = "oeoeoeoeoe"
```

FSC arrays are discussed further in previous sections of this chapter and also in the next chapter.

7.9 Parametric Overloading

Quite apart from the fact that lazy evaluation is too inefficient to be used in practice, one of the major limitations of Haskell for use as a numerical programming language is the fact that the class system of Haskell is not powerful enough to express all the operator overloadings which are common in this scenario. For example scalar multiplication of matrices or vectors may not be expressed under the type class mechanism. Also, the method by which Haskell resolves the type of an expression is not intuitive.

Consider the class *Num* in the Haskell standard prelude. For the instance of *Num* over type α there exists a method to construct a value of type α from an integer and also a multiplication method to compute the product of two values of type α. If we define a data type `VECTOR` denoting vectors of length 4:

```
>      data VECTOR = V [Double]
```

and make this an instance of class *Num*:

```
>      instance Num VECTOR where
>       fromInt x = [fromInt x |_ <- [1..4]]
>       (V x) * (V y) = V (zipWith (*) x y)
```

and define the following expression

```
>      exp = (1 * 2) * v where v = V [1.0,2.0,3.0,4.0]
```

then it would be intuitive to imagine that the computation would proceed as

```
(1 * 2) * v -> 2 * v
        -> fromInt 2 * v
        -> V [2.0,4.0,6.0,8.0]
```

where actually the computation is

```
(1 * 2) * v -> (fromInt 1 * fromInt 2) * v
        -> (V [1.0,1.0,1.0,1.0] * V [2.0,2.0,2.0,2.0]) * v
        -> V [1.0 * 2.0,1.0 * 2.0,1.0 * 2.0,1.0 * 2.0] * v
        -> V [2.0,2.0,2.0,2.0] * v
        -> V [2.0,4.0,6.0,8.0]
```

and so eight floating-point multiplications would be carried out instead of four!

To coerce the desired behaviour we would have to write

```
>    exp = fromInt x * v
>    x = 1 * 2
```

In the following sections we discuss how type classes may be used and implemented in a more intuitive manner which suits numerical applications. We also discuss the problem of overlapping instance definitions which occurs when type classes are generalised.

7.9.1 Multi-Parameter Type Classes

The version of Haskell type classes proposed are similar to those of watML [72] in that multi-parameter templates are allowed in class definitions. For example, we can define a class `Plus(a,b,c)` containing the function `(+)::a->b->c` and, using this, define instances such as `(INT,REAL,REAL)` to allow true, mixed-mode arithmetic.

7.9.1.1 Efficiency

We do not wish to lose efficiency via this use of overloading and so the method that has been chosen to implement FSC polymorphism is what has been described as *textual polymorphism* [70], where a function has many types only at the source level. The advantages [70] of this technique are that

- it can produce optimum code for each application of the polymorphic procedure,
- it can support non-uniform representations of data, and
- it can support *ad-hoc* polymorphism as well as universal polymorphism.

The disadvantages are that it cannot support true first class polymorphic values since it is a monomorphic specialisation, and there is a potential for an exponential explosion in the number of specialised polymorphic forms. In each case we believe these drawbacks will not prove to be detrimental due to the type of programs written within our chosen problem domain. This view is backed up by results from Chapter 5 and also in [53] where the specialisation of type classes actually reduced the final executable's size.

The remaining sections describe the motivation behind the use of this type system, together with the presentation of an algorithm to implement it.

7.9.2 Use and Motivation

The simplest usage of multi-parameter overloading is as follows, where we provide overloadings for + so that it may work on both integers and floats :

```
>   class Plus (a,b,c) where
>     (+) a -> b -> c
```

```
>   instance Plus(INT,INT,INT) where
>    (+) = primintintplus
>   instance Plus(INT,Float,Float) where
>    (+) = primintfloatplus
   ...
```

here `primintfloatplus` and `primintintplus` are primitive monomorphic functions which add a float to an int and add an int to an int respectively. In the main[9] expression we could write

```
>    main = 2 + 3 + 1.5
```

and the type of `main` is `[+:INT->INT->a][+:a->REAL->b] b`. A unique satisfying substitution exists for the above constraints and so the overloading is resolved. This is completely different to Haskell where implicit `fromInteger` and defaults are used.

A slightly more complicated example is

```
>    f x y z = x + y + z

>    main = f 1 2 3
```

where the function `f` exports two constraints and the types of `main` and `f` are:

```
f    :: [+:a  ->b  ->c][+:c ->d   ->e] a -> b -> d -> e
main :: [+:INT->INT->a][+:a ->INT ->b] b
```

The function `f` can be seen to form an abstraction requiring five type variables $\{a, b, c, d, e\}$.

The motivation behind the use of this system can be understood by considering the following example.

7.9.3 Example

Consider the linear system **Ax=b** with **A** symmetric positive definite. It is possible to solve this system in various ways, one being the Conjugate Gradient Method (CGM), whose iteration can be described in a Haskell-like language as:

```
> dot x y    = sum (zipWith (*) x y)

> sum []     = fromInteger 0
> sum [x]    = x
> sum (x:xs) = x + sum x

> conjugate_gradient_iteration A (x,p,r) = (x',p',r')
>  where x'    = x + alpha * p
```

[9] In this discussion we ignore I/O without loss of generality.

```
>       p'    = r' + beta * p
>       r'    = r - alpha * q
>       alpha = rr / pq
>       beta  = (alpha * qq) / pq -1
>       pq    = p `dot` q
>       rr    = r `dot` r
>       qq    = q `dot` q
>       q     = A * p
```

Often we do not wish the matrix **A** to be formed explicitly as this is sometimes very computationally intensive. In such situations the CGM iteration above can be thought of as forming an abstraction. If we define an overloading of `(*)` such as function application `(*)::(a -> b) -> (a -> b)`, we would not explicitly form the matrix **A**, although our iteration algorithm would retain its elegance, i.e. in the above example the identifier **A** is just regarded as a data type with the method **(*)** defined on it. If `A` were a function `(*)` could represent application such that `(A*)` transformed vectors in the same manner as matrix multiplication by the matrix **A**.

7.9.4 Description of the Type Inference Algorithm

Informally the typechecker runs through a Hindley-Milner inference [20] phase and collects type constraints. These constraints are then resolved. The approach taken here is very similar to [72], the difference being that since we have already made the decision to specialise our polymorphic forms and overloaded identifiers (see Chapter 5), we may use a more liberal resolution algorithm than [72].

7.9.4.1 Example

As an example, consider:

```
> double x = x + x

> main = double 2
```

After type inference `double` and `main` are typed as

```
> double :: [+: a  -> a  -> b]  a  -> b
> main   :: [+:INT -> INT -> a] a.
```

an intermediate code is generated resembling an extended second order λ-calculus lifted into recursive supercombinator definitions i.e. type variables within a definition are tagged onto the end of the definition as extra parameters:

```
> double ::([+: a -> a -> b] a -> b)[a,b](x:a RETURNS b)
>          = (+)[a,a,b](x,x)

> main   ::([+:INT -> INT -> a] a)[a](RETURNS a)
>          = double[INT,a](2)
```

A unique solution for the constraints attributed to **main** ($[+ : INT \rightarrow INT \rightarrow \alpha]$) is then searched for. If found, the code is specialised[10]. The formal syntax for expressions and types is given in Figs. 7.7 and 7.8 with an extension to Robinson's unification algorithm [80] allowing contraint gathering summarised in Fig. 7.9, where χ is a constructor, v is a variable, $\mathcal{C}$ is a

Formal Syntax of Expressions			
Identifier	x		
Expression	e	::=	x
		|	$e_1\ e_2$
		|	$\lambda x.e$
		|	let $x = e_0$ in e_1
		|	if e_0 then e_1 else e_2

Figure 7.7: Formal syntax of expressions

Formal Syntax of Types			
Type Variables	α		
Type Constructors	χ		
Type Constraints	$\mathcal{C}$	::=	$\{x : \tau\}$
		|	$\mathcal{C}_0 \cup \mathcal{C}_1$
Types	τ	::=	α
		|	$\tau_0 \rightarrow \tau_1$
		|	$\chi(\tau_1, \ldots, \tau_n)$
		|	$\mathcal{C}_1, \ldots, \mathcal{C}_n\ \tau$
Type Schemes	σ	::=	$\forall\alpha.\sigma$
		|	τ

Figure 7.8: Formal syntax of types

constraint, and the s_i are arbitrary terms. $unify_{\theta,\mathcal{C}^*}(x, y)$ computes the most general unifier of x and y under θ and $\mathcal{C}^*$, together with the set of constraints that must be satisfied for x and y to be unified.

The typechecking rules written in terms of a set of assumptions A are given in Fig. 7.10, where overloaded instances are checked against their templates and all resulting constraints are collected.

[10] Providing the type of main is now monomorphic.

Unification	
$unify_{\theta C^*}(C_1^*\ \tau_1, C_2^*\ \tau_2)$	succeeds if $unify_{\theta,(C^* \cup C_1^* \cup C_2^*)}(\tau_1, \tau_2)$ succeeds, and fails otherwise
$unify_{\theta,C^*}(\tau_1, C_1^*\ \tau_2)$	succeeds if $unify_{\theta,(C^* \cup C_1^*)}(\tau_1, \tau_2)$ succeeds, and fails otherwise
$unify_{\theta,C^*}(C_1^*\ \tau_1, \tau_2)$	succeeds if $unify_{\theta,(C^* \cup C_2^*)}(\tau_1, \tau_2)$ succeeds, and fails otherwise
$unify_{\theta,C^*}(v_1, v_2)$	succeeds; v_1 and v_2 are bound to the same variable in θ
$unify_{\theta,C^*}(\chi_1[s_{11}, \ldots, s_{1m}], \chi_2[s_{21}, \ldots, s_{2n}])$	succeeds if $\chi_1 = \chi_2 \wedge m = n$ $\wedge unify_{\theta,C^*}(s_{11}, s_{21}) \wedge \ldots \wedge unify_{\theta,C^*}(s_{1m}, s_{2n})$, and fails otherwise
$unify_{\theta,C^*}(v, \chi[s_1, \ldots, s_n])$	succeeds binding v to $\chi[s_1, \ldots, s_n]$ in θ if v does not occur in $s_1 \ldots s_n$, and fails otherwise
$unify_{\theta,C^*}(\chi[s_1, \ldots, s_n], v)$	succeeds similarly

Figure 7.9: Extension to Robinson's unification algorithm

Type rules		
TAUT	$A.x : \tau \vdash x : \tau$	
COND	$\dfrac{A \vdash e_0 : C_0^* bool \quad A \vdash e_1 : C_1^* \tau \quad A \vdash e_2 : C_2^* \tau}{A \vdash (if\ e_0\ then\ e_1\ else\ e_2) : (C_0^* \cup C_1^* \cup C_2^*)\tau}$	
ABS	$\dfrac{A.x : \sigma \vdash e : C^* \tau}{A \vdash (\lambda x.e) : C^* \sigma \rightarrow \tau}$	
LET	$\dfrac{A \vdash e_0 : \sigma \quad A.x : \sigma \vdash e_1 : \tau}{A \vdash (let\ x = e_0\ in\ e_1) : \tau}$	
GEN	$\dfrac{A \vdash e : \tau}{A \vdash e : \forall \alpha.\tau}$	(α not free in A)
SPEC	$\dfrac{A \vdash a : \forall \alpha.\tau}{A \vdash e : [\sigma/\alpha]\,\tau}$	
COMB	$\dfrac{A \vdash e_0 : C_0^* \sigma \rightarrow \tau \quad A \vdash e_1 : C_1^* \sigma}{A \vdash (e_0\ e_1) : (C_0^* \cup C_1^*)\tau}$	

Figure 7.10: Type inference and constraint-gathering rules

7.9.5 Overloading Resolution

The overloading resolution phase attempts to find a unique satisfying substitution of type instances over the type constraints resulting from type inference. This is described in Fig. 7.11 where I is

Overloading Resolution

$$RESOLVE(I,\{C_1,\ldots,C_n\}) = SEARCH(I,MATCHES(I,\{\},C_1),\{C_2,\ldots,C_n\})$$

$$MATCHES(I,\theta,C) = \{unify_{\theta,\{\}}(i_{type},C_{type}) \mid i \in I \wedge unify_{\theta,\{\}}(i_{type},C_{type}) \neq \bot \wedge C_{name} = i_{name}\}$$

$$SEARCH(I,P,C^*) = \begin{cases} \bot, & \text{if } P = \{\} \\ OUT(\{DERIVATION(p_i,I,C^*) | p_i \in P\}), & \text{otherwise} \end{cases}$$

$$OUT(\{x\}) = x$$
$$OUT(\{\}) = \bot$$
$$OUT(\{x_1,\ldots,x_n\}) = \bot$$

$$DERIVATION((\theta,\{\}),I,\{\}) = (\theta,\{\})$$
$$DERIVATION((\theta,\{\}),I,\{C_1,\ldots,C_n\}) = SEARCH(I,MATCHES(I,\theta,\theta(C_1)),\{C_2,\ldots,C_n\})$$
$$DERIVATION((\theta,C_1^*),I,C_2^*) = DERIVATION((\theta,\{\},I,C_1^* \cup C_2^*)$$

Figure 7.11: Overloading resolution

the set of instances, θ are substitutions, C are constraints and the P are possible matches. In our description X^* denotes a set of X's. All constraints are taken to be name-type pairs; for example, if $C = \{+ : INT \rightarrow INT \rightarrow INT\}$ then $C_{name} = +$ and $C_{type} = INT \rightarrow INT \rightarrow INT$. In the FSC implementation the typechecker is more detailed than suggested by the above, allowing such features as pattern matching and case statements.

7.10 Overlapping Type Class Instances

The type resolution algorithm described in the previous section is designed to operate on a set of type-class instances which do not overlap as overlapping instances are cannot be dealt with trivially. Overlapping may be understood by considering the set of types which match a given pattern. If we denote the patterns of a set of instances by

$$P_1, P_2, \ldots, P_n$$

and the set of types which matches P_i by $S_i, 1 \leq i \leq n$, then a set of non-overlapping instances is characterised by the fact that no two patterns match the same type, i.e.

$$S_i \cap S_j = \emptyset, 1 \leq i,j \leq n, \qquad i \neq j$$

The following set of patterns is non-overlapping

$$\{Real \rightarrow Real \rightarrow Real, Int \rightarrow Int \rightarrow Int\}$$

7.10.1 Specialisation

A pattern P_i is said to be a *specialisation* of a pattern P_j (or P_i is *more specific* than P_j) if each expression which matches P_i also matches P_j but not vice versa; that is, if S_i is a proper subset of S_j ($S_i \subset S_j$). Thus the type *Real* is a specialisation of the type variable τ but the type variable σ is not.

If we partially order our instances according to how specific they are, we may construct guards against patterns being matched where there exists a more specific pattern. That is, we order the patterns under the relation

$$P_i \succeq P_j \equiv S_i \subseteq S_j$$

Each pattern P_i is now replaced by P_i^* such that the set of types which match P_i^* is

$$S_i - \bigcup \{S_j | P_j \not\preceq P_i\}$$

7.10.1.1 Example

If we consider overlapping multiplications, the class `TIMES` could have the following instances, where `[a]` denotes a vector of element-type `a`:

```
>    instance TIMES(a,b,c) => Times(a,[b],[c]) ...
>    instance TIMES(a,b,c) => Times([a],b,[c]) ...
>    instance TIMES(a,b,c),PLUS(c,c,c) => Times([a],[b],c)...
```

representing right and left scalar multiplications and the dot product of two vectors.

Without paying attention to overlapping instances, typing the expression

```
>        x * y
>          where
>            x = [1,2,3]
>            y = [4,5,6]
```

would give rise to the constraint $\{* : [Int] \rightarrow [Int] \rightarrow \alpha\}$ which matches all of the above instances, resulting in ambiguity. The above instances compute xy^T, $(xy^T)^T$ and x^Ty respectively.

However, if we add constraints to these instances so that they reject types accepted by instances more specific or incomparable to themselves, then we find that this ambiguity disappears and the only instance which matches the expression above is x^Ty. This addition of constraints is the conversion of each P_i to P_i^* mentioned earlier.

This makes the above set of instances unambiguous. In general, to ensure that a set of patterns is unambiguous then the set must be (possibly) extended so that its members satisfy the following condition:

> **Unambiguity:** If any two instances overlap and neither is a specialization of the other then, for each type τ which matches each instance, there must be a third pattern which is the specialization of both instances that also matches τ.

In effect the above method is equivalent to creating this *third* pattern which automatically fails

due to an unresolvable sub-constraint. This can also be seen as forming a lattice from the poset of matchings by embedding each set of non-comparable instances inside a lattice by adding a least upper bound (l.u.b.) drawn from the set of all patterns. $\top$ represents the pattern with no matchings, i.e. the l.u.b. of the set of all patterns and the class template itself is the greatest lower bound (g.l.b.). The set of types accepted by an instance is then restricted to the set of types it accepted originally, less the union of the sets of types originally accepted by all its least upper bounds proper[11].

7.10.1.2 Example I

To visualise this procedure, consider the *MULT* class with overloadings for integer multiplication, dot products and scalar multiplication (Fig. 7.12) where α, β and γ are implicitly universally qualified. The act of adding extra *dummy* instances so that our unambiguity condition may be satisfied is also shown in Fig. 7.12.

7.10.1.3 Example II

A larger set of multiplication instances is shown in Fig. 7.13, and its disambiguated version shown in Fig. 7.14. Again α, β and γ are implicitly universally qualified.

7.10.2 Overloading Resolution under Overlapping Instances

A version of overloading resolution is given in Fig. 7.15, where I is the set of instances, θ are substitutions, $\mathcal{C}$ are constraints and the P are possible matches, X^* denotes a set of X's. All constraints are taken to be name, type and side-condition triples; for example, if $\mathcal{C} = \{* : \alpha \rightarrow \beta \rightarrow \gamma : \alpha \neq Int\}$ then $\mathcal{C}_{name} = *$, $\mathcal{C}_{type} = \alpha \rightarrow \beta \rightarrow \gamma$ and $\mathcal{C}_{SC} = \{(\alpha, Int)\}$.

7.11 Summary

In this chapter we discussed the rationale behind the features which appear in FSC, our prototype language. We are now in a position to present a definition of the FSC' language proper. This is the subject of the following chapter. **Note:** FSC' contains a transformational meta-language which allows program transformation. Discussion of this meta-language is left to Chapter 9 as it is built strictly above the core language.

[11] We define upper bound proper (u.b.p.) of an element e as an upper bound of e which is not equal to e.

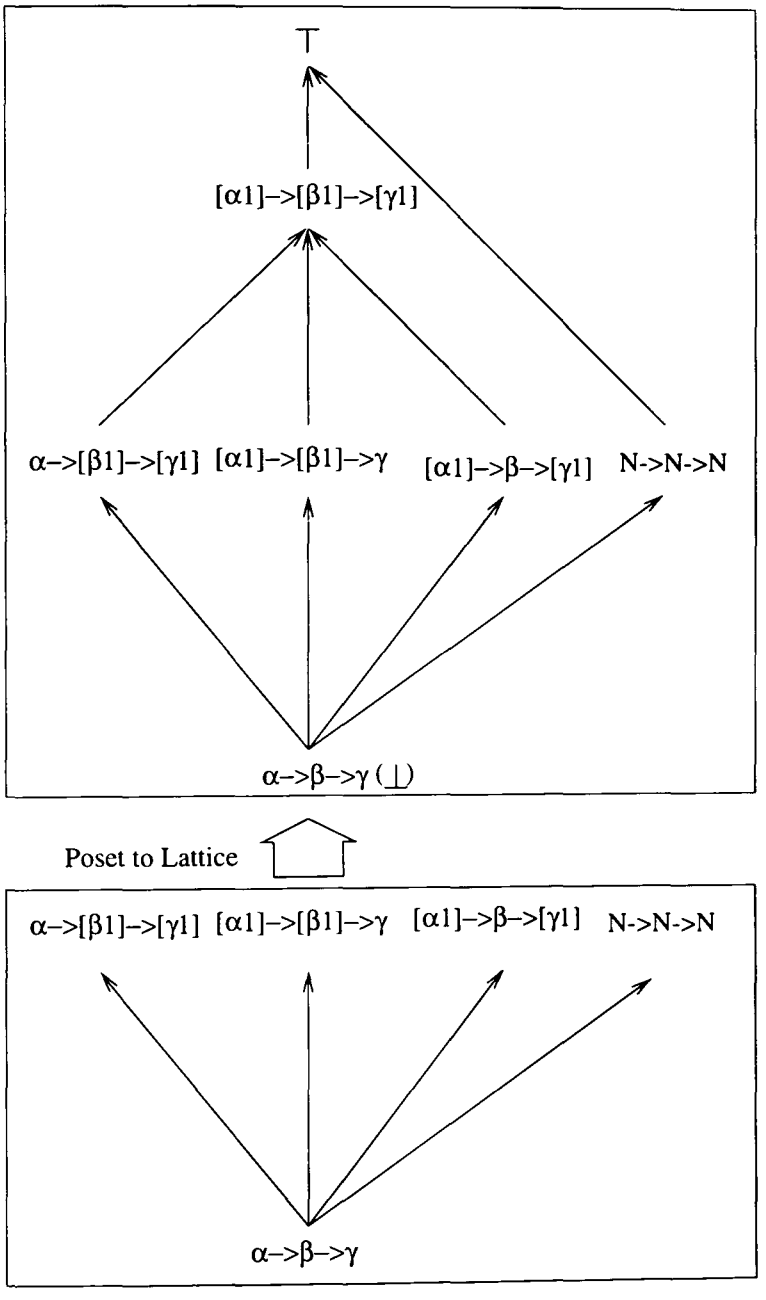


Figure 7.12: Embedding each non-comparable subset in a lattice

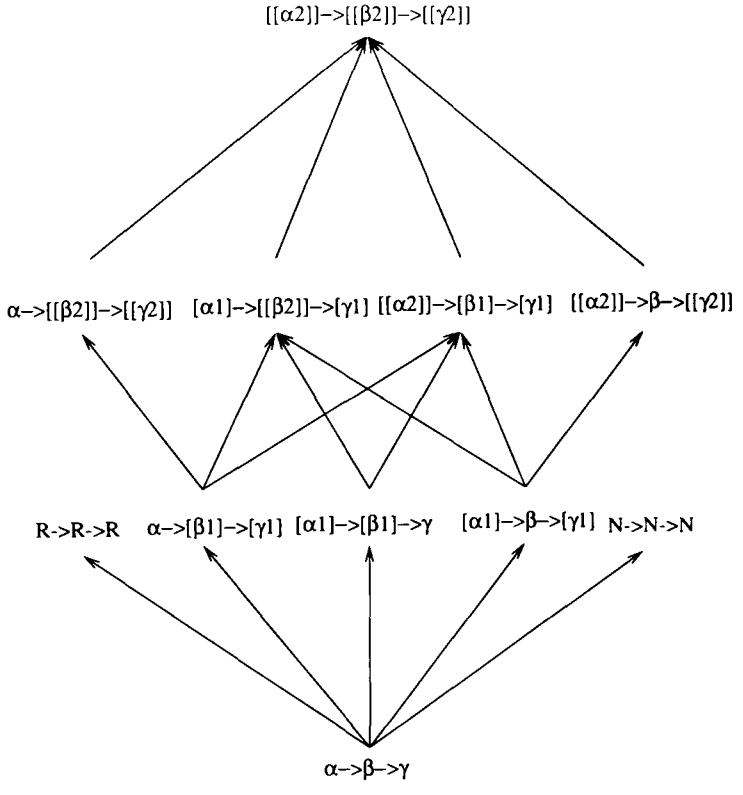


Figure 7.13: Partially-ordered set of instances for class MULT

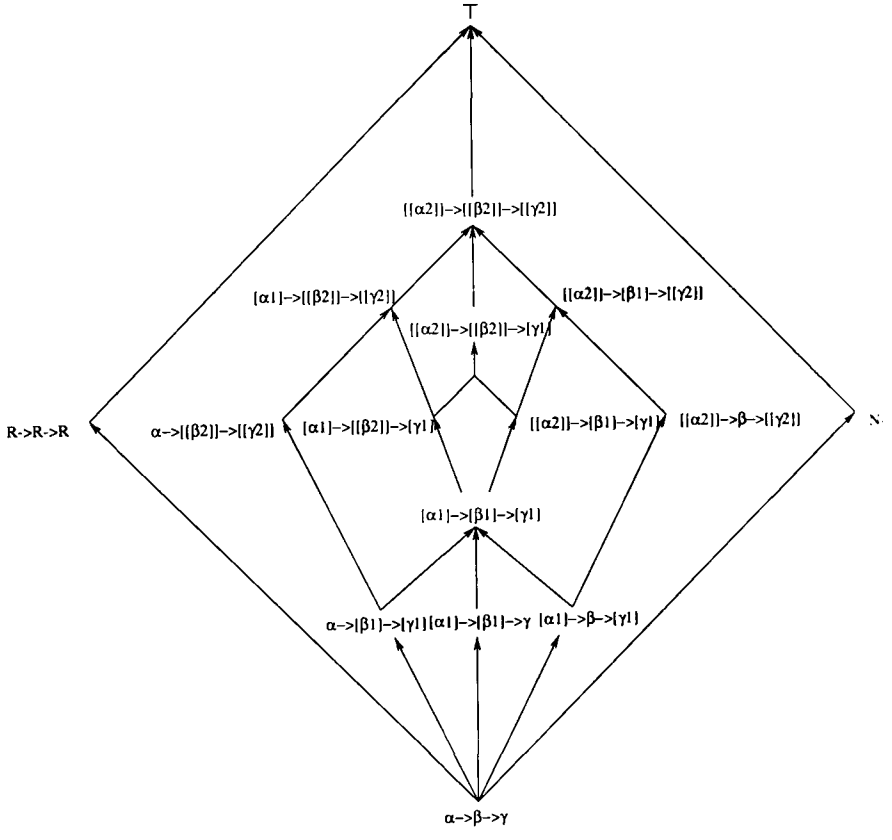


Figure 7.14: Lattice of instances for class MULT

Overloading Resolution with Overlapping Instances

$$RESOLVE(I,\{C_1,\ldots,C_n\}) = SEARCH(I, MATCHES(I,\{\},C_1,\{\}),\{C_2,\ldots,C_n\})$$

$$MATCHES(I,\theta,C,\mathcal{SC})$$
$$= \{(unify_{\theta,\{\}}(i_{type},C_{type}), f(i_{SC} \cup \mathcal{SC}) \mid i \in I \wedge unify_{\theta,\{\}}(i_{type},C_{type}) \neq \bot \wedge C_{name} = i_{name}\}$$
$$\text{where } f(\mathcal{SC}) = \{(unify_{\theta,\{\}}(i_{type},C_{type})(\sigma),\tau) \mid (\sigma,\tau) \in \mathcal{SC}\}$$

$$SEARCH(I,P,\mathcal{C}^*) = \begin{cases} \bot, & \text{if } P = \{\} \\ RESTRICT(\{DERIVATION(p_i,I,\mathcal{C}^*) \mid p_i \in P\}), & \text{otherwise} \end{cases}$$

$$RESTRICT(X) = \{\theta \mid ((\theta,\{\}),\mathcal{SC}) \in X, allowed(\theta,\mathcal{SC})\}$$
$$\text{where } allowed(\theta,\mathcal{SC}) = \bigwedge\{unify_{\theta,\{\}}(\sigma,\tau) = \bot \mid (\sigma,\tau) \in \mathcal{SC}\}$$

$$OUT(\{x\}) = x$$
$$OUT(\{\}) = \bot$$
$$OUT(\{x_1,\ldots,x_n\}) = \bot$$

$$DERIVATION(((\theta,\{\}),\mathcal{SC}),I,\{\}) = (\theta,\{\},\mathcal{SC})$$
$$DERIVATION(((\theta,\{\}),\mathcal{SC}),I,\{C_1,\ldots,C_n\})$$
$$= SEARCH(I, MATCHES(I,\theta,\theta(C_1),\mathcal{SC}),\{C_2,\ldots,C_n\})$$
$$DERIVATION(((\theta,C_1^*),\mathcal{SC}),I,C_2^*) = DERIVATION(((\theta,\{\}),\mathcal{SC}),I,C_1^* \cup C_2^*)$$

Figure 7.15: Overloading resolution with overlapping instances

Chapter 8

Definition of FSC

In this chapter we present a definition of the FSC programming language.

8.1 Introduction

FSC is a purely functional language incorporating many recent innovations in programming language research, including higher order functions, static polymorphic typing, user defined algebraic datatypes, pattern matching and a rich set of primitives. The design of this language has been heavily influenced by languages from the non-strict functional programming world such as Haskell, languages from the strict functional programming world such as ML, and also languages from the single assignment world such as SISAL (Fig. 8.1).

The emphasis of the FSC design is to provide an extremely efficient functional language with which to develop numerical software in a manner which best suits the problem domain. To this end, some of the ideas incorporated in FSC are either very experimental or do not possess the completeness[1] of similar concepts in languages such as Haskell. We do not consider this a failing, but merely recognise that we have a separate set of goals for which we consider efficiency paramount and try to embrace as much of the *flavour* of functional programming as possible.

Although FSC builds heavily on previous ideas it also has several innovative features which we consider useful in the area of numerical programming:

1. FSC extends Haskell-style overloading to cover typical linear algebra operations using techniques which do not create slower code through the use of overloading.

2. True arrays are provided as a primitive datatype, the novel feature being the combination of pattern matching and array operations.

3. A term rewriting system allows axioms to be expressed in FSC and code improving derivations to be written.

[1] An example of FSC's lack of syntactic completeness is the fact that `A[i]` $\neq$ `A([i])`.

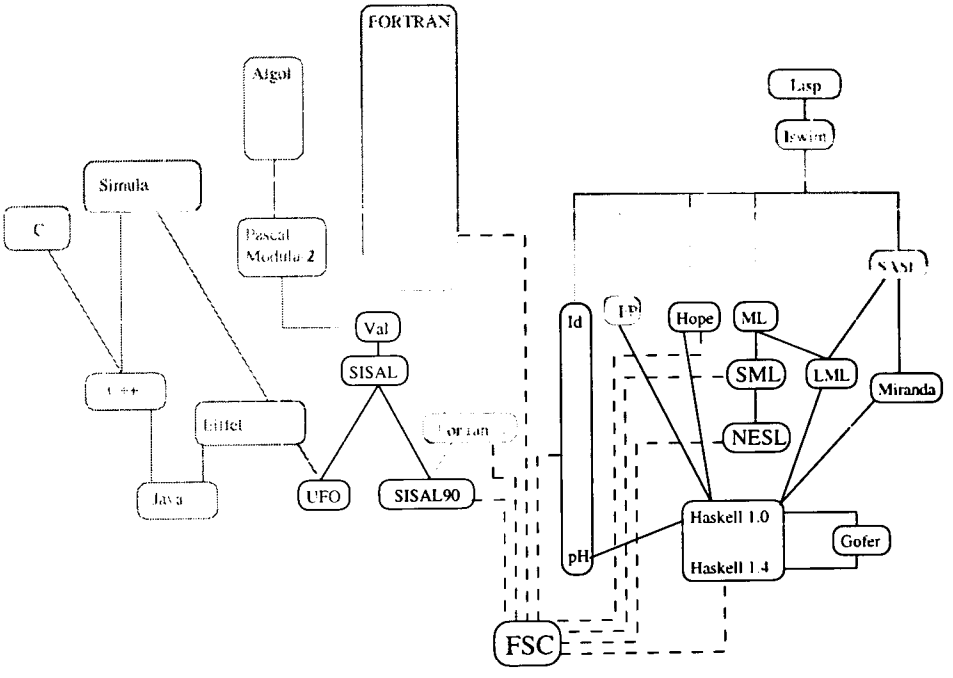


Figure 8.1: Language spectrum and lineage of FSC

4. A purely functional I/O facility is provided which unifies the I/O models of Haskell 1.3 and SISAL.

This chapter defines the syntax of FSC programs and an informal abstract semantics for the meaning of such programs. We leave as implementation dependent the ways in which FSC programs are to be compiled other than defining criteria on efficiency requirements which must be respected. This includes issues such as guaranteeing that the use of an overloaded function will not result in an increased execution time as it does in the implementation of languages such as Haskell. The structure, language and commentary of this chapter unashamedly borrows heavily from the Haskell 1.2 report [50] since our aim is to mimic Haskell syntactically.

8.2 Program Structure

In this section we describe the abstract syntactic and semantic structure of FSC, as well as how it relates to the organisation of the rest of this chapter.

1. At the topmost level an FSC program is a set of *modules*. Modules provide a way to control namespaces and to re-use software in large programs.
2. The uppermost level of an FSC module consists of a collection of *declarations*, of which there are several kinds. Declarations define things such as values, datatypes, type classes, fixity information and rewrite rules.
3. At the next level are *expressions*. An expression denotes a *value* and has a *static type*.
4. At the lowest level is FSC's *lexical structure*. The lexical structure captures the concrete representation of FSC scripts in text files.

Examples of FSC fragments in running text are given in typewriter font as in:

```
> let x = 1
>     z = x+y
> in  z+1
```

8.2.1 The FSC Kernel

FSC has many of the common syntactic structures that are popular in functional programming. In these cases we give a translation into a subset of FSC which we call the *FSC kernel*. This fulfils a similar role to the Haskell kernel in that we can provide a straightforward denotational semantics for the kernel and a translation of each syntactic structure into the kernel. This allows the presentation of the language to be simplified.

Program	*prog*	→	*binds*	
Bindings	*binds*	→	$bind_1; \ldots; bind_n$	$n \geq 1$
	bind	→	*var* = *expr*	
Expression	*expr*	→	$expr_1\ expr_2$	Application
		|	$\lambda\ var\ .\ expr$	Lambda abstraction
		|	case *expr* of *alts*	Case expression
		|	if $expr_1$ then $expr_2$ else $expr_3$	Conditional
		|	let *bind* in *expr*	Local definition
		|	letrec *binds* in *expr*	Local recursion
		|	*con*	Constructor
		|	*var*	Variable
		|	*literal*	
Literal values	*literal*	→	*Integer*	
		|	*Float*	
		|	*Character*	
		|	*Double*	
		|	*Boolean*	
Alternatives	*alts*	→	$calt_1; \ldots; calt_n$	$n \geq 1$
Constructor alt	*calt*	→	$con\ var_1 \ldots\ var_n$ -> *expr*	$n \geq 0$

Figure 8.2: Syntax of the FSC kernel

8.2.1.1 FSC Kernel Syntax, Semantics and Intrinsic Operations

The syntax and semantics of the FSC kernel are shown in Fig. 8.2 and 8.3 respectively. The FSC language contains true arrays and, as such, we regard these as primitive with operations shown in Figs. 8.4 and 8.5. Other primitive, hard-wired, functions exist over the ground types {*INT*, *REAL*, *DOUBLE*, *CHAR*, *BOOL*} such as

```
>          _primIntIntEqual :: INT -> INT -> BOOL
```

8.2.2 Expressions and Types

An expression evaluates to a *value* and has a static *type*. Types are not first order in FSC although the type system allows user-defined datatypes and permits parametric polymorphism and *ad-hoc* polymorphism (using *type classes*). Because of FSC's standpoint of efficiency, not all polymorphic values are allowed in FSC[2].

[2] All values must have a set of static monomorphic resolutions and polymorphic objects such as `a = hd []` are disallowed.

$\mathcal{P}[\![program]\!] : Val$
$\mathcal{P}[\![prog]\!] = \mathsf{EVAL}[\![\ \texttt{letrec}\ prog\ \texttt{in main}\ ]\!]\ \rho_{init}$

$\mathsf{EVAL}[\![expr]\!] : Env \rightarrow Val$

$$
\begin{array}{lcl}
\mathsf{EVAL}[\![k]\!]\ \rho & = & \mathcal{K}[\![k]\!] \\
\mathsf{EVAL}[\![x]\!]\ \rho & = & \rho\ x \\
\mathsf{EVAL}[\![e_1\ e_2]\!]\ \rho & = & (\mathsf{EVAL}[\![e_1]\!]\ \rho)\ (\mathsf{EVAL}[\![e_2]\!]\ \rho) \\
\mathsf{EVAL}[\![\lambda x.e]\!]\ \rho & = & strict(\lambda x_{new}.\mathsf{EVAL}[\![e]\!]\ (\rho \dotplus x \mapsto x_{new})) \\
\mathsf{EVAL}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\ \rho & = & \mathsf{EVAL}[\![\texttt{case}\ e_1\ \texttt{of True ->}\ e_2\texttt{;} \\
 & & \quad \texttt{False ->}\ e_3]\!]\ \rho \\
\mathsf{EVAL}[\![\texttt{let}\ x = e\ \texttt{in}\ b\ ]\!]\ \rho & = & \mathsf{EVAL}[\![(\lambda x.b)\ e]\!]\ \rho \\
\mathsf{EVAL}[\![\texttt{letrec}\ binds\ \texttt{in}\ e\ ]\!]\ \rho & = & \mathsf{EVAL}[\![e]\!]\ (\rho \oplus fix(\lambda\rho'.\mathcal{B}[\![binds]\!](\rho \dotplus \rho'))) \\
\mathsf{EVAL}[\![c]\!]\ \rho & = & \lambda\epsilon_1 \ldots \lambda\epsilon_n.\langle c, \epsilon_1, \ldots, \epsilon_n\rangle
\end{array}
$$

$\mathsf{EVAL}[\![\texttt{case}\ x\ \texttt{of}\ c_1\ x_{11} \ldots x_{1a_1}\texttt{->}\ e_1\texttt{;} \ldots \texttt{;}\ c_n\ x_{n1} \ldots x_{na_n}\texttt{->}\ e_n]\!]\ \rho$

$$
\begin{array}{lcl}
= \ \textbf{case}\ \mathsf{EVAL}[\![e]\!]\ \rho\ \textbf{of} & & \\
\quad [\!]\ \bot & \rightarrow & \bot \\
\quad [\!]\ \langle c_1, \epsilon_{11}, \ldots, \epsilon_{1a_1}\rangle & \rightarrow & \mathsf{EVAL}[\![e_1]\!]\ (\rho \oplus \{x_{11} \mapsto \epsilon_{11}, \ldots, x_{1a_1} \mapsto \epsilon_{1a_1}\}) \\
\quad [\!]\ \ldots & & \\
\quad [\!]\ \langle c_n, \epsilon_{n1}, \ldots, \epsilon_{na_n}\rangle & \rightarrow & \mathsf{EVAL}[\![e_n]\!]\ (\rho \oplus \{x_{n1} \mapsto \epsilon_{n1}, \ldots, x_{na_n} \mapsto \epsilon_{na_n}\}) \\
\quad \textbf{end} & &
\end{array}
$$

$\mathcal{B}[\![binds]\!] : Env \rightarrow Env$

$$
\mathcal{B}[\![x_1 = e_1; \ldots; x_n = e_n]\!]\ \rho \quad = \quad \{x_i \mapsto \mathsf{EVAL}[\![e_i]\!]\ \rho | i \in \{1, \ldots, n\}\}
$$

$strict : (\ Val \rightarrow Val\ ) \rightarrow Val \rightarrow Val$
$strict\ f\ x = f\ x$, if $x \neq \bot$
$strict\ f\ \bot = \bot$

Figure 8.3: Denotational semantics of the FSC kernel

_access	::	*Array* α	→	*Int*	→	α		
_cons	::	α	→	*Array* α	→	*Array* α		
_snoc	::	α	→	*Array* α	→	*Array* α		
_head	::	*Array* α	→	α				
_tail	::	*Array* α	→	*Array* α				
_nil	::	*Array* α						
_last	::	*Array* α	→	α				
_init	::	*Array* α	→	*Array* α				
_append	::	*Array* α	→	*Array* α	→	*Array* α		
_replace	::	*Array* α	→	*Int*	→	α	→	*Array* α
_fill	::	*Int*	→	*Int*	→	α	→	*Array* α
_size	::	*Array* α	→	*Int*				
_iml	::	*Array* α	→	*Int*				
_imh	::	*Array* α	→	*Int*				
_subarray	::	*Array* α	→	*Int*	→	*Int*	→	*Array* α
_setindex	::	*Array* α	→	*Int*	→	*Array* α		
_isEmpty	::	*Array* α	→	*Bool*				

Figure 8.4: Intrinsic array operations

8.2.3 Namespaces

The distinctions between namespaces for variables and constructors are not disjoint. This is to allow users to denote matrices using upper-case identifiers. There are the following constraints on naming:

- Constructors and type constructors begin with upper-case letters; type variables begin with lower-case letters.
- Type class names begin with upper-case letters.
- Operators may be any valid identifier or any string of characters excluding the semicolon, comma, underscore, brace and quote characters.

8.2.4 Layout

FSC permits the use of a layout (or "off-side") rule in the style of Haskell. For example

```
>          let
>              x = 1
>              y = 2
>          in
>              x + y
```

may be written in place of

```
>          let {x = 1; y = 2} in (x+y)
```

For a more in depth discussion of this layout convention see [50]. In this chapter, we do not use the offside rule as it is a mere preprocessor phase and hence of little interest.

$$\_access\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, j \mapsto e_j, \cdots, n \mapsto e_n]\ j = e_j$$

$$\_cons\ a\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n] = [(i-1) \mapsto a, i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n]$$

$$\_snoc\ a\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n] = [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n, (n+1) \mapsto a]$$

$$\_head\ [i \mapsto e_i, \cdots, n \mapsto e_n] = e_i$$

$$\_tail\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n] = [(i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n]$$

$$\_nil = []$$

$$\_last\ [i \mapsto e_i, \cdots, n \mapsto e_n] = e_n$$

$$\_init\ [i \mapsto e_i, \cdots, (n-1) \mapsto e_{i+1}, n \mapsto e_n] = [i \mapsto e_i, \cdots, (n-1) \mapsto e_{n-1}]$$

$$\_append\ [i \mapsto e_i, \cdots, n \mapsto e_n]\ [j \mapsto e_j, \cdots, m \mapsto e_m] = [i \mapsto e_i, \cdots, n \mapsto e_n, (n+1) \mapsto e_j, \cdots, ((m-j)+n+1) \mapsto e_n]$$

$$\_replace\ [i \mapsto e_i, \cdots, j \mapsto e_j, \cdots, n \mapsto e_n]\ j\ a = [i \mapsto e_i, \cdots, j \mapsto a, \cdots, n \mapsto e_n]$$

$$\_fill\ i\ n\ a = [i \mapsto a, \cdots, \cdots, n \mapsto a]$$

$$\_liml\ [i \mapsto e_i, \cdots, n \mapsto e_n] = i$$

$$\_limh\ [i \mapsto e_i, \cdots, n \mapsto e_n] = n$$

$$\_size\ [i \mapsto e_i, \cdots, n \mapsto e_n] = (n-i)+1$$

$$\_subarray\ [i \mapsto e_i, \cdots, j \mapsto e_j, \cdots, k \mapsto e_k, \cdots, n \mapsto e_n]\ j\ k = [j \mapsto e_j, \cdots, k \mapsto e_k]$$

$$\_setindex\ [i \mapsto e_i, \cdots, n \mapsto e_n]\ j = [j \mapsto e_i, \cdots, (n-i)+j \mapsto e_n]$$

$$\_isEmpty\ A = \_primIntIntEqual\ (\_size\ A)\ 0$$

Figure 8.5: Intrinsic array operation identities

8.3 Lexical Structure

In this section we describe the low level lexical structure of FSC. Many of the details may be skipped on a first reading.

8.3.1 Notational Conventions

The following notational conventions are used for presenting syntax:

$[pattern]$	optional
$\{pattern\}$	zero or more repetitions
$(pattern)$	grouping
$pat_1 \mid pat_2$	choice
$pat_{pat'}$	difference-elements generated by *pat* except those generated by *pat'*
`factorial`	terminal syntax in typewriter font

Because the syntax in this section describes *lexical* syntax, all white space is expressed explicitly; there is no implicit space between juxtaposed symbols. BNF-like syntax is used throughout, with productions having the form:

$$nonterm \longrightarrow alt_1 \mid alt_2 \mid \ldots \mid alt_n$$

Care must be taken in distinguishing metalogical syntax such as | and [...] from concrete terminal syntax (given in typewriter font) such as `|` and `[...]`, although usually the context makes the distinction clear.

8.3.1.1 Lexical Program Structure

program → $\{$*lexeme*|*whitespace*$\}$
lexeme → *tyid*|*conid*|*varsym*|*consym*|*literal*|*special*|*reservedop*|*reservedid*
literal → *integer*|*float*|*double*|*character*|*string*|*boolean*
special → `(|)|,|;|[|]|_|'|{|}`

whitespace → *whitestuff*$\{$*whitestuff*$\}$
whitestuff → *whitechar*|*comment*
whitechar → *space*|*newline*|*tab*
comment → `--`$\{$*any*$\}$*newline*
any → *graphic*|*space*|*tab*
graphic → *large*|*small*|*digit*
| `!|"|#|$|%|&|'|(|)|*|+`
| `,|-|.|/|:|;|<|=|>|?|@`
| `[|\|]|^|_|'|{|}|~`

small → `a|b|...|z`
large → `A|B|...|Z`
digit → `0|1|...|9`

Characters not in the category *graphic* or *whitestuff* are not valid and should result in a lexing error.

8.3.2 Identifiers and Operators

```
tyid       → (small{small|large|digit|'|_})reservedid
varid      → tyid|conid
conid      → large{small|large|digit|'|_}
reservedid → all|array|at|by|case|class|cross|datatype|do|dot|else
           | for|if|ininfix|infixr|infixl|instance|let|of|ordered
           | prefix|repeat|return|returns|suffix|then|type|with|when
           | where|until|unless
```

An identifier consists of a letter followed by zero or more letters, digits, underscores, and acute accents. Identifiers are lexically divided into two classes: those beginning with a lower-case letter (type variable identifiers) and those beginning with an upper-case letter (constructor identifiers).

```
varsym     → {symbol}reservedop
symbol     → !|#|$|%|&|*|+|.|/|<|=|>|^|?|@|\|~|`
reservedop → ..|::|=>|=|@|\|||<-|->|:
```

Operator symbols are formed from one or more symbol characters, as defined above.

8.3.3 Boolean Literals

```
boolean → True|False
```

8.3.4 Numeric Literals

There are three distinct kinds of numeric literals: *integer*, *float* and *double*.

```
integer → digit{digit}
float   → integer.integer[(e|E)[~|+]integer]
        | integer(e|E)[~|+]integer
        | integer.
double  → integer.integer[(d|D)[~|+]integer]
        | integer(d|D)[~|+]integer
```

8.3.5 Character and String Literals

A character literal is written between acute accents and a string literal between double quotes. The use of backslash characters for newline, tab, etc. is identical to that of the C programming language.

String literals are actually abbreviations for arrays of characters.

8.4 Expressions

In this section, we describe the syntax and semantics of FSC *expressions*, including their translation into the FSC kernel where appropriate.

exp	→	*aexp* :: *type*	(expression type signature)
	|	exp_0	
exp_0	→	`let` {*decls*[;]} `in` *exp*	(let expression)
	|	`\`$apat_1$ ... $apat_n$`->` *exp*	(lambda abstraction $n \geq 1$)
	|	`if` *exp* `then` *exp* `else` *exp*	(conditional)
	|	`case` *exp* `of` {*alts*[;]}	(case expression)
	|	`do` *type computations*	(do (IO) expressions)
	|	*iteration*	(iteration/array expressions)
	|	*fexp*	
fexp	→	*fexp aexp*	(function application)
	|	*fexp* $[args_{array}]$	(array access)
	|	*fexp* $\{args_{exp}\}$	(bracketed application)
	|	*aexp*	
aexp	→	*var*	(variable)
	|	*con*	(constructor)
	|	*literal*	
	|	()	(unit)
	|	(*exp*)	(parenthesised expression)
	|	$(exp_1, \ldots, exp_k)$	(tuple $k \geq 2$)

The FSC grammar is simplified by handling the parsing of operators from outside the grammar. Table 8.1 shows how examples are initially parsed and then transformed to prefix expressions.

The expression	parses as	and is transformed to
`f x + g y`	`((((f x) +) g) y)`	`(+ (f x)) (g y)`
`n!`	`(n !)`	`(! n)`
`(x+)`	`BRACK(x +)`	`(+ x)`

Table 8.1: Operator parsing

8.4.1 Variables, Constructors and Operators

var	→	*varid* | *varsym* |(*op*)	(variable)
con	→	*conid* | *consym*	(constructor)
consym	→	*varsym*	
op	→	*varid* | *varsym* | `` `varid` ``	(operator)

Alphanumeric operators are constructed by either declaring the operator as `infix` in a fixity declaration, or by enclosing it between grave accents (backquotes). In this way *fun x y*. with *fun* a prefix function, is equivalent to *x* `` `fun` `` *y* .

Similarly, any non-prefix identifier may be used as a (Curried) variable by enclosing it in parentheses. If *op* is an infix or suffix operator then an expression or pattern of the form *x op y* is equivalent to (*op*) *x y*.

8.4.2 Curried Applications and Lambda Abstractions

$$\begin{array}{lcl} exp & \rightarrow & \backslash apat_1 \ \ldots \ apat_n \texttt{ -> } exp \\ fexp & \rightarrow & fexp\ aexp \\ & | & fexp\ \{args_{exp}\} \end{array}$$

Function application is written $e_1\ e_2$. Application associates to the left and so the parentheses may be omitted in `(f x) y`, for example. The second form of function application uses the braces {,} and comma separators such that the expression

```
> f {x,y,z}
```

is equivalent to

```
> f x y z
```

The advantage of this use of braces is that a function may be partially applied to any combination of its arguments via the use of underscores. That is, the expression

```
> f {x,_,y,_,_,z}
```

is equivalent to

```
> (\ p q r -> f {x,p,y,q,r,z})
```

Note: Underscores may not be used without braces, i.e. the application

```
> f x _ z
```

is invalid. Such partial applications are disallowed since

```
> f x _ y
```

would have a very different meaning to

```
> (((f x) _ ) y) = ((\z -> (f x) z) y) = f x y
```

i.e. it is not compatible with left association of function application.

8.4.2.1 Lambda Abstractions

Lambda abstractions are written $\backslash p_1 \cdots p_n$ -> e, where the p_i are *patterns*(Fig. 8.6). If any pattern contains a constructor with *arity* > 0 then it must be surrounded by parentheses. FSC requires these patterns to be *linear*, i.e. with no variables appearing more than once in the set. Unlike Haskell, admitting non-linear patterns would not cause semantic problems [48] in FSC. The reason for not allowing non-linear patterns is that it could potentially cloud computationally expensive expressions.

Translation:	The lambda abstraction \ $p_1 \cdots p_n$ -> e is equivalent to \ $x_1 \cdots x_n$ -> case $(x_1, \cdots, x_n)$ of $(p_1, \cdots, p_n)$ -> e where the x_i are new identifiers. If the pattern fails the result is $\perp$.

Figure 8.6: Translation rule for pattern-matching lambda expressions

8.4.3 Operator Applications

$$\begin{array}{rcll} exp & \longrightarrow & exp_1\ op_{\text{infix}}\ exp_2 & \text{(infix operator application)} \\ & | & exp\ op_{\text{suffix}} & \text{(suffix operator application)} \end{array}$$

The form $e_1\,op\,e_2$ is the infix application of a binary operator op to the expressions e_1 and e_2. The form $e_1\,op$ is the suffix application of a unary operator op to the expression e_1 (Fig. 8.7).

Translation:	If op is an infix operator, $e_1\ op\ e_2$ is equivalent to $(op)\ e_1\ e_2$. If op is a suffix operator, $e_1\ op$ is equivalent to $(op)\ e_1$. Otherwise $e_1\ op\ e_2$ is equivalent to $((e_1\ (op))\ e_2)$.

Figure 8.7: Translation rule for operators

8.4.4 Sections

$$\begin{array}{rcl} aexp & \longrightarrow & (exp\ op) \\ & | & (op\ exp) \end{array}$$

Sections are written as (*op e*) or (*e op*), where *op* is a binary operator and *e* is an expression. Sections are convenient syntax for partial application of binary operators (Fig. 8.8).

The normal rules of syntactic precedence apply to sections; for example `(*a+b)` is invalid, but `(+a*b)` and `(*(a+b))` are valid.

Translation:	For binary operator op and expression e, if x is a variable that does not occur free in e, the section $(op\ e)$ is equivalent to \$x$->$ $x\ op\ e$, and $(e\ op)$ is equivalent to \$x$->$ $e\ op\ x$.

Figure 8.8: Translation rule for sections

8.4.5 Conditionals

$$exp \longrightarrow \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3$$

A *conditional expression* has the form `if` e_1 `then` e_2 `else` e_3 and returns the value of e_2 if the value of e_1 is `True`, and e_3 if e_1 is `False`, where `True` and `False` are the two built-in nullary constructors for booleans. Conditional expressions are left unchanged in the kernel although an equivalent translation is given in Fig. 8.9.

Translation:	if e_1 then e_2 else e_3 is equivalent to case e_1 of {True -> e_2;False -> e_3} where True and False are the two built-in nullary constructors from the booleans.

Figure 8.9: Translation rule for conditionals

8.4.6 Arrays

$$\textit{iteration} \quad \rightarrow \quad [e_1, \ldots, e_n] \qquad (k \geq 0)$$

Arrays are of the form $[e_1, \cdots, e_n]$, where $n \geq 0$; the empty array is written []. *Arrays* are the predominant datatype in FSC and hence much work has been done to facilitate their use. Since FSC is based heavily around pattern matching there is a facility to pattern match against array arguments, with many analogies being drawn with *lists* in languages such as Haskell. Standard operations for constructing arrays are shown in Fig. 8.10. Other array operations exist and these

Operation	Translation
A +> a	append a to the end of A
a <+ A	append a to the front of A
A ++ B	concatenate A and B
l <: A	set the lower bound of A to l
A :> u	set the upper bound of A to u
[]	The empty array

Figure 8.10: Intrinsic operations over arrays

will be discussed later in this chapter. Now that we have these constructors we may give a meaning to $[e_1, \ldots, e_n]$ (Fig. 8.11).

Translation:	$[e_1, \ldots, e_n]$ is equivalent to e_1 <+ (e_2 <+ ($\cdots$ (e_n <+ [])))). The types of e_1 to e_n must all be τ for some type τ and the type of the overall expression is Array(τ)

Figure 8.11: Translation rules for array patterns

8.4.7 Tuples

$$\textit{aexp} \quad \rightarrow \quad (e_1, \ldots, e_n) \qquad (n \geq 0)$$

Tuples are of the form $(e_1, \ldots, e_n)$ and may be of arbitrary length $n \geq 2$ (Fig. 8.12).

8.4.8 Unit Expressions and Parenthesised Expressions

$$\begin{array}{lll} \textit{aexp} & \rightarrow & (e) \\ & | & () \end{array}$$

Translation: $(e_1, \ldots, e_n)$ for $n \geq 2$ is an instance of an n-tuple and requires no translation.

Figure 8.12: Translation rule for tuples

The form (e) is simply a *parenthesised expression* and is equivalent to e (Fig. 8.13). The *unit expression* () has type () (the void type).

Translation: (e) is equivalent to e.

Figure 8.13: Translation rule for parentheses

8.4.9 Arithmetic Sequences and Strides

iteration	→	$[e_1 .. e_2]$	(range)
	|	$[e_1 : e_2 \{: e_3\}]$	(stride)

The form $[e_1 .. e_2]$ denotes the *arithmetic sequence* from e_1 to e_2 either up or down in steps of 1. This generates an array of integers. Index arrays with non unit strides are generated in a similar manner. The form $[e_1 : e_2 : e_3]$ denotes the *arithmetic sequence* from e_1 in increments of e_3 of values not greater than e_2. If e_3 is omitted then an increment of 1 is assumed.

8.4.10 Array Comprehensions

iteration	→	$[retexp_1, \ldots, retexp_n \mid range\ [\mid inits]\ ]$	$n \geq 1$
	|	$[exp_1; \ldots; exp_n \mid range\ [\mid inits]\ ]$	$n \geq 1$
retexp	→	*[aexp* `of` *] exp[(`when`|`unless`) exp]*	
inits	→	$init_1; \ldots; init_n$	$n \geq 1$
init	→	*pat* = *exp*	
range	→	*updates* (`while`|`until`) *exp*	
	|	(`while`|`until`)*aexp updates*	
	|	*{updates;} inrange*	
	|	`all` *var*	
updates	→	$update_1; \ldots; update_n$	$n \geq 1$
update	|	*pat* <- *exp*	
inrange	→	*pat* `in` *exp* [`at` *var*] [(`dot`|`cross`|;) *inrange*]	

Array comprehensions are similar to list comprehensions in Haskell. They must not be recursive. Before providing translation rules for array comprehensions we simplify these comprehensions via the elimination of ;, `at`, `dot` and `cross` constructs[3].

[3]These examples aim to show semantic equivalence rather than efficient implementation.

8.4.10.1 Semicolon-Comprehensions

The form

$$[exp_1; \ldots; exp_n \,|\, range\ [|\, inits]\ ] \qquad n \geq 1$$

is equivalent to

$$concat\ [[exp_1, \ldots, exp_n]\ |\, range\ [|\, inits]\ ] \qquad n \geq 1$$

8.4.10.2 at constructs

Array comprehensions involving the `at` construct may be simplified as shown below:

$$[\ retexps\ |\ \ldots\ a\ \texttt{in}\ A\ \texttt{at}\ i \ldots\ ]$$

is equivalent to

$$[\ retexps\ |\ \ldots\ i\ \texttt{in}\ [(\_liml(A))..(\_limh(A))]..]\llbracket a/A[i]\rrbracket$$

where we use $e\llbracket a/A\rrbracket$ to denote the replacement of all free occurrences of a in e with A.

8.4.10.3 dot Loop-Fusion

Array comprehensions involving the `dot` construct may be simplified as shown below:

$$[\ retexps\ |\ \ldots\ a\ \texttt{in}\ A\ \texttt{dot}\ b\ \texttt{in}\ B\ \ldots\ ]$$

is equivalent to

```
let
    hi = max(_size(A),_size(B)) - 1
in
    [ retexps | ... i in [0..hi]..]⟦a/A[i + _liml(A)]⟧⟦b/B[i + _liml(B) ]⟧
```

e.g. Using the `dot` construct a dot-product may be written as

```
dot X Y = sum[x * y | x in X dot y in Y]
```

8.4.10.4 cross Loop-Fusion

Array comprehensions involving the `cross` construct may be simplified as shown below:

$$[\ exp|\ \ldots\ a\ \texttt{in}\ A\ \texttt{cross}\ b\ \texttt{in}\ B\ \ldots\ ]$$

is equivalent to

$$[\ [\ exp\ |\ \ldots\ b\ \texttt{in}\ B\ \ldots\ ]\ |\ a\ \texttt{in}\ A\ ]$$

If the expression contains a guard or a function `of` application then the application is transported to the outer comprehension but the guard is not.

$$[\ e_1\ \texttt{of}\ e_2\ \texttt{when}\ e_3\ |\ \ldots\ a\ \texttt{in}\ A\ \texttt{cross}\ b\ \texttt{in}\ B\ \ldots\ ]$$

is equivalent to

$$\texttt{[} e_1 \texttt{ of [} e_1 \texttt{ of } e_2 \texttt{ when } e_3 \texttt{ | } \ldots\ b \texttt{ in } B \ \ldots \texttt{] | } a \texttt{ in } A \texttt{]}$$

An identity matrix could be generated using the `cross` construct as follows

```
idM N = [if i==j then 1 else 0 | i in [1..N] cross j in [1..N]]
```

and all the elements in a matrix could be incremented by 1 via

```
inc A = [1+A[i,j] | i in [1..M] cross j in [1..N]]
```

A semicolon between `in`-ranges defines the cross product of the two ranges *à la* Miranda, i.e.

$$\texttt{[} exp \texttt{| } \ldots\ a \texttt{ in } A \texttt{ ; } b \texttt{ in } B \ \ldots \texttt{]}$$

is equivalent to

$$concat \texttt{ ([[} exp \texttt{ | } \ldots\ b \texttt{ in } B \ \ldots \texttt{] | } a \texttt{ in } A \texttt{])}$$

If the expression contains a guard or a function `of` application then the application is transported outside the comprehension but the guard is not.

$$\texttt{[} e_1 \texttt{ of } e_2 \texttt{ when } e_3 \texttt{ | } \ldots\ a \texttt{ in } A \texttt{ ; } b \texttt{ in } B \ \ldots \texttt{]}$$

is equivalent to

$$e_1(concat \texttt{ ([[} e_2 \texttt{ when } e_3 \texttt{ | } \ldots\ b \texttt{ in } B \ \ldots \texttt{] | } a \texttt{ in } A \texttt{]}))$$

8.4.10.5 (for) `all` Comprehensions

Translation of the keyword `all` in

```
>       dotProd A B = sum[A[i]*B[i] | all i]
```

proceeds as follows

- All (*sic*) the `all` indices are normalised. That is, each array access containing all-variable i is transformed to the form $ai + c$, where a and c are integer constants. If this is not possible a static error is returned.
- The lower and upper bounds for each access is calculated as in Section 7.5 and these values are used in an `in` comprehension.

8.4.10.6 in, while and until Comprehensions

The array comprehensions of FSC are very similar to the list comprehensions of Haskell. If we disregard the fact that Haskell's list comprehensions are lazy then the Haskell expression

$$\texttt{[} e_1 \texttt{ | } i_1 \texttt{ <- } I_1, Q_1(i_1), i_2 \texttt{ <- } I_2 \texttt{]}$$

is equivalent to the FSC array comprehension

$$\texttt{[} e_1 \texttt{ when } Q_1(i_1) \texttt{ | } i_1 \texttt{ in } I_1; i_2 \texttt{ in } I_2 \texttt{]}$$

However, FSC extends this notation to allow the definition of multiple values and allow iteration. The manner in which it allows multiple values is that it allows ranges to be shared between computations, that is,

```
(sum[i | i in [1..10]],product[i | i in [1..10]])
```

may be written as

```
[sum of i,product of i | i in [1..10]]
```

and the array traversal may be shared between computations. FSC also allows iterations to be performed within array comprehensions *à la* SISAL. (This syntax is similar to that proposed in Chapter 5.) As an example of an array comprehension containing iteration, consider the conjugate gradient iteration `cgIteration` function from Section 5.2.4. This may be used to write an iteration in FSC:

```
>       conj_grad x0 eps A b
>        = last[x | (x,p,r) <- cgIteration (x,p,r)
>              until (norm r<eps)|(x,p,r) = (x0,b-A*x0,p0)]
```

The remainder of this section details how these expressions may be represented in the FSC kernel.

8.4.10.7 Translations

The following translations (Figs. 8.14-8.19) apply and make use of the following *repeat* functions:

```
> repeat :: (s->a)->(s->Bool)->(s->s)->s->[a]
> repeat prj p upd init
>  = [prj init] ++ if (p init) then repeat prj p upd (upd init) else []

> repeat2 :: (s->a)->(s->Bool)->(s->s)->s->[a]
> repeat2 prj p upd init
>  = if (p init) then [prj init] ++ repeat prj p upd (upd init) else []

> repeat3 :: (s->a)->(s->i->s)->s->[i]->[a]
> repeat3 prj upd init [] = []
> repeat3 prj upd init (a <+ A) = prj uia <+ repeat3 prj upd uia A
>                                 where uia = upd init a
```

Translation: The array comprehension

$[rv_1, \ldots, rv_n \mid \texttt{while}\ (cond)\, x_1 \leftarrow e_1; \ldots; x_m \leftarrow e_m \mid x_1 = i_1; \ldots; x_m = i_m\ ]$

is equivalent to

let $upd\,(x_1, \ldots, x_m) = (e_1, \ldots, e_m)$

$cnd\,(x_1, \ldots, x_m) = cond$

$init = (i_1, \ldots, i_m)$

$prj_1(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$\vdots$

$prj_n(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$pack_1 = \mathsf{TE}_{RV_2}[\![rv_1]\!]$

$\vdots$

$pack_n = \mathsf{TE}_{RV_2}[\![rv_n]\!]$

in

$(pack_1(repeat\ prj_1\ cnd\ upd\ init), \ldots, pack_n(repeat\ prj_n\ cnd\ upd\ init))$

Figure 8.14: Translation rule for prefix-while array comprehension

Translation: The array comprehension

$[rv_1, \ldots, rv_n \mid \texttt{until}\ (cond)\, x_1 \leftarrow e_1; \ldots; x_m \leftarrow e_m \mid x_1 = i_1; \ldots; x_m = i_m\ ]$

is equivalent to

let $upd\,(x_1, \ldots, x_m) = (e_1, \ldots, e_m)$

$cnd\,(x_1, \ldots, x_m) = not\ (cond)$

$init = (i_1, \ldots, i_m)$

$prj_1(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$\vdots$

$prj_n(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$pack_1 = \mathsf{TE}_{RV_2}[\![rv_1]\!]$

$\vdots$

$pack_n = \mathsf{TE}_{RV_2}[\![rv_n]\!]$

in

$(pack_1(repeat\ prj_1\ cnd\ upd\ init), \ldots, pack_n(repeat\ prj_n\ cnd\ upd\ init))$

Figure 8.15: Translation rule for prefix-until array comprehension

Translation: The array comprehension

$[rv_1, \ldots, rv_n \mid x_1 \leftarrow e_1; \ldots; x_m \leftarrow e_m$ **while**$(cond) \mid x_1 = i_1; \ldots; x_m = i_m\ ]$

is equivalent to

let $upd\ (x_1, \ldots, x_m) = (e_1, \ldots, e_m)$

$cnd\ (x_1, \ldots, x_m) = cond$

$init = (i_1, \ldots, i_m)$

$prj_1(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$\vdots$

$prj_n(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$pack_1 = \mathsf{TE}_{RV_2}[\![rv_1]\!]$

$\vdots$

$pack_n = \mathsf{TE}_{RV_2}[\![rv_n]\!]$

in

$(pack_1(repeat2\ prj_1\ cnd\ upd\ init), \ldots, pack_n(repeat2\ prj_n\ cnd\ upd\ init))$

Figure 8.16: Translation rule for postfix-while array comprehension

Translation: The array comprehension

$[rv_1, \ldots, rv_n \mid x_1 \leftarrow e_1; \ldots; x_m \leftarrow e_m$ **until**$(cond) \mid x_1 = i_1; \ldots; x_m = i_m\ ]$

is equivalent to

let $upd\ (x_1, \ldots, x_m) = (e_1, \ldots, e_m)$

$cnd\ (x_1, \ldots, x_m) = not(cond)$

$init = (i_1, \ldots, i_m)$

$prj_1(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$\vdots$

$prj_n(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$pack_1 = \mathsf{TE}_{RV_2}[\![rv_1]\!]$

$\vdots$

$pack_n = \mathsf{TE}_{RV_2}[\![rv_n]\!]$

in

$(pack_1(repeat2\ prj_1\ cnd\ upd\ init), \ldots, pack_n(repeat2\ prj_n\ cnd\ upd\ init))$

Figure 8.17: Translation rule for postfix-until array comprehension

Translation: The array comprehension

$[rv_1, \ldots, rv_n \mid x_1 \leftarrow e_1; \ldots; x_m \leftarrow e_m \ ; ind \ \texttt{in} \ I \mid x_1 = i_1; \ldots; x_m = i_m \ ]$

is equivalent to

$\texttt{let} \ \ upd\ (x_1, \ldots, x_m) ind = (e_1, \ldots, e_m)$

$init = (i_1, \ldots, i_m)$

$prj_1(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$\vdots$

$prj_n(x_1, \ldots, x_m) = \mathsf{TE}_{RV_1}[\![rv_1]\!]$

$pack_1 = \mathsf{TE}_{RV_2}[\![rv_1]\!]$

$\vdots$

$pack_n = \mathsf{TE}_{RV_2}[\![rv_n]\!]$

in

$(pack_1(repeat3\ prj_1\ upd\ init\ I), \ldots, pack_n(repeat3\ prj_n\ upd\ init\ I))$

Figure 8.18: Translation rule for element array comprehension

Translation: TE_{RV_1} and TE_{RV_1} are defined as

$\mathsf{TE}_{RV_1}[\![$ *return* `of` *e* `unless` *p* $]\!]$	$\Rightarrow$ `if` *p* `then [] else [`*e*`]`
$\mathsf{TE}_{RV_1}[\![$ *return* `of` *e* `when` *p* $]\!]$	$\Rightarrow$ `if` *p* `then [`*e*`] else []`
$\mathsf{TE}_{RV_1}[\![$ *return* `of` *e* $]\!]$	$\Rightarrow$ `[`*e*`]`
$\mathsf{TE}_{RV_1}[\![$ *e* `unless` *p* $]\!]$	$\Rightarrow$ `if` *p* `then [] else [`*e*`]`
$\mathsf{TE}_{RV_1}[\![$ *e* `when` *p* $]\!]$	$\Rightarrow$ `if` *p* `then [`*e*`] else []`
$\mathsf{TE}_{RV_1}[\![$ *e* $]\!]$	$\Rightarrow$ `[`*e*`]`
$\mathsf{TE}_{RV_2}[\![$ *return* `of` *e* `unless` *p* $]\!]$	$\Rightarrow$ *return*
$\mathsf{TE}_{RV_2}[\![$ *return* `of` *e* `when` *p* $]\!]$	$\Rightarrow$ *return*
$\mathsf{TE}_{RV_2}[\![$ *return* `of` *e* $]\!]$	$\Rightarrow$ *return*
$\mathsf{TE}_{RV_2}[\![$ *e* `unless` *p* $]\!]$	$\Rightarrow$ `id`
$\mathsf{TE}_{RV_2}[\![$ *e* `when` *p* $]\!]$	$\Rightarrow$ `id`
$\mathsf{TE}_{RV_2}[\![$ *e* $]\!]$	$\Rightarrow$ `id`

Figure 8.19: Translation rule for array comprehension guards

Translation:	TE_{RV_3} is defined as	
	$TE_{RV_3}[\![$ *return* `of` *e* `unless` *p* $]\!]$	$\Rightarrow$ *return*
	$TE_{RV_3}[\![$ *return* `of` *e* `when` *p* $]\!]$	$\Rightarrow$ *return*
	$TE_{RV_3}[\![$ *return* `of` *e* $]\!]$	$\Rightarrow$ *return*
	$TE_{RV_3}[\![$ *e* `unless` *p* $]\!]$	$\Rightarrow$ `last`
	$TE_{RV_3}[\![$ *e* `when` *p* $]\!]$	$\Rightarrow$ `last`
	$TE_{RV_3}[\![$ *e* $]\!]$	$\Rightarrow$ `last`

Figure 8.20: Translation rule for value comprehension guards

8.4.11 Value Comprehensions

iteration	$\rightarrow$	($retval_1, \ldots, retval_n$ | *range* [| *inits*])	$n \geq 1$
retval	$\rightarrow$	*exp*[(`when`|`unless`) *exp*]	
inits	$\rightarrow$	$init_1; \ldots; init_n$	$n \geq 1$
init	$\rightarrow$	*pat* = *exp*	
range	$\rightarrow$	*updates* (`while`|`until`) *exp*	
	|	(`while`|`until`)*aexp updates*	
	|	{*updates*;} *inrange*	
	|	`all` *var*	
updates	$\rightarrow$	$update_1; \ldots; update_n$	$n \geq 1$
update	|	*pat* `<-` *exp*	
inrange	$\rightarrow$	*pat* `in` *exp* [`at` *var*] [(`dot`|`cross`|;) *inrange*]	

Value comprehensions are similar to array comprehensions. Again, they may not be recursive.

$$(e_1(\texttt{when}|\texttt{unless})e_{1c}, \ldots, e_n(\texttt{when}|\texttt{unless})e_{1c} \mid range\ [\mid inits]\ )$$

is equivalent to

$$[\texttt{last of}\, e_1(\texttt{when}|\texttt{unless})e_{1c}, \ldots, \texttt{last of}\, e_n(\texttt{when}|\texttt{unless})e_{1c} \mid range\ [\mid inits]\ ]$$

The translation rules for value comprehensions are identical to those of array comprehensions with the following exception: translation TE_{RV_2} is everywhere replaced by TE_{RV_3} (Fig. 8.20). where `last (A +> a) = a` and `last [] =` $\perp$.

8.4.12 previous Value Definitions

In many applications, values from different iterations are used. While it is possible to express this fact in the above framework it involves the manual copying of data between iterations which is syntactically ugly. Hence we introduce the use of the `prev` modifier which allows access to the value bound to an identifier in the previous iteration.

The modifier `prev` is syntactically treated like a function `prev :: a -> a` but may only appear within (array/value) comprehensions, and moreover, only those involving `while` or `until` terminators.

Any number of **prev** modifiers may be used; e.g. **prev (prev (prev x))** is legal and refers to the value that was bound to x three iterations previously.

8.4.12.1 Example Usage: Newton Iteration

Using a single **prev** modifier we may write a Newton iteration as:

```
> sqrt :: DOUBLE -> DOUBLE -> DOUBLE -> DOUBLE
> sqrt init TOL a = ( x  | while (abs(x - prev x) > TOL)
>                                  x <- newton a x
>                                 |x =  init)
> newton :: DOUBLE -> DOUBLE -> DOUBLE
> newton a X = ( X + a / X) /  2
```

8.4.12.2 Translation

prev-modifiers are translated out by adding extra identifiers to hold the previous values and the first n iterations performed outside the loop via unrolling (where n is the length of the longest chain of **prev** applications). The **prev** values are then simply updated at each iteration. That is,

```
> sqrt :: DOUBLE -> DOUBLE -> DOUBLE -> DOUBLE
> sqrt init TOL a = ( x  | while (abs(x - prev x) > TOL)
>                                  x <- newton a x
>                                 |x =  init)
```

is translated to

```
> sqrt :: DOUBLE -> DOUBLE -> DOUBLE -> DOUBLE
> sqrt init TOL a
>  = let
>   _newton_a_init = newton a init
>    in
>                 ( x  | while (abs(x - _prev_x) > TOL)
>                                  x <- newton a x;
>                                  _prev_x <- x
>                               |
>                                  _prev_x =          init;
>                                  x       = _newton_a_init)
```

Note: care must be taken to ensure that unrolling of the first n iterations does not cause re-computation. That is if **prev** x and **prev (prev** x**))** co-exist then the initialisation should not be written as

```
>       let
>          _f_init   = f init
>          _f_f_init = f (f init)
>        in
>        ... | _prev_prev_x =      init;
>              _prev_x      =   _f_init;
>              x            = _f_f_init
>       ]
```

but instead as

```
>       let
>           _f_init   = f init
>           _f_f_init = f _f_init
>       in
>       ... | _prev_prev_x =      init;
>             _prev_x      =   _f_init;
>             x            = _f_f_init
>       ]
```

8.4.13 Let Expressions

Let expressions have the general form `let` $\{ d_1 ; \ldots ; d_n \}$ `in` e, and introduce a nested, lexically-scoped, *mutually-recursive* list of declarations (`let` is often called `letrec` in other languages, and in the translation to the FSC core we also denote recursive declarations via this keyword). The scope of the declarations is the expression e and the right-hand side of the declarations (Fig. 8.21). Declarations are described in Section 8.5. Pattern bindings are only allowed on irrefutable patterns.

Translation: The semantics of the expression `let` $\{ d_1 ; \ldots ; d_n \}$ `in` e_0 is captured as follows:
After removing all type signatures, each declaration d_i is translated into an equation of the form $p_i = e_i$, where p_i and e_i are patterns and expressions respectively,
Once done, these equalities hold, which may be used as a translation into the kernel:

`let` $\{p_1 = e_1; p_2 = e_2; \ldots; p_n = e_n\}$ `in` e_0
$=$ `let` $\{p_1 = e_1\}$ `in` (`let` $\{p_2 = e_2; \ldots; p_n = e_n\}$ `in` e_0)
where none of $p_1 \ldots p_n$ appear in e_1

`let` $\{p_1 = e_1; p_2 = e_2; \ldots; p_n = e_n\}$ `in` e_0
$=$ `let` $\{p_2 = e_2; \ldots; p_n = e_n\}$ `in` (`let` $\{p_1 = e_1\}$ `in` e_0)
where none of $e_1 \ldots p_n$ reference in p_1

`let` $\{p_1 = e_1; p_2 = e_2; \ldots; p_n = e_n\}$ `in` e_0
$=$ `let` $\{p_2 = e_2; \ldots; p_n = e_n; p_1 = e_1\}$ `in` e_0

`let` $\{p_1 = e_1; p_2 = e_2; \ldots; p_n = e_n\}$ `in` e_0
$=$ `letrec` $\{p_1 = e_1; p_2 = e_2; \ldots; p_n = e_n\}$ `in` e_0
where the p_i are *mutually-recursive*.

Figure 8.21: Translation rule for let expressions

8.4.14 For Expressions

iteration	→	**for** [**initial** {*inits*}]	$range_{for}$ **returns** (*retexp*$_1$, ..., *retexp*$_n$)	$n \geq 1$
retexp	→	[*aexp* **of**] *exp*[(**when**|**unless**) *exp*]	
inits	→	*init*$_1$; ...; *init*$_n$		$n \geq 1$
init	→	*pat* = *exp*	
$range_{for}$	→	**repeat** {*updates* }(**while**|**until**) *exp*	
	|	(**while**|**until**)*aexp* {**repeat** *updates*}	
	|	{*updates*;} *inrange*	
	|	**all** *var*	
updates	→	*update*$_1$; ...; *update*$_n$		$n \geq 1$
update	|	*pat* <- *exp*	
inrange	→	*pat* **in** *exp* [**at** *var*] [(**dot**|**cross**|;) *inrange*]	

If a lot of information is to be conveyed in an array comprehension then the syntax can get very cluttered. For this reason FSC also provides *for expressions*.

For expressions are semantically equivalent to array comprehensions:

for [**initial** {*inits*}]$range_{for}$ **returns** (*retexp*$_1$, ..., *retexp*$_n$)

≡

[*retexp*$_1$, ..., *retexp*$_n$ | *range* [| *inits*]]

8.4.15 Case Expressions

exp	→	**case** *exp* **of** { *alts*[;]}	
alts	→	*alt*$_1$; ...; *alts*$_n$	$(n \geq 1)$
alt	→	*pat* -> *exp* [**where** {*decls* [;]}]	
	|	*pat gdexp* [**where** { *decls* [;]}]	
gdpat	→	*gd* -> *exp* [*gdpat*]	
gd	→	| *exp*	

A *case expression* has the general form

case *e* **of** {$p_1 match_1$; ...; $p_n match_n$}

where each $match_i$ is of the general form

$|\, g_{i1}$->e_{i1};
...
$|\, g_{im}$->e_{im};
where {$decls_i$}

A case expression must have at least one alternative and each alternative must have at least one body. Each body must have the same type, and the type of the whole expression is that type.

A case expression is evaluated by pattern matching the expression e against the individual alternatives. The matches are tried sequentially, from top to bottom. The first successful match causes evaluation of the corresponding alternative body, in the environment of the case expression extended by the bindings created during the matching of that alternative, and by the *decls* associated with that alternative. If no match succeeds, the result is $\perp$. Pattern matching is described in Subsection 8.4.18

8.4.16 Do Expressions

exp	→	`do` *domain*`{`*actions*`[;]}`	
actions	→	$action_1 ; \ldots ; action_n$	$n \geq 1$
action	→	`return` *exp*	(unit)
	|	*pat* `<-` *exp*	(bind)
	|	*exp*	
domain	→	*type*	(domain choice)

In FSC all side-affecting computations, such as I/O, are carried out via the do expression. This expression serves a similar purpose to the do-expression in the GoFER language in that it is a syntactic sugaring for a monad computation. However, in FSC we do not allow user-defined extension of the do syntax via the use of a monad constructor class, but instead make these an implementation dependent feature with the type variable immediately after the do keyword selecting which monad is to be chosen. Although the choice of monads is implementation dependent we insist that the I/O monad be available with intrinsic operations prototyped in Appendix H. This explicit monad naming will aid compilation and simplify error reporting.

8.4.17 Expression Type-Signatures

$$exp \rightarrow aexp :: type$$

Expression type-signatures have the form $e::t$, where e is an expression and t is a type. They are used to type an expression explicitly. The value of the expression is just that of *exp*. The only rule FSC specifies regarding the value of t is that t must be unifiable with the principal type deliverable from *exp*. It is an error to give a type not comparable to the principal type.

8.4.18 Pattern Matching

Patterns appear in lambda abstractions, function definitions and case expressions. However, the first two of these ultimately translate into case expressions, and so defining the semantics of pattern-matching for case expressions is sufficient.

8.4.18.1 Patterns

Patterns have the syntax:

pat	→	*apat*	
	|	*con fpats*	
apat	→	*var*[@*apat*]	(as pattern)
	|	*con*	(arity *con=0*)
	|	*literal*	
	|	_	(wildcard)
	|	()	(unit pattern)
	|	(*pat*)	(parenthesised pattern)
	|	$(pat_1, \ldots, pat_k)$	(tuple pattern, $k \geq 2$)
	|	$[pat_1, \ldots, pat_k]$	(array pattern, $k \geq 0$)
	|	$[pat_1, \ldots, pat_k)$	(left edge array pattern, $k \geq 0$)
	|	$(pat_1, \ldots, pat_k]$	(right edge array pattern, $k \geq 0$)
fpats	→	$\{pat_1, \ldots, pat_k\}$[*fpats*]	(application pattern, $k \geq 1$)
	|	*apat*[*fpats*]	

The arity of a constructor must match the number of sub-patterns associated with it; one cannot

Translation: $[e_1, \ldots, e_n]$ is equivalent to
(_ +> e_1 +> e_2 +> $\cdots$ e_n)

Translation: $[e_1, \ldots, e_n)$ is equivalent to
(e_1 <+ e_2 <+ $\cdots$ e_n <+ _)

Figure 8.22: Translation rule for arrays

match against a partially applied constructor.

All patterns must be *linear*, no variable may appear more than once.

Patterns of the form *var@pat* are called *as-patterns*, and allow one to use *var* as a name for the value being matched by *pat*.

Patterns of the form _ are *wildcards* and are useful when some part of the pattern is not referenced on the right-hand side. Right and Left edge array patterns are given a translation in Fig. 8.22.

In this chapter we distinguish two kinds of patterns. An *irrefutable* pattern is a pattern which cannot fail, such as a variable or tuple. All other patterns are *refutable*.

8.4.18.2 Informal Semantics of Pattern Matching

Patterns are matched against values. Attempting to match a pattern may result in one of three results: it may *fail*; it may *succeed*, returning a binding for each variable in the pattern; or it may *diverge* (i.e. return $\perp$). Pattern matching proceeds left-to-right, and outside in, according to the following rules:

1. Matching a value v against the irrefutable pattern *var* always succeeds and binds *var* to v. Matching a value v against the pattern *var@apat* always fails if matching *apat* to v fails; and

diverges if matching *apat* to v diverges. Otherwise *var* and the free variables in *apat* are bound to the appropriate values from v.

2. Matching $\perp$ against any pattern (refutable or otherwise) always diverges and, since any value containing $\perp$ is identical to $\perp$, any value containing $\perp$ fails similarly.

3. Matching a non-$\perp$ value succeeds only if the outermost constructor matches and all the sub-patterns in the pattern in question also match.

8.4.18.3 Formal Semantics of Pattern Matching

The semantics of all pattern matching constructs other than `case` expressions is defined by giving identities that relate those constructs to `case` expressions. The semantics of `case` expressions themselves are in turn given as a series of identities, as shown in Fig. 8.23. Any implementation should behave so that these identities hold; it is not expected that it will use them directly, since this would generate rather inefficient code.

In Fig. 8.23 e, e' and e_i are expressions; g and g_i are Boolean-valued expressions; p and p_i are patterns; x and x_i are variables; K and K' are constructors (including tuple constructors); and k is a character, string or numeric literal.

```
case e0 of { p1 match1;...;pn matchn }
    =case e0 of { p1 match1;_ -> ... case e0 of { pn matchn;_ -> error "no match" }...}
case e0 of { _ -> e;...;_ -> e' } = e
case e0 of { k -> e;_ -> e' } = if (e0 == k) then e else e'
case e0 of { x -> e;_ -> e' } = (\ x -> e) e0
case ( K' e1...en ) of { (K x1...xn) -> e ;_ -> e' } = e' , K ≠ K'
case ( K' e1...en ) of { (K x1...xn) -> e ;_ -> e' } = e[x1/e1,...,xn/en] , K = K'
case ( K' e1...en ) of { (K p1...pn) -> e ;_ -> e' }
    = case ( K' e1...en ) of { (K x1...xn) -> case x1 of {
        p1 -> ... case xn of { pn -> e; _ -> e' } ... _ -> e' } _ -> e' }
```

Figure 8.23: Case expression identities

8.5 Declarations and Bindings

In this section we describe the syntax and informal semantics of FSC *declarations*.

module	→	**module** *modid* [*exports*]**where** *body*	
	|	*body*	
body	→	{[*impdecls*;][[*fixdecls* ;]*topdecls* [;]]}	
	|	{*impdecls* [;]}	
topdecls	→	$topdecl_1; \ldots; topdecl_n$	$(n \geq 1)$
topdecl	→	**type** *simple* = *type*	
	|	**datatype** *simple* = *constrs*	
	|	**class**[*context* =>]*class* [**where** {*cbody* [;]}]	
	|	**instance**[*context* =>]*tycls inst* [**where** {*valdefs* [;]}]	
	|	*transformation*	
	|	*decl*	
decls	→	$decl_1; \ldots; decl_n$	$(n \geq 0)$
decl	→	*arraydecl*	
	|	*valdef*	

Like Haskell, the declarations in the syntactic category *topdecls* are only allowed at the top level of an FSC module, whereas *decls* may be used either at the top level or in nested scope (i.e. those within a `let` or `where` construct).

We divide declarations into three groups: user-defined datatypes, consisting of **type** and **datatype** declarations; type-classes and overloading consisting of `class` and **instance** declarations; and nested declarations consisting of value bindings and type signatures.

8.5.1 Overview of Types and Classes

Like Haskell, FSC uses a traditional Hindley-Milner polymorphic type system to provide a static type semantics, but the type system has been extended with *type classes* that provide a structured way to introduce *overloaded* functions.

A `class` definition introduces a new *type class* and the overloaded *operations* that must be supported by any type that is an instance of that class. An **instance** declaration declares that a type is an *instance* of a class and includes the definitions of the overloaded operations (called *methods*) instantiated on the named type.

For example, suppose we wish to overload the operation `(+)` on types **INT** and **REAL**. We introduce a class called `PLUS`:

```
> class PLUS (a,b,c) where
>   (+) :: a -> b -> c
```

This declaration may be read "a triple `(a,b,c)` is an instance of the class `PLUS` if there is an overloaded operation `(+)` of the appropriate types, defined on it."

We may then declare instances of this class involving **REAL** and **INT**:

```
> instance PLUS (INT,INT,INT) where
>  x + y = addIntInt x y

> instance PLUS (REAL,REAL,REAL) where
>  x + y = addRealReal x y

> instance PLUS (REAL,INT,REAL) where
>  x + y = addRealInt x y

> instance PLUS (INT,REAL,REAL) where
>  x + y = addIntReal x y
```

where `addIntInt`, `addRealReal`, `addIntReal` and `addRealInt` are assumed to be primitive functions, but, in general, could be any user-defined function. The first declaration above may be read "`(INT,INT,INT)` is an instance of class `PLUS` as witnessed by the definition for `(+)`".

8.5.1.1 Syntax of Types

type	→	*btype*[^ *digit*][-> *type*]	
btype	—	$atype_1 \ldots atype_k$	(arity *tycon* $= k$, $k \geq 1$)
	|	*atype*	
atype	→	*tyvar*	
	|	*tycon*	(arity *tycon* = *0*)
	|	()	(unit type)
	|	(*type*)	(parenthesised type)
	|	$(type_1, \ldots, type_k)$	(tuple type $k \geq 2$)
	|	[*type*]	(array type)

The syntax for FSC *type expressions* is given above. They are built in the usual way from type variables, function types, type constructors, tuple types, and array types. Type variables are identifiers beginning with a lower-case letter and type constructors are identifiers with an upper-case letter.

The array type [*t*] (also written *Array t*) is an array with element type *t*. The caret may be used in types as a short hand for multiple function construction.

```
>          [a]^0 = ()
>          [a]^1 = [a]
>          [a]^2 = [a] -> [a]
>          [a]^3 = [a] -> [a] -> [a]
>          ...
```

All type variables are taken to be implicitly universally quantified except those in class declarations which are explicitly existentially qualified.

8.5.1.2 Syntax of Class Assertions and Contexts

context	→	*class*	
	|	$(class_1, \ldots, class_n)$	$(n \geq 1)$
class	→	$tycls(type_1, \ldots, type_n)$	$(n \geq 1)$
tycls	→	*conid*	

A *class assertion* has the form $tycls(type_1, \ldots, type_n)$ and indicates membership of the tuple

$$(type_1, \ldots, type_n)$$

in the class *tycls*. A class identifier begins with a capital letter.

A *context* consists of one or more class assertions, and has the general form

$$(C_1 u_1, \ldots, C_n u_n)$$

where $C_1, \ldots, C_n$ are class identifiers and the $u_1, \ldots, u_n$ are n-tuples ($n \geq 1$) of types. As in Haskell, we use c to denote a context and write c=> t to indicate the type t restricted by the context c. Note that, unlike Haskell, FSC does not enforce explicit contexts in user provided type declarations.

Further details of FSC type classes were given in Chapter 7.

8.5.2 User-Defined Datatypes

In this section we describe algebraic datatypes (`datatype` declarations) and type synonyms (`type` declarations). These declarations may only appear at the top level of a module.

8.5.2.1 Algebraic Datatype Declarations

topdecl	→	**datatype** *simple* = *constrs*	
simple	→	$tycon\ tyvar_1 \ldots tyvar_k$	(arity $tycon = k$, $k \geq 0$)
constrs	→	$constr_1 \mid \ldots \mid constr_n$	$(n \geq 1)$
constr	→	$con\ atype_1 \ldots atype_k$	(arity $con = k$, $k \geq 0$)

Infix/suffix constructor functions are handled in exactly the same way as infix/suffix identifiers.

An algebraic datatype declaration introduces a new type and constructors over that type and has the form

$$\texttt{datatype}\ T\ u_1 \ldots u_k = K_1\ t_{11} \ldots t_{1k} \mid \ldots \mid K_n\ t_{n1} \ldots t_{nk}$$

The type variables $u_1 \ldots u_k$ must be distinct and may appear in the t_{ij}. It is a static error for any other type variable to appear in the t_{ij}.

For example, the declaration

```
datatype Tree a = Leaf a | Branch (Tree a) (Tree a)
```

introduces a new type constructor `Tree` and constructors `Leaf` and `Branch` with types

$$\begin{aligned} &\texttt{Leaf} &&:: \forall a.a \rightarrow \texttt{Tree}\ a \\ &\texttt{Branch} &&:: \forall a.\texttt{Tree}\ a \rightarrow \texttt{Tree}\ a \rightarrow \texttt{Tree}\ a \end{aligned}$$

8.5.2.2 Type Synonym Declarations

$$\begin{array}{lcll} \textit{topdecl} & \rightarrow & \texttt{type}\ \textit{simple} = \textit{type} & \\ \textit{simple} & \rightarrow & \textit{tycon}\ \textit{tyvar}_1 \ldots \textit{tyvar}_k & (\text{arity}\ \textit{tycon} = k,\ k \geq 0) \end{array}$$

A type synonym declaration introduces a new type which is equivalent to an old type and has the form

$$\texttt{type}\ T\ u_1 \ldots u_k = t$$

which introduces a new type constructor T. The type $(T\ t_1 \ldots t_k)$ is equivalent to the type $t[\![t_1/u_1, \ldots, t_k/u_k]\!]$. The type variables u_i must be distinct and are scoped only over t. It is a static error for any other type variable to appear in t. As in Haskell, mutually recursive type synonyms are disallowed. In essence **type** declarations declare a new name for an existing type. **datatype** declarations create a brand new type.

8.5.3 Type Classes and Overloading

8.5.3.1 Class Declarations

$$\begin{array}{lcll} \textit{topdecl} & \rightarrow & \texttt{class}\ [\textit{context} \texttt{=>}]\textit{class}\ [\texttt{where}\ \{\ \textit{cbody}[;]\}] & \\ \textit{cbody} & \rightarrow & \textit{csigns} & \\ \textit{csigns} & \rightarrow & \textit{csign}_1; \ldots; \textit{csign}_n & (n \geq 1) \\ \textit{csign} & \rightarrow & \textit{vars} :: [\textit{context} \texttt{=>}\ \textit{type} & \\ \textit{vars} & \rightarrow & \textit{var}_1, \ldots, \textit{var}_n & (n \geq 1) \end{array}$$

A *class declaration* introduces a new class and the operations on it. A class declaration has the general form:

$$\texttt{class}\ C\ u\ \texttt{where}\ \{\ v_1 :: c_1 \texttt{=>}\ t_1; \ldots;\ v_2 :: c_2\ \}$$

This introduces a new class name C; the type variable u is scoped only over the method signatures in the class body. The class declaration introduces new *class* methods $v_1, \ldots, v_n$ whose scope extends outside the **class** declaration.

Two classes in scope at the same time may not share any of the same methods as otherwise an overloaded method could not be attributed uniquely to a class.

8.5.3.2 Instance Declarations

$$\begin{array}{lcll} \textit{topdecl} & \rightarrow & \texttt{instance}\ [\textit{context} \texttt{=>}]\textit{tycls}\ \textit{inst}\ [\texttt{where}\ \{\ \textit{valdefs}\ [;]\ \}] & \\ \textit{inst} & \rightarrow & \textit{type} & \\ & & & \\ \textit{valdefs} & \rightarrow & \textit{valdef}_1; \ldots; \textit{valdef}_n & (n \geq 1) \end{array}$$

An *instance declaration* introduces an instance of a class.

8.5.4 Transformation Declarations

FSC also has top level *transformation declarations.* However, these form a supershell to FSC and are discussed in the next chapter.

8.5.5 Nested Declarations

The following declarations may be used in any declaration list, including at the top level of a module.

8.5.5.1 Type Signatures

$$\begin{array}{lcll} decl & \rightarrow & vars :: type & \\ vars & \rightarrow & var_1, \ldots, var_n & (n \geq 1) \end{array}$$

A type signature specifies the types for variables. A type signature has the form:

$$x_1, \ldots, x_n :: t$$

which is equivalent to asserting $x_i :: t$ for each of the i. Each x_i must have a value binding in the same declaration list that contains the type signature; i.e. it is illegal to give a type signature for a variable bound in an outer scope. Moreover, it is illegal to give more than one type signature for a variable.

As mentioned earlier, every type variable appearing in a signature is universally quantified over that signature, and hence the scope of the type variable is limited to the type signature that contains it.

A type signature for x may be more (or less) specific than the principal type derivable from the value binding of x, but it is an error to give a type that is incomparable to the principal type. If a more specific type is given then all occurrences of the variables must be used at the more specific type, or at a more specific type still. The aim of this feature is to allow the user to offer a type signature without having to provide type class constraints.

8.5.5.2 Function and Pattern Bindings

$$\begin{array}{lcl} decl & \rightarrow & valdef \\ & | & arraydecl \\ \\ valdef & \rightarrow & lhs \texttt{ = } exp \texttt{ [where \{ } decls[;]\texttt{\}]} \\ & | & lhs\ gdrhs \texttt{ [where \{ } decls[;]\texttt{\}]} \\ \\ lhs & \rightarrow & pat \\ & | & funlhs \\ funlhs & \rightarrow & var \texttt{ \{ } apat_1, \ldots, apat_k\texttt{\}\{(\ \{ } apat_1, \ldots, apat_k\texttt{\}|}apat\texttt{)\}} \\ & | & apat\ apat\ \{apat\} \\ \\ gdrhs & \rightarrow & gd \texttt{ = } exp[gdrhs] \\ gd & \rightarrow & \texttt{|}\, exp \end{array}$$

We distinguish two cases within this syntax: a *pattern* binding occurs when the *lhs* is *pat*; otherwise, the binding is called a *function binding*. Binding may either appear at the top level of a module, or within a `where` or `let` construct.

8.5.5.2.1 Function Bindings A function binding binds a variable to a function value. The general form of a function binding for variable x is:

$$\begin{array}{llll} x & p_{11} & \ldots & p_{1k} \quad match_1 \\ \ldots & & & \\ x & p_{n1} & \ldots & p_{nk} \quad match_n \end{array}$$

where each p_{ij} is a pattern, and each $match_i$ is of the general form:

$$= c \texttt{ where } \{decls\}$$

or

$$\begin{array}{l} \mid g_{i1} \quad = c_{i1} \\ \ldots \\ \mid g_{im} \quad = c_{im} \\ \qquad \texttt{where } \{decls_i\} \end{array}$$

The set of patterns corresponding to each match must be linear (no variable is allowed to appear more than once in the entire set) (Fig. 8.24).

Translation:	The general binding form for functions is semantically equivalent to the equation:
	$x\ x_1\ x_2\ \ldots\ x_k = \texttt{case } (x_1, x_2, \ldots, x_k) \texttt{of} \quad \begin{array}{l} (p_{11}, p_{12}, \ldots, p_{1k}) match_1 \\ \ldots \\ (p_{m1}, p_{m2}, \ldots, p_{mk}) match_m \end{array}$
	where the x_i are new identifiers.

Figure 8.24: Translation rule for function bindings

8.5.5.2.2 Pattern Bindings A pattern binding binds variables to values. A *simple* pattern binding has the form $p = e$. The general form of a pattern binding is p *match*, where *match* is the same structure as for function bindings above; i.e.

$$\begin{array}{lll} p & \mid g_1 & = e_1 \\ & \mid g_2 & = e_2 \\ & \ldots & \\ & \mid g_m & = e_m \\ & & \texttt{where } \{decls_i\} \end{array}$$

which is semantically equivalent to:

```
p = let decls in
    if g1 then e1 else
    if g2 then e2 else
    ...
    if gm then em else error "Unmatched pattern"
```

Since FSC does not have the concept of *lazy binding* it is suggested that refutable patterns are not used in pattern bindings, although this is not enforced. That is, it is considered dangerous to use a pattern involving a constructor if that constructor does not span the datatype.

8.5.6 Array Declarations

arraydecl	→	`array` [var_1 `[`[*bnd*]`..`*bnd*`]`[*ordering*][*mutexp*]`where`]	*vardecls*
vardecls	→	{*vardecl*; ...; *vardecl*}	
vardecl	→	var_1 [p_1 , ... ,p_n] *grhs*[`where` {*decls*[;]}]	
	|	var_1 [p_1 , ... ,p_n]`=` *exp*[`where` {*decls*[;]}]	
ordering	→	`ordered` *ords*	
mutexp	→	`overwrites` var_2	($var_1 \neq var_2$)
ords	→	*ord* {(`then`|`and`)*ords*}	
ord	→	`by` *var* `in` *exp iteration*	
	|	*iteration*	
bnd	→	($exp_1, \ldots, exp_n$)	($n \geq 1$)
	→	*exp*	

An alternative to an array comprehension is an array declaration. Whether to use array declarations or array comprehensions is often a matter of taste, as many forms are equivalent.

```
>    array x[i] = a[i]
```

The extent of i is implicitly quantified by the bound of array a. This quantification may be made explicit:

```
>    array x[1..10] where x[i] = a[i]
```

Non-recursive array declarations are translated as follows

`array` x`[`$i_1, \ldots, i_N$`]=` e

is equivalent to

x`= [...[`e`| all` i_N`]...| all` x_1`]`

If the quantification is made explicit the *all*-comprehensions above are replaced by *in*-comprehensions. Array definitions may be written in the same manner as functions, with recursive definitions being allowed. An example would be a wave-front computation:

```
array A[(1,1)..(N,N)] where
                              A[1,j] = 1
                              A[i,1] = 1
                              A[i,j] = A[i,j-1] + A[i-1,j-1] + A[i-1,j]
```

Note: Recursive array definitions cannot be implicitly bounded. These declarations are transformed into a worker function which recursively passes a data array (and possibly a tag array) in a single-threaded manner and a wrapper which returns only the array component of the computation. For further discussion of recursive array declarations see Chapter 7, where this process is discussed at length.

8.5.7 Input/Output

The I/O system in FSC is purely functional, yet has the expressive power of conventional language I/O systems. This is achieved via the use of a *monad* to integrate the I/O operations into a purely functional context.

The I/O monad used by FSC mediates between the values natural to a functional language and the actions which characterise I/O operations and imperative programming in general. The order of evaluation of expressions in FSC is constrained only by data dependencies; an implementation has a great deal of freedom in choosing this order. However, actions must be ordered in a well-defined manner for program execution *and I/O in particular* to be meaningful. The type of `main`, and hence the type of the program, must match the signature $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \mathrm{IO}(\beta_1, \ldots, \beta_m)$, where $m \geq 0$ and $n \geq 0$. In addition there is a special value `argv` :: [*String*] which may be used in `main` to access command line arguments. The number of command line arguments is the length of the array `argv`.

The arguments to `main` are interpreted as initialisation arguments to be given at runtime in FSC syntax.

8.5.8 Static Semantics of Functional Pattern Bindings

The static semantics of function and pattern binding lists is discussed in this section.

8.5.8.1 Dependency Analysis

As with Haskell, in general the static semantics are given by the normal Hindley-Milner inference rules, except that a *dependency analysis transformation* is first performed to enhance polymorphism, as follows. Two variables bound by value declarations are in the same *declaration group* if either

1. they are bound to the same pattern binding, or
2. their bindings are mutually recursive (perhaps by some other declarations which are also part of the group).

Application of the following rules causes each `let` or `where` construct to bind only the variables in a single declaration group, thus capturing the required dependency analysis:

1. The order of the declarations in a `where/let` construct is irrelevant.

2. `let` $\{d_1; d_2\}$ `in` $e \equiv$ `let` $\{d_1\}$ `in` (`let` $\{d_2\}$ `in` e)
(when no identifier bound in d_2 appears in d_1).

8.5.8.2 Type Inference

The type inference used by FSC is described in Chapter 7, with the following post-inference check:

- All recursive bindings (bindings inside a letrec) must be proved to have a type which is a specialisation of $\alpha \rightarrow \beta$.

The consequence of this is that

```
f x = f x
```

is a valid (albeit foolish) FSC declaration as its type is $\forall\alpha, \beta.\alpha \rightarrow \beta$. However the same declaration after η-reduction:

```
f = f
```

is not valid as its type is $\forall\alpha.\alpha$. This measure is to prevent non terminating computations. All recursive arrays are translated to recursive functions and so this measure does not effect them.

8.6 Modules

A module defines a collection of values, datatypes, synonyms, classes and transformations[4], and exports some of these resources, making them available to other modules. We use the term *entity* to refer to the values, types and classes defined in, and perhaps exported from, a module.

Each FSC program is a collection of modules, one of which, like Haskell, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in the module `Main`, and `main` must have type

$$\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \texttt{IO}(\beta_1, \ldots, \beta_m) \qquad (n, m \geq 0)$$

(see Chapter 7).

Modules may reference other modules via explicit `import` declarations, each giving the name of the module to be imported, specifying its entries to be imported. Unlike Haskell, modules **may not** be mutually recursive for compiler and specialisation simplicity.

The name-space for modules is flat, with each module being associated with a unique module name of lexical type *conid*.

8.6.1 Overview

Like Haskell, a module consists of an *interface* and an *implementation* of that interface. The interface provides complete information about the static semantics of that module, including type signatures, type declarations and class definitions. If a module M imports modules $M_1, M_2, \ldots, M_n$ then all information needed to compile M to an object is provided in $M_1, M_2, \ldots, M_n$.

[4] For transformations see Chapter 9.

8.6.2 Module Implementations

A module implementation defines a mutually recursive scope containing declarations for value bindings, data types, type synonyms, classes, etc.

module	→	**module** *modid* [*exports*]**where** *body*	
	|	*body*	
body	→	{[*impldecls* ;][[*fixdecls* ;]*topdecls* [;]]}	
	|	{[*impldecls* [;]}	
modid	→	*conid*	
impdecls	→	$impdecl_1; \ldots; impdecl_n$	$(n \geq 1)$
topdecls	→	$topdecl_1; \ldots; topdecl_n$	$(n \geq 0)$

A module implementation begins with a header: the keyword **module**, the module name, and a list of entities (enclosed in round parentheses) to be exported. The header is followed by an optional list of **import** declarations that specify the modules to be imported. This is followed by an optional list of fixity declarations and the module body. The module body is simply a list of top-level declarations.

As in Haskell, an abbreviated form of module is permitted, which consists only of the module body. If this is used the header is assumed to be **module Main where**.

8.6.2.1 Export Lists

exports	→	$(export_1, \ldots, export_n)$	$(n \geq 1)$
export	→	*entity*	
	|	*modid*	
entity	→	*var*	
	|	*tycon*	
	|	*tycon*(..)	
	|	$tycon(con_1, \ldots, con_n)$	$(n \geq 1)$
	|	*tycls*	
	|	*tycls*(..)	
	|	$tycls(var_1, \ldots, var_n)$	$(n \geq 0)$

An *export list* identifies the entities to be exported by a module declaration. If the export list is omitted than all top-level declarations defined in that module are exported.

The FSC export system is similar to that found in Haskell and similarly the exported definitions must possess closure, e.g. it would be a static error to export a function `f::a -> Tree` and not export the datatype `Tree`.

8.6.2.2 Import Declarations

$$impdecl \rightarrow \texttt{import}\ modid$$

The entities exported by a module are brought into scope with an `import` declaration.

8.6.2.3 Abstract Datatypes

As in Haskell abstract datatypes are constructed via the use of the module system to export a datatype but not its constructors. This datatype is accessed though the functions which are also exported from the module.

8.6.2.4 Fixity Declarations

$$\begin{array}{lcl} fixdecls \rightarrow fix_1; \ldots; fix_n & & (n \geq 1) \\ fix & \rightarrow & \texttt{infix}\ [integer]vars \\ & | & \texttt{infixl}\ [integer]vars \\ & | & \texttt{infixr}\ [integer]vars \\ & | & \texttt{prefix}\ [integer]vars \\ & | & \texttt{suffix}\ [integer]vars \\ \\ vars & \rightarrow & var_1, \ldots, var_n \end{array}$$

A fixity declaration gives a fixity and a binding precedence of a set of operators. Fixity declarations may appear only at the start of a module and may only be given for identifiers defined in that module. Fixity declarations may not be overwritten and an identifier may only have one fixity declaration.

There are five kinds of fixity, non-, left-, right-, pre- and post-, and an integer precedence with a range of at least [0..1000]. If the precedence is omitted a precedence of 1000 is assumed.

8.6.2.4.1 Example The infamous factorial example may be written:

```
>     suffix !
>    (!) :: INT -> INT

>    0! = 1
>    n! = product[1..n]
```

where `product` is defined in the standard prelude (see Appendix H).

Chapter 9

Implementation of FSC

In this chapter we discuss how various aspects of FSC may be implemented.

9.1 2^{nd}-order Lambda Calculus

As part of compilation it is suggested that the code be transformed into a form similar to an extended second-order lambda calculus[56]. That is, types are treated as first-order objects and identifiers are explicitly typed, i.e. rather than

$$\begin{array}{lll} \text{Identifier} & x & \\ \text{Expression} & \epsilon & ::= \quad x \\ & & \mid \quad \epsilon_1\ \epsilon_2 \\ & & \mid \quad \lambda x.\epsilon \end{array}$$

we have as our calculus

$$\begin{array}{lll} \text{Identifier} & x:\tau & \\ \text{Expression} & \epsilon & ::= \quad x:\tau \\ & & \mid \quad \epsilon_1\ \epsilon_2 \\ & & \mid \quad \lambda x.\epsilon \\ & & \mid \quad \Lambda t.\epsilon \\ & & \mid \quad e\ [\tau] \end{array}$$

where t is a type variable and τ is a type. The code

```
>     f :: a -> a
>     f y = id y
```

is translated to

$$\texttt{f} = \lambda\texttt{y.id y}$$

and then to

$$\texttt{f} : \forall\alpha.\alpha \rightarrow \alpha = \Lambda\alpha.(\lambda\texttt{y}.((\texttt{id} : \forall\beta.\beta \rightarrow \beta)[\alpha])\ \texttt{y} : \alpha)$$

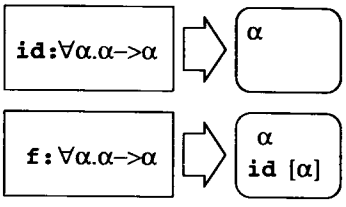


Figure 9.1: Needs analysis 1

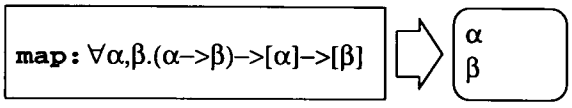


Figure 9.2: Needs analysis 2

This use of Λ also matches our view of polymorphism in that we view `f` to be a function which must be made monomorphic before being evaluated. The use of this extended calculus allows us to reserve type information after type inference and overloading resolution.

9.1.1 Needs Analysis

During compilation, a database of dependencies should be kept. We call the building of this database *needs analysis*. Needs analysis is the creation of a parameterised database entry for each identifier, and type, which details the entities which are needed to evaluate it. **Note:** These identifiers should only be the *free* variables within a function body and variables bound by lambda abstractions should not be entered into the database. The identifier itself should not be entered into the database on multiple occasions unless it has different types bound to it. That is, a recursive function such as `map` should not have any identifier needs but a method instance of a type class which calls a different instance of the same method should. Figs. 9.1, 9.2 show the entries for `id`, `f` and `map` respectively. The function

```
>          instance PLUS(a,b,c) => PLUS((a,a),(b,b),(c,c)) where
>              (a1,a2) + (b1,b2) = (a1+b1,a2+b2)
```

has an entry shown in Fig. 9.3. With an abstract version of the source code, these entries may

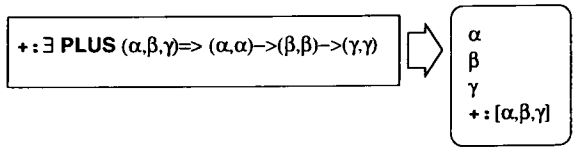


Figure 9.3: Needs analysis 3

be used to generate specialisations of functions *on demand* which can be appended to an existing library. The entry in Fig. 9.3 may be read as "to build an instance of plus which adds pairs of type a to pairs of type b returning pairs of type c there must exist the types a, b and c and an instance of addition from a and b to c".

9.1.2 Boxing

Because of the emphasis we place on efficiency, boxing should not be used in an implementation and specialisations should be generated. Results in Chapter 5 show this not to be a problem in the numerical domain.

9.2 Target Language

It is suggested that FSC be compiled to SISAL, or more specifically SISAL 90, as:

- SISAL has features similar to the FSC kernel.
- Mature implementations exist on architectures including PC's, shared memory multi-processors and vector architectures.
- Experimental implementations exist for distributed memory machines.
- SISAL is implicitly parallel.
- Efficiency is comparable with C/Fortran on uniprocessors and often better than C on multi-processors [18].
- SISAL has iteration constructs so transformations such as the linear expansion theorem (LET) [32] may be viewed as source to source.
- SISAL has a primitive foreign language interface.
- SISAL90 has all the features needed for a simple kernel language including higher-order functions, case expressions and user-defined reductions.

Interest in SISAL itself seems to be dwindling. However, currently, there seems to be much interest in the use of SISAL as an intermediate parallel language for compiler construction [7]. Although SISAL is an applicative language, its foreign language interface allows the use of non-pure functions. In FSC this feature is not available to the user although this is wrapped up to provide I/O. We suggest that FSC be compiled down into an efficient SISAL kernel in a manner similar to that described in Chapter 8.

9.3 Input/Output

Input to and output from FSC programs is handled via the generation of a parser and a printer. If an FSC program has the type

```
>          main :: Int -> Tree INT -> IO (Tree DOUBLE)
```

i.e. it reads and writes the user defined datatype `Tree`

```
>          datatype Tree a = Leaf a | Branch (Tree a) (Tree a)
```

then the implementation should generate a parser accepting an integer followed by a tree of integers (on stdin) and should also generate a pretty printer to display a tree of doubles.

9.3.1 Format

If `fsc.out` is an executable with an entry point `main`

```
>          main :: x -> IO (x)
```

for some monomorphic type `x` then program-output is grammatically correct program-input (provided that there is no I/O within the program body). Moreover, the I/O end points are grammatically correct FSC-expressions of type `x`.

9.4 Parallelism

Since SISAL is an implicitly parallel language it would be foolish for FSC not to try to retain this implicit parallelism. In SISAL, functions are split into two categories

- Reductions
- Normal (non-reducing) Functions

Reductions accept the output of an iteration and reduce it, in parallel, to a single value. In SISAL these functions cannot be applied simply to arguments and, similarly, functions cannot be applied to the result of an iteration. FSC has no concept of a *reduction* and all functions may be applied to the output of an iteration. However in order to harness SISAL reductions it is suggested that applications of the function `reduce :: (a -> a -> a) -> (() -> a) -> [a] -> a` be used to denote reductions, that is, the definition

```
>          sum :: PLUS(a,a,a),ADDID(a) => [a] -> a
>          sum = reduce (+) zero
```

defines a reduction and, if applied to an iteration, will reduce the iteration in parallel using addition, otherwise it will create an iteration across its argument and reduce this in parallel. Unfortunately this relys on the existence of user defined reductions and hence requires translation to SISAL90.

9.5 Higher Order Functions

It is suggested that higher order functions be in-lined as much as possible as these constitute a major efficiency overhead. Both recursive and non-recursive parameterisation[1] may be easily compiled out via partial evaluation. That is, the function

```
>          map :: (a->b) -> [a] -> [b]
>          map f xs = [f x | x in xs]
```

which we regard as being non-recursively parameterised, may be in-lined, and the function

```
>          map :: (a->b) -> [a] -> [b]
>          map f []         = []
>          map f (x <+ xs) = f x <+ map f xs
```

which we regard as being recursively parameterised, may be specialised around the function `f`.

9.5.1 Non-Parameterising Higher Order Functions

Occasionally an example of a non-parameterising higher order function arises although it may easily be rewritten. The function

```
>          contrived :: (INT->INT) -> (INT->INT) -> (INT->INT)
>          contrived f g | f 0 == 1 = contrived g (f.g)
>                        | f 1 == 1 = contrived (g.g.f) (g.f)
>                        | otherwise = (g.f)
```

is such an example. Pointers to these functions may be stored in a closure together with an environment in which it has been partially applied. However, we do not envisage the need to generate this type of code very often as the type of example given above does not frequently occur in our domain of interest. Alternatively higher order functions may be implemented directly if SISAL 90 is used as the resulting intermediate language.

9.6 Specialisation and Separate Compilation

The use of specialisation as a method for compiling polymorphic functions is easy to understand. However, specialisation relies on *a priori* -knowledge of the complete call path for an executable.

[1] **Definition:** A Parameterising function is a function in which all recursive function applications withing the body exhibit at least one constant argument which itself is a parameter to the function. This argument must be at the same position for all applications and definitions.

```
>          id :: a -> a
>          id x = x

>          f :: a -> a
>          f y = id y

>          main :: INT -> IO INT
>          main x = do IO {return id (f x)}
```

- An implementation based on specialisation initially checks that `main` has a monomorphic type and generates a set containing the single element {`main::INT-> IO INT`}.
- Next, it adds to the set the specialisations needed by the elements already contained in the set.
- This last step repeats until the process converges on a fixed point.

This fixed point is the set of required specialisations. In the above example the set is initially

$$S = \{\texttt{main::}\ \ \texttt{INT -> IO INT}\}$$

Next the functions called by `main` are added

$$S = \{\texttt{main::}\ \ \texttt{INT -> IO INT}, \texttt{f::INT -> INT}, \texttt{id::INT->INT}\}$$

and the functions called by `f` and `id`. In this example the only extra function called by either of these is `id`, which is called by `f`. More specifically, the polymorphic function

$$f :: \forall\alpha.\alpha \rightarrow \alpha$$

makes a call to the monomorphic function

$$id :: \alpha \rightarrow \alpha$$

The type instantiated on f must be carried through to `id` and since `main` requires an instance of `f` with its type instantiated to `INT -> INT` it also requires an instance of `id` with its type instantiated to `INT -> INT`. Since such an instance of `id` is already in the set, and since there are no other calls in the tree, we have reached the fixed point and the set of required instances is

$$S = \{\texttt{main::}\ \ \texttt{INT -> IO INT}, \texttt{f::INT -> INT}, \texttt{id::INT->INT}\}$$

9.6.1 Separate Compilation

Specialisation is not directly compatible with separate compilation. If the example from the previous section was divided into three files, one for each function, then only the file containing `main` could be compiled independently of the others. This tends to suggest that the use of libraries containing polymorphic functions would prove problematic if not impossible. This issue is tackled in the next nextion.

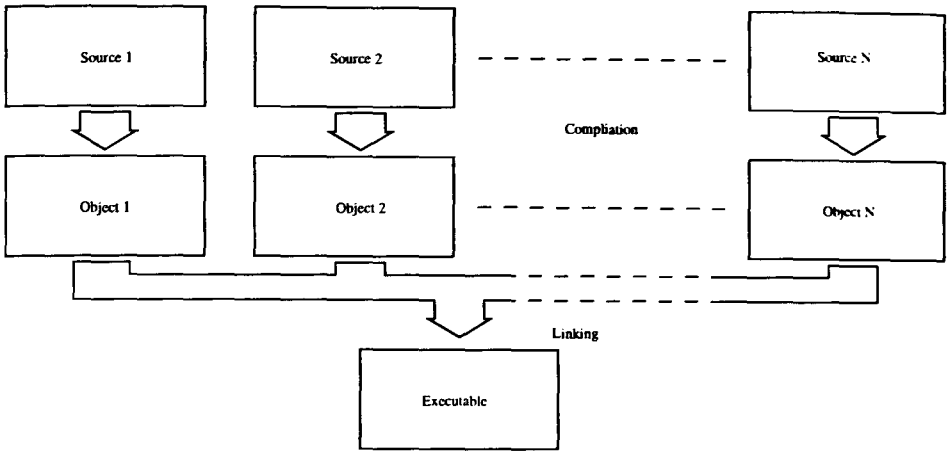


Figure 9.4: Standard compilation

9.7 Library-Based Separate Compilation

In the previous section we identified the problem which specialisation poses. This arises from considering separate compilation as divided object-file-wise (Fig. 9.4). If, however, we consider compilation to be arranged into libraries rather than objects we may initially start with an empty library and incrementally add to it provided we have some abstract code with enough information to generate further specialisations (Fig. 9.5). Linkage of precompiled objects may update these objects and may extend the number of specialisations in a library. At this point, of course, no type checking is necessary as all the source is in an abstract, typed form.

9.8 Transformations and Derivations

In this section the nature of transformations used in the FSC system is discussed. However, before discussing how such a system may be implemented we discuss its properties and the features it should support. The following features form part of the FSC language, however their development is currently not part of any compiler. To a large extent this section could be considered as further work. We include it here however as it is part of the FSC definition with only the implmentation and usefulness in practice requiring further investigation.

9.8.1 Rewrite Rules

The form that transformations in FSC take is similar to other systems [15, 14, 33] in that they consist of a series of rewrite rules of a *pattern* to match and a *replacement* expression. For example:

$$\mathsf{TRANSFORM}(x_{\text{meta}})[\![x_{\text{meta}} \wedge \text{True}]\!] \Rightarrow [\![x_{\text{meta}}]\!]$$

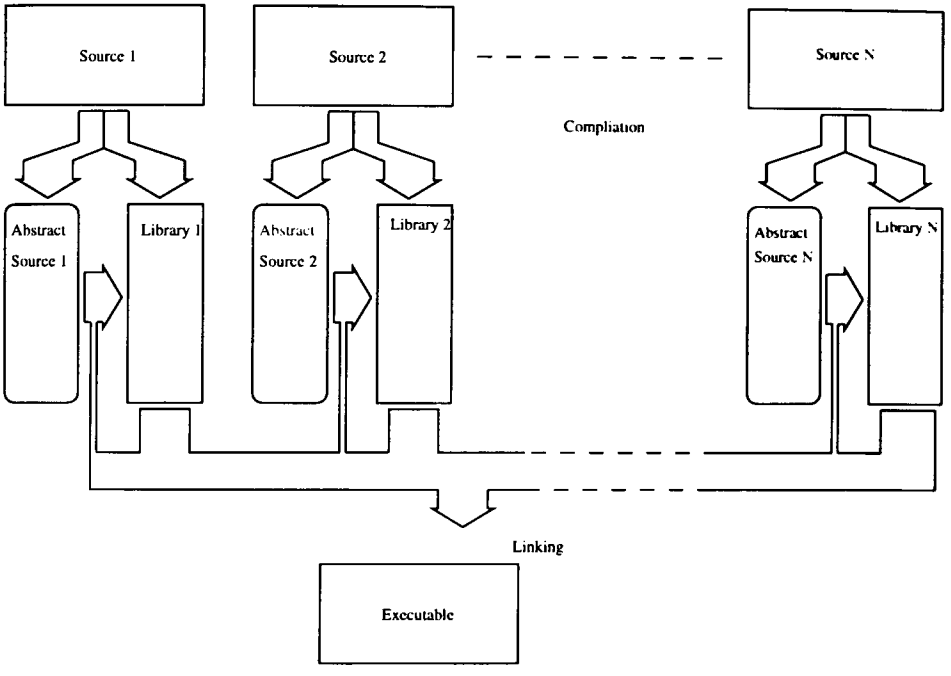


Figure 9.5: FSC compilation strategy

and

$$\mathsf{TRANSFORM}(x_{\text{meta}})[\![x_{\text{meta}} * 1]\!] \Rightarrow [\![x_{\text{meta}}]\!]$$

both form valid transformations. The identifiers in parentheses denote variables in the transformation which *must* be bound to expressions before they are used. Any free variables relate to globally scoped identifiers. The following is illegal

$$\mathsf{TRANSFORM}(x_{\text{meta}})[\![0]\!] \Rightarrow [\![x_{\text{meta}}]\!]$$

since x does not have a value bound to it before it is used. For convenience, in the rest of the discussion we shall omit the meta-tag notation. To guarantee the preservation of correctness, the following property must hold for all expressions e

$$\mathsf{EVAL}(\mathsf{TRANSFORM}(x_1, \ldots, x_n)[\![pat.]\!] \Rightarrow [\![replacement]\!](e)) = \mathsf{EVAL}(e)$$

i.e. a transformation must not alter the value associated with an expression. This property must be checked by the user when defining transformations. Unfortunately, demanding that transformations exhibit this property in a *strict* semantic framework prevents us from defining useful transformations such as

$$\mathsf{TRANSFORM}(x)[\![x \wedge \mathrm{False}]\!] = [\![\mathrm{False}]\!]$$

or

$$\mathsf{TRANSFORM}(x)[\![x * 0]\!] \Rightarrow [\![0]\!]$$

However a weaker version of the above property can be used which allows such transformations. This rule is that the transformation should not reduce the information content of an expression:

$$\mathsf{EVAL}(e) \preceq \mathsf{EVAL}(\mathsf{TRANSFORM}(x_1, \ldots, x_n)[\![pat.]\!] \Rightarrow [\![repl.]\!](e))$$

or, more precisely

$$\mathsf{EVAL}_{\text{applic.}}(e) \preceq \mathsf{EVAL}(\mathsf{TRANSFORM}(x_1, \ldots, x_n)[\![pat.]\!] \Rightarrow [\![repl.]\!](e)) \preceq \mathsf{EVAL}_{\text{lenient}}(e)$$

That is, if the expression fails to terminate under applicative order evaluation but terminates under lenient order evaluation a transformation may convert it into an expression which terminates under applicative order evaluation. This is shown in Fig. 9.6 where three examples of spurious trasformations are shown. Two of these transformations are invalid (bold top and middle) due to the fact they transform one value to a completely different value. The other (bold bottom) is spurious as it reduces the information content in an expression.

The FSC transformation system may be regarded as an interpreted language which executes at compile time. More precisely, the transformation system in FSC transforms 2^{nd}-order extended lambda calculus expressions to another 2^{nd}-order extended lambda calculus expression and since FSC is a heavily overloaded language, *must* preserve type information.

Our first example

$$\mathsf{TRANSFORM}(x)[\![x \wedge \mathrm{True}]\!] \Rightarrow [\![x]\!]$$

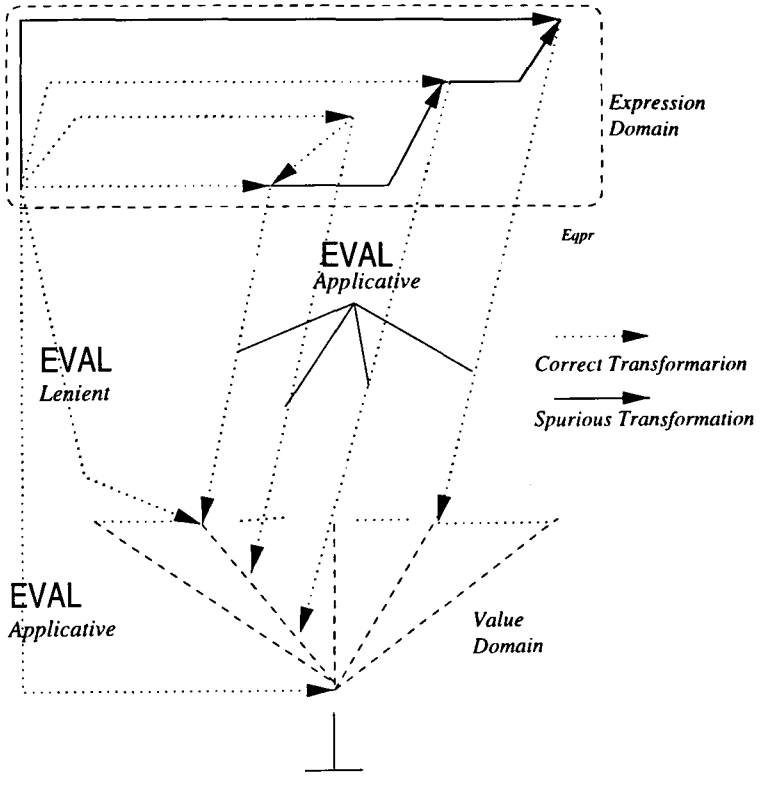


Figure 9.6: Correct and spurious transformations

is actually a sugared version of the following transformation

$$\begin{array}{lll} \mathsf{TRANSFORM}(x, \tau_1)[\![x \wedge \mathrm{True}]\!] \mid & [\![x]\!] :: \tau_1 \\ \wedge & [\![x \wedge \mathrm{True}]\!] :: \tau_1 \\ \Rightarrow & [\![x]\!] \end{array}$$

A slightly more complex example is the solution of a linear system

$$\mathsf{TRANSFORM}(A, x)[\![A^{-1} * x]\!] \Rightarrow [\![\mathrm{solve}(A, x)]\!]$$

which is translated to

$$\begin{array}{lll} \mathsf{TRANSFORM}(A, x, \tau_1, \tau_2, \tau_3)[\![A^{-1} * x]\!] \mid & [\![A]\!] :: \tau_1 \\ \wedge & [\![x]\!] :: \tau_2 \\ \wedge & [\![A^{-1} * x]\!] :: \tau_3 \\ \wedge & \exists [\![\mathrm{solve}]\!] :: (\tau_1, \tau_2) \rightarrow \tau_3 \\ \Rightarrow & [\![\mathrm{solve}(A, x)]\!] \end{array}$$

That is, all subexpressions should have a type associated with them and all newly introduced free variable identifiers should have their existence checked.

9.8.2 Predicates

There is a set of predicates associated with transformations to test expressions {id,const,app,...} which can be used to test individual terms. New predicates may be introduced as follows:

$$\mathsf{PREDICATE}(x)[\![simple(x)]\!] \Rightarrow var(x) \vee const(x)$$

$$\mathsf{PREDICATE}(x)[\![anything(x)]\!] \Rightarrow \mathrm{True}$$

9.8.3 Evaluation

EVAL declares we trust a function and that function should be evaluated at compile time were it to be applied to constant arguments. For example,

$$\mathsf{EVAL}[\![+ :: INT \rightarrow INT \rightarrow INT]\!]$$

declares that addition of integers is a *safe* function to evaluate.

9.8.4 Derivations

Transformations of the same level are applied in textual order and transformations of different levels are applied in level order, i.e. it is possible to interpret levels as priorities. Composite transformations

may also be built up by enclosing a list of transformations in braces and the transformations will be composed, that is,

$$\begin{array}{l}\mathsf{TRANSFORM}\\ \{\\ \mathsf{TRANSFORM}(x_1,\ldots,x_n)[\![pat_1]\!] \Rightarrow [\![repl._1]\!];\\ \mathsf{TRANSFORM}(y_1,\ldots,y_n)[\![pat_2]\!] \Rightarrow [\![repl._2]\!]\\ \}\end{array}$$

is equivalent to:

$$(\mathsf{TRANSFORM}(y_1,\ldots,y_n)[\![pat_2]\!] \Rightarrow [\![repl._2]\!]) \circ (\mathsf{TRANSFORM}(x_1,\ldots,x_n)[\![pat_1]\!] \Rightarrow [\![repl._1]\!])$$

9.8.5 Subconditions

It is sometimes neater to have subconditions in a transformation, e.g.

$$\mathsf{TRANSFORM}(y,a,x)[\![a*x+a*y]\!] \Rightarrow [\![a*(x+y)]\!]$$

could be extended to encompass a commuting $*$ and written as

$$\begin{array}{rll}\mathsf{TRANSFORM}(x,y,z1,z2,a)[\![x+y]\!] \mid & ([\![a*z1]\!] \texttt{ is } [\![x]\!] \vee [\![z1*a]\!] \texttt{ is } [\![x]\!])\\ \wedge & ([\![a*z2]\!] \texttt{ is } [\![y]\!] \vee [\![z2*a]\!] \texttt{ is } [\![y]\!])\\ \Rightarrow & [\![a*(z1+z2)]\!]\end{array}$$

Which reads: If an expression matches $x+y$ for some x,y then if x matches either $a*z1$ or $z1*a$ for some a, $z1$and y matches either $a*z2$ or $z2*a$ for some $z2$ and for the a bound previously then the complete expression may be replaced with $a*(z1+z2)$. A more general example could be

$$\begin{array}{rll}\mathsf{TRANSFORM}(f,x,y,e,v,x',y')[\![f\ x\ y]\!] \mid & [\![e]\!] \texttt{ in } [\![x]\!]\\ \wedge & [\![e]\!] \texttt{ in } [\![y]\!]\\ \wedge & [\![v]\!] \texttt{ is new}\\ \wedge & [\![x']\!] \texttt{ is } [[\![v]\!]/[\![e]\!]][\![x]\!]\\ \wedge & [\![y']\!] \texttt{ is } [[\![v]\!]/[\![e]\!]][\![y]\!]\\ \Rightarrow & [\![\text{let } v\text{=e in } f\ x'\ y']\!]\end{array}$$

9.8.6 Required Features

To be able to support the above system we need:

- The forms ::, $\exists$, `is` and `in`.
- The notion of substitution.
- The notions of Success and Failure.
- A unique name supply.
- The notion of backtracking *à la* Prolog.

- The ability to compose transformations.
- The "logical" connectives $\neg$, $\wedge$ and $\vee$.

We place the word "logical" in quotes as these connectives do not behave exactly like $\neg$, $\wedge$ and $\vee$ over simple variables. For example within a transformation the statement $\forall a.\neg(\neg a) = a$ is not everywhere valid. However, $\forall a.\neg(\neg(\neg a)) = \neg a$ *is* everywhere valid. The reason for this becomes clear if we view the success state as *success with a set of bindings* and the failure state as *binding-less*. Negation either constructs a trivial success state from failure or constructs failure from a success state. However, for premises τ_1, τ_2 in which no binding occurs

$$\begin{aligned} \neg(\neg\tau_1) &= \tau_1 \\ \neg(\tau_1 \vee \tau_2) &= \neg\tau_1 \wedge \neg\tau_2 \end{aligned}$$

do hold, and for all premises (regardless of binding) the following laws hold:

$$\begin{aligned} (A \wedge B) \vee (A \wedge C) &= A \wedge (B \vee C) \\ (A \wedge B) \wedge C &= A \wedge (B \wedge C) \\ (A \vee B) \vee C &= A \vee (B \vee C) \\ A \wedge \neg A &= \texttt{Failure} \\ A \wedge A &= A \\ A \vee A &= A \\ \neg(\neg(\neg A)) &= \neg A \end{aligned}$$

9.8.7 Translation

The translation of these transformations is easily carried out in terms of a *transforming monad* interpreting each as a T-resultric function, i.e. a transformation is typed

$$Expression \rightarrow T(Expression)$$

for some type constructor T. If we form a monad with a notion of Success/Failure from T, then we may interpret the conjunctions in our transformation as sequencing, i.e.

$$\mathsf{TE}[\![f \wedge g]\!] \Rightarrow \mathsf{TE}[\![f]\!] >> \mathsf{TE}[\![g]\!]$$

and interpret the result similarly

$$\mathsf{TE}[\![f \Rightarrow g]\!] \Rightarrow \mathsf{TE}[\![f]\!] >> \mathsf{TR}[\![g]\!]$$

That is, $(f \wedge g)$ or $(f \Rightarrow g)$ succeeds if f succeeds and then g succeeds. Disjunctions are handled similarly by considering monads with the combining operation $++$,

$$\mathsf{TE}[\![f \vee g]\!] \Rightarrow \mathsf{TE}[\![f]\!] ++ \mathsf{TE}[\![g]\!]$$

That is, $(f \vee g)$ can only fail if both f and g fail. Here the combinators $>>$ and $++$ are typed

$$++ :: T\ a \rightarrow T\ a \rightarrow T\ a$$

and

$$>>:: T\ a \rightarrow T\ b \rightarrow T\ b$$

We give a concrete definition of these presently. Interestingly we find that Kleisli composition is also useful in this area:

$$\mathsf{TE}[\![\mathsf{TRANSFORM}\{t_1;\ldots;t_n\}]\!] \Rightarrow \mathsf{TE}[\![t_n]\!] \star \cdots \star \mathsf{TE}[\![t_1]\!]$$

We may model pattern matching and the introduction of variables via the addition of the functions `variables` and `match`, which introduce variables and bind expressions to patterns respectively.

$$\begin{aligned}\mathsf{TE}[\![\mathsf{TRANSFORM}(x_1,\ldots,x_n)[\![pat.]\!]|guards \Rightarrow [\![repl.]\!]]\!] \\ \Rightarrow\ \lambda x.\ \ \texttt{variables}\ [x_1,\ldots,x_n]\ \ >> \\ \texttt{match}\ \mathsf{TP}[\![pat.]\!]\ x\ \ >> \\ \mathsf{TE}[\![guards \Rightarrow [\![repl.]\!]]\!]\end{aligned}$$

We also have a function **subExps** which binds sub-expressions to identifiers and we have the following translation

$$\mathsf{TE}[\![[\![x]\!]\ \texttt{in}\ [\![y]\!]]\!] \Rightarrow \texttt{subExps}\ \mathsf{TP}[\![x]\!]\ \mathsf{TT}[\![y]\!]$$

$$\mathsf{TE}[\![[\![x]\!]\ \texttt{is}\ [\![y]\!]]\!] \Rightarrow \texttt{match}\ \mathsf{TP}[\![x]\!]\ \mathsf{TT}[\![y]\!]$$

where TT evaluates the bound variables and TP evaluates and renames the bound variables so they cannot interfere with program variables.

9.8.8 Building Transformations

Before proceeding further we model our transformation system and build simple expressions. This model takes the form of a Haskell script. Initially we see that a transformation with suitable unit and bind operations forms a monad.

```
type Transformation = Expr -> T Expr
```

The monad is formed from `(T,>>=::T a -> (a -> T b) -> T b,unit::a -> T a)` and we gain Kliesli composition `(.*.)` and sequencing `(>>)` in the usual manner

```
>       (.*.)::(a -> T b ) -> (b -> T c) -> (a -> T c)
>       f .*. g  = \x -> f x >>= g

>       (>>)::(T a) -> (T b) -> (T b)
>       f >>  g  = f >>= \_ -> g
```

9.8.8.1 Transformation Monads

For pragmatic reasons we need to be able to put some upper bound on the amount of time/resources that are allocated on transformation. With this in mind we define `T` as

```
>       type T a = Int -> Int -> ... -> (Int,Int ...  a ... )
```

i.e. an execution count and upper bound are threaded through the computation. In fact we want it also to include an environment, a bound-variable list, a unique variable supply, a dictionary of functions and also model several interpretations of the same transformation. Thus, we define `T` as

```
>       type T a = Count     ->
>                  Bound     ->
>                  Dict      ->
>                  Uniq      ->
>                  BVL       ->
>                  Env       -> ((Dict,Bound),[(Count,Uniq,BVL,Env,a)])
```

9.8.8.2 Building Blocks

Now that we have a simple type which models transformations, we consider how transformations may be constructed. Our first building block is the identity transformation `unitTrans`, i.e. it always succeeds, has only one interpretation, and does not affect an expression

```
>         unitTrans :: a -> T a
>         unitTrans exp = \i j dic u bvl env ->
>                                    check j dic  [(i,u,bvl,env,exp)]
>         check j dic []  = ((dic,j),[])
>         check j dic ((i,u,b,en,ex):xs)
>            = if i < j then let (_,xs') = check j dic xs
>                            in ((dic,j),(i+1,u,b,en,ex):xs')
>                       else check j dic xs
```

We can also define a transformation `zeroT` which always fails:

```
>         zeroTrans :: a -> T a
>         zeroTrans exp = zeroT

>         zeroT :: T b
>         zeroT = \i j dic u bvl env  -> ((dic,j),[])
```

We now look at how transformations may be built from other transformations and from values of type `T a`. We first define the *monadic* combinator `bindTrans`.

```
>         bindTrans :: T a -> (a -> T b) -> T b
>         bindTrans r t =
>           \i j dic u bvl env  ->
>             check dic j (concat [ (snd'.t) e i' j dic  u2 b2 e2
>                 | (i',u2,b2,e2,e) <- (snd' r) i j dic u bvl env])
>         snd' f = \i j d u b e -> case f i j d u b e of (_,x) -> x
```

Thus the triple `(T,bindTrans,unitTrans)` forms a monad-like structure[2] which, after performing a set (normally very large) number of transformations, returns failure.

[2]This is not an actual monad as the pragmatic execution count violates the monad laws. i.e. `bindTrans x unitTrans = x` is not always valid as the last `unitTrans` could cause the execution count to exceed the limit.

9.8.8.3 Success and Failure

Success and failure may be easily coded as:

```
>          successT = unitTrans ()
>          failureT = zeroT
```

9.8.8.4 Negation

Negation may be coded as:

```
>          neg :: T a -> T ()
>          neg t = \i j dic u bvl env -> case  t i j dic u bvl env of
>                                  (_,[]) -> check j dic  [(i,u,bvl,env,())]
>                                  _  -> check j dic   []
```

9.8.8.5 Existence

The existence of an identifier can be checked by looking in the dictionary which is carried throughout the monad. This dictionary should contain all prototypes for all identifiers in scope.

9.8.8.6 Evaluation

Evaluation of functions is carried out by carrying the abstract source for the evaluatable functions in the dictionary and interpreting it at compile time. As such, the abstract source for these functions should be exported as part of the module interface. This interpretation should update the threshold variable to avoid non-termination.

9.8.8.7 Connectives

We define $\vee$, $\wedge$ as:

```
>          t1 .\/. t2 = addTrans t1 t2

>          a ./\. b = bindTrans a (\_ -> b)
>          addTrans :: T a -> T a -> T a
>          addTrans t1 t2
>           = \i j dic u bvl env ->  check j dic  ((snd' t1) i j dic u bvl env
>                                                 ++ (snd' t2) i j dic u bvl env )
```

9.8.8.8 Matchings, Sub-Expressions and Substitutions

The function `match` may be defined as:

```
>         match pat exp  = lookupT pat `bindTrans` \exp1 ->
>                          lookupT exp `bindTrans` \exp2 ->
>                          match2 exp1 exp2
```

```
>       lookupT (App e1 e2) = lookupT e1 'bindTrans' \e1 ->
>                             lookupT e2 'bindTrans' \e2 ->
>                             unitT (App e1 e2)
>       lookupT (Var x)
>        = \i j d u bvl env -> if x 'elem' bvl
>                       then check j dic [(i,u,bvl,env,lookup env x)]
>                       else check j dic [(i,u,bvl,env,BVar x)]
>             ...
>             etc

>       match2 (App e1 e2) (App e3 e4) = match2 e1 e3 ./\. match e2 e4


>       match2  (Var x) (Var y) =  if (x == y) then successT
>                                              else failureT
>       match2  (BVar x) e = addBinding x e

>       ...
>       etc
```

where `addBinding x e` attempts to add the binding $x \mapsto e$ to the environment and either fails or succeeds. Success is returned if there is no earlier binding for x or the earlier binding is e. Subexpressions are constructed in a similar way to matchings although no binding occurs within a subexpression. Similarly, substitutions recursively descend the abstract syntax tree.

9.8.9 Unique Variables

The line

$$[\![x]\!] \texttt{ is new}$$

is translated to

```
>         newVar 'bindTrans' addBinding x
```

where `newVar` is defined as

```
>         newVar = \i j dic u b e -> check j dic [(i,incU u,b,e,mkVarU u)]
```

and `incU` and `mkVarU` increment the variable supply and make a new variable name from the old u value, respectively.

9.9 Application of Transformations

The transformations may be applied in a top down manner to the abstract syntax tree

```
>          applyTopDn :: [Transformation] -> Transformation
>          applyTopDn tlist = applyTopDn2 [] tlist

>          applyTopDn2 ys [] e = unitTrans e
>          applyTopDn2 ys (x:xs) e
>           = \i j dic u b en -> case x e i j dic u b en of
>            (_,[]) -> (applyTopDn (ys ++ [x]) xs e i j dic u [] [])
>            (_,((i,u,bvl,env,ne):_)) -> (
>                                          applyTopDn2 [] (ys++(x:xs))
>                                          ne i j dic u [] []
>                                        )


>          applyTopDn' :: (Exp -> T Exp) -> Exp -> T Exp
>          applyTopDn' t e
>           = t e ./\. case e of
>                    (App e1 e2) -> ((applyTopDn' t e1 'bindTrans' \te1 ->
>                                       unitTrans (App te1 e2))
>                                          ./\.
>                                    (applyTopDn' t e2 'bindTrans' \te2 ->
>                                       unitTrans (App e1 te2)))
>                  ... etc
```

i.e. **applyTopDn** takes a list of transformations and applies the first one in a top down manner to the abstract syntax tree. If this has any effect then the original list is regenerated and the process repeated. However, if this transformation fails then the same process is repeated starting with the next transformation. This continues until all transformations fail (either because they are no longer applicable or because we have exceeded our self imposed limit on operations). Other application schemes, such as bottom-up application, can be written easily.

9.10 Syntax of Transformations

The syntax of transformations is as follows:

topdecls → *transdecl*
| PREDICATE($var_1, \ldots, var_n$) [$var(var_1, \ldots, var_m)$] => *simple*
| EVAL [*var* :: *type*]
| TRANSFORM{*transdecl*{; *transdecl*}}

transdecl → TRANSFORM($var_1, \ldots, var_n$) [*exp*] *trhs*

trhs → |*premises*=> [*exp*]
| => [*exp*]

premises → [~]*premises* {(∨|∧)*premises* }
| *texp*
| (*premises*)

simple → [~]*simple*{(∨|∧)*simple*}
| *var*(*x*)
| (*simple*)

texp → [*exp*] :: *type*
| exist [*exp*] :: *type*
| [*exp*] is [*exp*]
| [*exp*] is [new]
| [*exp*] is [[*exp*]/[*exp*]] [*exp*]
| [*exp*] in [*exp*]
| *var*(*x*)

9.11 Summary

In the last three chapters we introduced the FSC language.

- In Chapter 7 we described our rationale behind the design of FSC and suggested how the features of FSC may be implemented.
- In Chapter 8 we defined the FSC language (excluding its transformational meta-language).
- In this chapter we introduced the transformational meta-language and explained how it could be implemented in a Haskell-like language. We also discussed how some of the implementation dependent features could be realised. Like the language definition of Chapter 8 the transformation description is not meant to be taken as a *real-world* implementation but as a simplified description of how the system could be modelled.

In the next chapter we compare FSC with other languages, concentrating on its expressiveness and conciseness. We express a simple functional finite element code in FSC and demonstrate its expressiveness by presenting examples written in FSC. We also make use of a prototype implementation of the FSC language and present two case studies, comparing its performance with Haskell/C/C++ as a *proof of concept*.

Chapter 10

Use of FSC in Practice

In this chapter we use the FSC language to implement various numerical algorithms and use a prototype compiler to demonstrate that it compares favourably with C/C++/Haskell. We begin by rewriting the SASL finite element example given in [26] in FSC. Originally this was thought clear and concise and we demonstrate how FSC enhances this. We then present various numerical algorithms written in FSC and present in-depth case studies where the clarity, conciseness, and efficiency of FSC programs are compared with their counterparts written in Haskell, C and C++.

10.1 Functional Programming for Finite Elements

This section takes its title from the paper by J.F.Dwyer [26]. We repeat the original commentary and insert FSC below the original SASL implementation. Syntactic convention of SASL which the reader may not be familiar with are shown in Fig. 10.1 with their Haskell translation.

SASL	Haskell
`DIV`	`(/)`
`/`	`[]`
`lambda` x `.` e	`\`x `->` e
p `->` e_1 `;` e_2	`if` p `then` e_1 `else` e_2
`let` x `be` y `in` e	`let` x `=` y `in` e

Figure 10.1: SASL: syntactic conventions

10.1.1 The Finite Element Problem

The example presented is made as simple as possible, to illustrate the concepts of functional programming rather than any advanced finite element theories. The ordinary differential equation to

be solved is

$$\frac{d}{dx}\left(x\frac{dU(x)}{dx}\right) = \frac{2}{x^2}, \qquad a < x < b. \tag{10.1}$$

with given boundary conditions $U(a)$ (essential constrained boundary condition) and

$$(-xdU(x)/dx)_{x=b}$$

(natural unconstrained boundary condition). The finite element formulation given here is that presented in [16]. The domain (a, b) is broken into $(n-1)$ subdomains by the nodal points $x_1, x_2, \ldots, x_n$ and two linear shape functions are used on each subdomain

$$\begin{aligned} \phi_j^{(1)}(x) &= \tfrac{x_{j+1}-x}{x_{j+1}-x_j}, \qquad j = 1, 2, \ldots, n-1 \\ \phi_j^{(2)}(x) &= \tfrac{x-x_j}{x_{j+1}-x_j}, \qquad j = 1, 2, \ldots, n-1 \end{aligned} \tag{10.2}$$

The standard assembly process leads to a tridiagonal symmetric stiffness matrix $\mathbf{K}$, where

$$\begin{aligned} K_{nn} &= \int_{x_{n-1}}^{x_n} x\left(\tfrac{d\phi_{n-1}^{(2)}(x)}{dx}\right)^2 dx \\ K_{jj} &= \int_{x_{j-1}}^{x_j} x\left(\tfrac{d\phi_{j-1}^{(2)}(x)}{dx}\right)^2 dx \\ &+ \int_{x_j}^{x_{j+1}} x\left(\tfrac{d\phi_j^{(1)}(x)}{dx}\right)^2 dx \qquad 2 \le j \le n-1 \\ K_{j,j+1} &= \int_{x_j}^{x_{j+1}} x\left(\tfrac{d\phi_j^{(1)}(x)}{dx}\right)\left(\tfrac{d\phi_j^{(2)}(x)}{dx}\right) dx. \end{aligned} \tag{10.3}$$

The force vector $\mathbf{b}$ is given by

$$\begin{aligned} b_1 &= -\int_{x_1}^{x_2} \tfrac{2}{x^2}\phi_1^{(1)}(x)dx - \left(-x\tfrac{dU(x)}{dx}\right)_{x=a} \\ b_n &= -\int_{x_{n-1}}^{x_n} \tfrac{2}{x^2}\phi_{n-1}^{(2)}(x)dx - \left(-x\tfrac{dU(x)}{dx}\right)_{x=b} \\ b_j &= -\int_{x_{j-1}}^{x_j} \tfrac{2}{x^2}\phi_{j-1}^{(2)}(x)dx \\ &\quad -\int_{x_j}^{x_{j+1}} \tfrac{2}{x^2}\phi_j^{(1)}(x)dx \qquad 2 \le j \le n-1. \end{aligned} \tag{10.4}$$

The natural boundary condition given above can be substituted immediately into the expression for b_n. The usual procedure for handling the essential boundary condition is to multiply the first column of $\mathbf{K}$ by the given value and subtract the result from the force vector and then delete the first row and first column of $\mathbf{K}$. This modification only affects the original b_2 which now becomes

$$b_2 - U(a)K_{12} \tag{10.5}$$

Hence the problem reduces to the solution of an $(n-1)\times(n-1)$ symmetric tridiagonal system. In the interests of space efficiency this matrix can be stored as a vector whose components are determined from the matrix elements in the following way:

$$l_{2j-1} = K_{jj} \qquad 1 \le j \le n$$

$$l_{2j} = K_{j,j+1} \qquad 1 \leq j \leq n-1$$

$$\begin{bmatrix} l_1 & l_2 & & & \\ & l_3 & l_4 & & \\ & & \ddots & \ddots & \\ & & & l_{2n-3} & l_{2n-2} \\ & & & & l_{2n-1} \end{bmatrix}$$

The Gaussian elimination triangulation process for the system reduces to

$$\begin{aligned} l'_1 &= l_1 \\ l'_{2j} &= l_{2j} \qquad 1 \leq j \leq n-1 \\ l'_{2j+1} &= l_{2j+1} - \frac{l^2_{2j}}{l'_{2j-3}} \qquad 1 \leq j \leq n-1 \end{aligned} \tag{10.6}$$

and

$$\begin{aligned} b'_1 &= b_1 \\ b'_j &= b_j - \frac{l_{2j-2}}{l'_{2j-3}} b_{j-1} \qquad 2 \leq j \leq n \end{aligned} \tag{10.7}$$

where l'_i and b'_i are the modified components of $\mathbf{l}$ and $\mathbf{b}$ respectively.

The recursive definitions given here are very suitable for immediate translation into a functional program which is presented in the next section.

The back-substitution algorithm is given by

$$\begin{aligned} a_n &= \frac{b'_n}{l'_{2n-1}} \\ a_j &= \frac{b_j - l'_{2j} a_{j+1}}{l'_{2j-1}} \qquad 1 \leq j \leq n-1 \end{aligned} \tag{10.8}$$

where $a_j, j = 1 \ldots n$, are the elements of the solution vector

10.1.2 The SASL (FSC) Program

It is instructive to consider the top-down design approach to programming the solution to the given problem. First it is noted that the final answer is the vector of nodal quantities. Hence a function must be defined which takes as arguments the stiffness matrix (the vector $\mathbf{l}$ in the previous section) and the force vector $\mathbf{b}$ and returns the solution vector as its result. The vectors $\mathbf{l}$ and $\mathbf{b}$ must be assembled from the terms in equations (10.3) and (10.4). Therefore functions must be defined which take the subinterval boundary points x_j and use numerical integration to return as a result the vectors $\mathbf{l}$ and $\mathbf{b}$. These functions in turn require values of o_j and do_j/dx; so finally functions for these shape functions are provided. This kind of breakdown forces the programmer into a modular decomposition of the problem. Each of the required functions will now be presented in the sequence in which they have to be written (not necessarily designed) to satisfy the scope rules of SASL. This order is essentially the reverse of the design order outlined above. **Note:** FSC does not require this ordering and we could write the FSC function definitions in *any* order. The functions o_j and do_j/dx from (10.2) are defined with the help of the following function which returns the number at position i in the vector x:

SASL:

```
Let POSN be lambda x.lambda i.
                i = 1 -> hd x;
                         POSN (tl x) (i-1)
```

FSC:

We may use the primitive array indexing for this purpose. If we wanted to implement the above function we could write

```
> POSN :: INT -> [a] -> a
> POSN i x = x[i]
```

In the following definitions, **xj** (**X** in FSC) is the vector of $\mathbf{x}_j$'s and **i** denotes which of the two element shape functions is being computed.

SASL:

```
let PHI be lambda x. lambda j. lambda i. lambda xj.
  i = 1 -> (POSN xj (j+1)-x        ) DIV (POSN xj (j+1)-POSN xj j);
           (x             -POSN xj j) DIV (POSN xj (j+1)-POSN xj j)

let DPHIDX be lambda x.lambda j. lambda i. lambda xj
  i = 1 -> (-1) DIV (POSN xj (j+1) - POSN xj j);
           (1) DIV (POSN xj (j+1) - POSN xj j)
```

FSC:

```
> PHI :: DOUBLE -> INT -> INT -> [DOUBLE] -> DOUBLE
> PHI x j 1 X = (X[j+1] - x) / (X[j+1] - X[j])
> PHI x j _ X = (x - X[j])   / (X[j+1] - X[j])

> DPHIDX :: DOUBLE -> INT -> INT -> [DOUBLE] -> DOUBLE
> DPHIDX x j 1 X = ~1/(X[j+1] - X[j])
> DPHIDX x j _ X =  1/(X[j+1] - X[j])
```

The integrands in (10.3) and (10.4) can now be evaluated using PHI and DPHIDX:

SASL:

```
let F be lambda x. 2 DIV (x*x)

in

let FINT be lambda x. lambda j. lambda i. lambda k. lambda xj.

        k = 3 -> (F x) * (PHI x j i xj);
                  (DPHIDX x j i xj) * (DPHIDX x j k ij) * x
```

FSC:

```
> FINT :: DOUBLE -> INT -> INT -> INT -> [DOUBLE] -> DOUBLE
> FINT x j i k X | k == 3     = 2/x^2 * PHI{x,j,i,X}
>               | otherwise = DPHIDX{x,j,i,X} * DPHIDX{x,j,k,X} * x
```

Here $k = 1$ or 2 determines whether $\phi_j^{(1)}$ or $\phi_j^{(2)}$ is calculated, whereas $k = 3$ indicates calculations of the integrand for the force vector.

Numerical integration is carried out using Gaussian quadrature. The arguments for such a function are the specifications for the integrand given earlier as well as the lists of the absicissae **xg** and the weights **wg** for the quadrature and the end points of the integration interval a_1 and b_1.

SASL:

```
let GQUAD be lambda b1. lambda a1. lambda j. lambda i. lambda k.
             lambda xj. lambda xg. lambda wg.

        xg = / -> 0;
        ((b1 - a1) DIV 2) * (hd wg)
         * FINT ((hd xg) * ((b1 - a1) DIV 2)
         + ((b1 + a1) DIV 2)) j i j xj
         + GQUAD b1 a1 j i k xj (tl xg) (tl wg)
```

FSC:

```
> GQUAD :: DOUBLE->DOUBLE->INT->INT->INT->[DOUBLE]->[DOUBLE]->[DOUBLE]->DOUBLE
> GQUAD b a j i k X xs ws
>      = sum[pr*w*FINT{x*pr+pm,j,i,k,X}| w in ws dot x in xs]
>                                 where pm = (b+a)/2
>                                       pr = (b-a)/2
```

The assembly of the above integrals into the vector representing the stiffness matrix follows according to (10.3):

SASL:

```
let L be lambda j. lambda xj. lambda xg. lambda wg. lambda n.

j= (n-1) -> GQUAD (POSN xj (j+1))
                  (POSN xj j) (n-1) 2 2 xj xg wg:/;
            GQUAD (POSN xj (j+1))
                  (POSN xj j) j 2 2 xj xg wg+
                  GQUAD (POSN xj (j+2))
                  (POSN xj (j+1)) (j+1) 1 1 xj xg wg):
                  GQUAD (POSN xj (j+2))
                  (POSN xj (j+1)) (j+1) 1 2 xj xg wg):
                  L (j+1) xj xg wg n
```

FSC:

```
> mkL :: [DOUBLE]->[DOUBLE]->[DOUBLE]->INT->[DOUBLE]
> mkL X xs ws N
>     = [   GQUAD X[j+1] X[j]   j     2 2 X xs ws +
>           GQUAD X[j+2] X[j+1] (j+1) 1 1 X xs ws ;
>           GQUAD X[j+2] X[j+1] (j+1) 1 2 X xs ws |
>       j in [1..N-2]] +> GQUAD X[N] X[N-1] (N-1) 2 2 X xs ws
```

An analogous function assembles the force vector according to (10.4) and noting (10.5). bc_1 is the essential boundary condition $U(a)$ while bc_2 is the natural boundary condition $(-xdU/dx)_{x=b}$.

SASL:

```
let B be lambda j. lambda xj. lambda xg.
         lambda wg. lambda n. lmdab bc1. lmdab bc2.
j=2 -> - GQUAD (POSN xj 2) (POSN xj 1) 1 2 3 xj xg wg
       - GQUAD (POSN xj 3) (POSN xj 2) 2 1 3 xj xg wg
       - bc1 * GQUAD (POSN xj 2) (POSN xj 1) 1 1 2 xj xg wg
        : B (j+1) xj wg n bc1 bc2;
k=n -> - GQUAD (POSN xj n) (POSN xj (n-1)) (n-1) 2 3 xj xg wg - bc2:/;
       - GQUAD (POSN xj j) (POSN xj (j-1)) (j-1) 2 3 xj xg wg
       - GQUAD (POSN xj (j+1)) (POSN xj j) j 1 3 xj xg wg
       : B (j+1) xj xg wg n bc1 bc2
```

FSC:

```
> mkB :: [DOUBLE]->[DOUBLE]->[DOUBLE]->INT->DOUBLE->DOUBLE->[DOUBLE]
> mkB X xs ws N bc1 bc2
>  = ~GQUAD X[2] X[1] 1 2 3 X xs ws
>    -GQUAD X[3] X[2] 2 1 3 X xs ws
>    -bc1 * GQUAD X[2] X[1] 1 1 2 X xs ws
>       <+ [ ~GQUAD X[j] X[j-1] (j-1) 2 3 X xs ws
>            - GQUAD X[j-1] X[j] j 1 3 X xs ws | j in [3..N-1] ] +>
>    ~GQUAD X[N] X[N-1] (N-1) 2 3 X xs ws - bc2
```

The system equations have now been developed and can be used later on in the solver which is now defined. First, SASL functions are written to find the i^{th} component of the vector $\mathbf{l}$ according to (10.6).

SASL:

```
let LP be lambda L. lambda i.

i = 1 -> POSN L 1;
i MOD 2 = 0 -> POSN L i;
POSN L i - (POSN L (i-1) * POSN L (i-1)) DIV LP L (i-2)
```

FSC:

```
> LP :: [DOUBLE]->[DOUBLE]
> LP (1 <:L:> N)
>  = let array L'[1..N] ordered [1..N] where
>              L'[1] = L[1]
>              L'[i] | (i is even) = L[i]
>                    | otherwise = L[i] - L[i-1]^2/L'[i-2]
>    in L'
```

Similarly, from (10.7) the modified i^{th} component of $\mathbf{b}$ is

SASL:

```
let BP be lambda L. lambda b. lambda i.

i=1 -> POSN b 1;

POSN b i - (POSN L (2*i-2) * POSN b (i-1)) DIV LP L (2 *i - 3)

let BSUB be lambda L. lambda b. lambda n. lambda i.

i = 1->(BP L b i) DIV (LP L (2*n-1)):/;
       (BP L b i - LP L (2*i) * hd (BSUB L b n (i+1)) DIV (LP L (2*n-1))
      : BSUB L b n (i+1)
```

FSC:

```
> BP :: [DOUBLE] -> [DOUBLE] -> [DOUBLE]
> BP L' (1 <: B :> N)
>  = let array B'[1..N] ordered [1..N] where
>             B'[1] = B[1]
>             B'[j] = B[j]-L'[2j-2]/L'[2j-3]*B'[j-1]
>    in B'
```

These functions are then used in the function for back-substitution which follows from (10.8).

SASL:

```
let BSUB be lambda L. lambda b. lambda n. lambda i.

i = 1->(BP L b i) DIV (LP L (2*n-1)):/;
       (BP L b i - LP L (2*i) * hd (BSUB L b n (i+1)) DIV (LP L (2*n-1))
      : BSUB L b n (i+1)
```

FSC:

```
> BSUB :: [DOUBLE] -> [DOUBLE]->[DOUBLE]
> BSUB L (1 <: B :> N)
>  = let array a[1..N] ordered [N..1] where
>              a[j] | (j == N)  = B[N]/L[2N-1]
>                   | otherwise = (B[j] - L[2j]*a[j+1])/L[2j-1]
>    in a
```

The final solution vector `a` can then be obtained by applying `BSUB` to the stiffness matrix and force vector defined above:

SASL:

```
let A be lambda n. lambda xj. lambda xg.
        lambda wg. lambda bc1. lambda  bc2.
BSUB (L 1 xj xg wg n) (B 2 xj xg wg n bv1 bv2) (n-1) 1
```

FSC:

```
> A :: INT->[DOUBLE]->[DOUBLE]->[DOUBLE]->DOUBLE->DOUBLE->[DOUBLE]
> A N X x_g w_g BC1 BC2
>  = BSUB L' B' where
>         L' = LP L
>         B' = BP L' B
>         L  = mkL X x_g w_g N
>         B  = mkB X x_g w_g N BC1 BC2
```

10.1.3 Conclusions

In the original paper Dwyer concludes that the functions presented above demonstrate the power and conciseness of functional languages. Each function is short and therefore can easily be tested in an independent manner. The design of the solution forces the programmer to think in a highly-modular fashion and thus adhere strictly to top-down design principles. Dwyer also mentions that there are, however, some major disadvantages associated with programming in a functional manner. The efficiency of execution of most functional programs is low. We believe FSC improves the readability of this particular example and, as we show in the sections that follow, FSC addresses these efficiency concerns also.

10.2 Summed axpys

In this section we concentrate on the raw power and flexibility of FSC by considering the computation

$$axpy = a\mathbf{x} + \mathbf{y}$$

In order that we may compare this with Haskell the computation we actually perform is

$$\sum_{i=1}^{N}(ax_i + y_i)$$

We use this computation since its output is small and hence I/O does not taint the results too badly.

10.2.1 The programs

The programs to carry out the above computation are as follows:

10.2.1.1 C summed axpy

```
 double summed_daxpy(int N, double a, double *x, double *y) {
        int i;
        double acc=0.0;
        for(i=N;i--;)
                acc += (a * x[i]+ y[i]);
        return acc;
}
main() {
        int N,i;
        time_t timer=1;
        double a;
        double *x;
        double *y;

        scanf("%d\n",&N);  /* read N */
        scanf("%f\n",&a);  /* read a */

        x = (double*)malloc((unsigned) (N * sizeof(double)));
        for (i=0;i<N;i++) {
                scanf("%lf\n",&x[i]); /* read in elements of X */
        }

        x = (double*)malloc((unsigned long) (N * sizeof(double)));
        for (i=0;i<N;i++) {
                scanf("%lf\n",&y[i]); /* read in elements of Y */
        }

        timer = clock();              /* start the clock */
        a = summed_daxpy(N,a,x,y);    /* perform computation */
        timer = clock() - timer;      /* evaluate interval */

        printf(" result = %f\n",a);   /* print result for validation */
        printf(" time elapsed = %ld\n",timer); /* print interval */

        exit(0);
}
```

10.2.1.2 Haskell summed axpy

```
>         main :: Dialogue
>         main  = interact (show.process.lines)
```

```
>       process :: [String]->Double
>       process [astr,xstr,ystr] = summed_axpy a x y
>          where (a,x,y)
>           = (read astr,
>              read xstr,
>              read ystr)::(Double,[Double],[Double])

>       summed_axoy a xs ys :: Double -> [Double] -> [Double] -> Double
>       summed_axpy a xs ys = sum [ a * x + y | (x,y) <- zip xs ys]
```

10.2.1.3 FSC summed axpy

```
>       main :: DOUBLE -> [DOUBLE]^2  ->  IO DOUBLE

>       main a X Y  = do IO return summed_axpy a X Y

>       summed_axpy :: DOUBLE -> [DOUBLE] -> [DOUBLE] -> DOUBLE
>       summed_axpy a X Y = sum[a*x+y|x in X dot y in Y]
```

The codes were then compiled and executed to evaluate

$$\sum_{i=1}^{N}(2.5 \times 0.1 + 1.1) = 1.35N$$

for various values of N.

10.2.2 Executable Size

Code	C	Haskell	FSC
Size of stripped executable(bytes)	3888	717352	64628

Table 10.1: C / Haskell / FSC executable sizes.

Table 10.1 shows that the prototype FSC compiler generated an executable larger than the C version but much smaller than the Haskell version.

10.2.3 Execution Efficiency

The programs were timed using various values for N. In the following discussion we use the convention X(math) to denote the program in language X disregarding the non-mathematical work such as I/O.

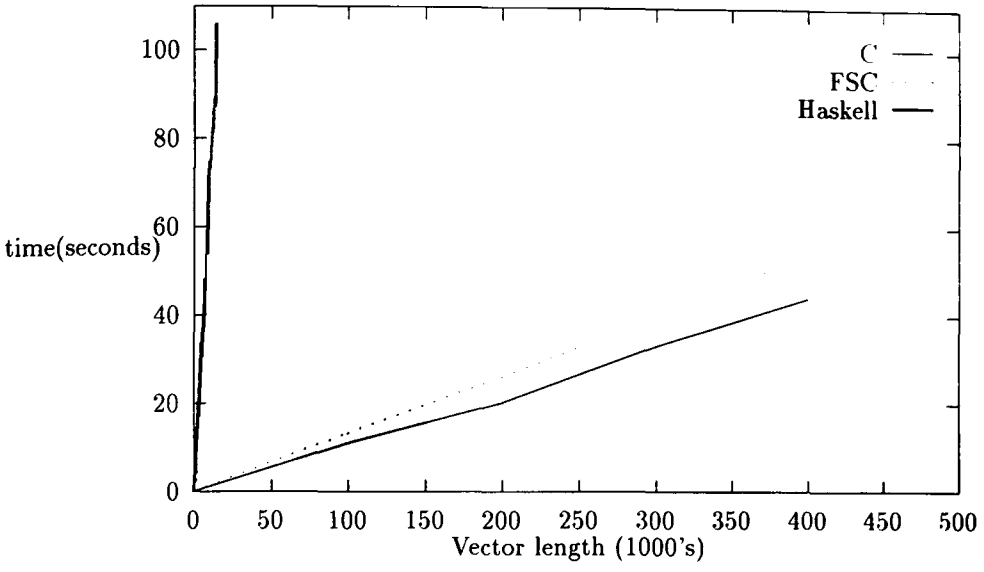


Figure 10.2: Timings (unix) for summed axpys with I/O

10.2.4 Results

Fig. 10.2 shows the results of running the above code on various values of N timing the execution (including I/O). Fig. 10.3 shows the results of running the C and FSC code on various values of N timing only the computational kernel. These values show that for this example the computational kernel of FSC runs at a speed comparable with C and FSC easily outstrips the corresponding Haskell program.

10.3 Argument Domains

We continue to take as our example the axpy computation and examine the generality of this computation. In FSC, we consider the function

```
>       axpy a X Y = [a*x + y| x in X dot y in Y]
```

as being general, as we may apply it to vectors and scalars of many different component types. that is, all the following calls

```
>       floats :: REAL
>       floats  = axpy 2.5 [0.1] [1.1]
>       doubles :: DOUBLE
>       doubles = axpy 2.5d [0.1d] [1.1d]
>       ints :: INT
```

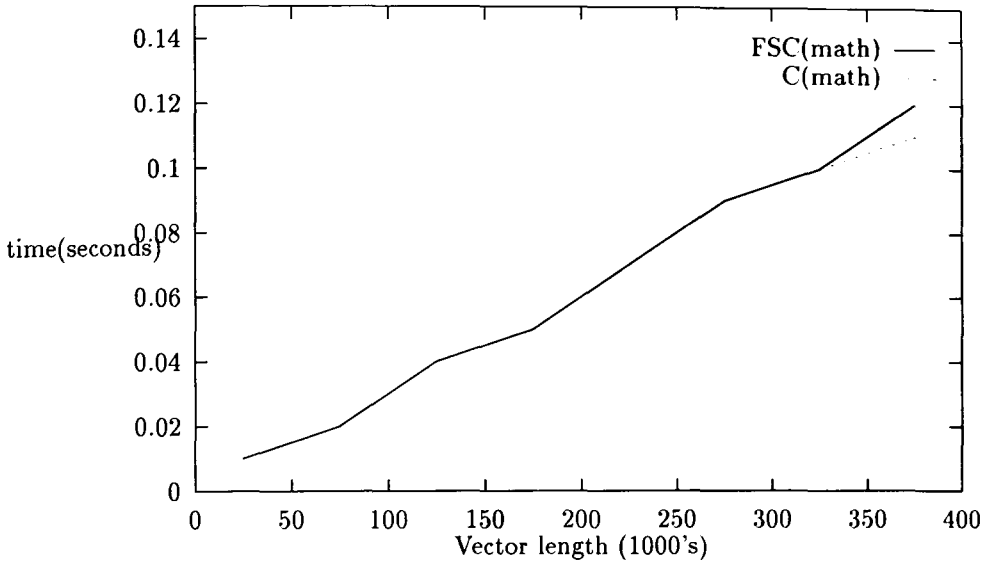


Figure 10.3: Timings (in-code) for summed axpys without I/O

```
>       ints    = axpy 2 [0] [1]
```

are valid. Languages such as C++ support templates which allow the same behaviour although explicit templates can get confusing. In addition to the above examples we may also calculate

```
>       real_int_real :: REAL
>       real_int_real =  axpy 2.5 [1] [1.0]
```

That is, our type system allows much more freedom than is available in template based systems and allows functions whose return types are non-trivial functions of their input types.

10.4 Dot Products

We now consider the computation

$$dot(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$$

which may be written tersely in a functional language (such as FSC) as

```
dotprod X Y = sum(zipWith (*) X Y)
```

where `zipWith` is defined as

```
>       zipWith f X Y = [ f x y | x in X dot y in Y]
```

Because we are able to reason at a higher level about this function, the FSC compiler is able to automatically inline the call to `zipWith`, reducing the number of array traversals. If we time this against the comparable C code retaining the modularity.

```
double* timesVec(int N,double *x,double *y)
{
        int i;
        for (i=0;i<N;i++)
                x[i] *= y[i];
        return x;
}

double sum(int N,double *x)
{
        int i;
        double acc=0.0;
        for (i=0;i<N;i++)
                acc += x[i];
        return acc;
}

double dot(int N,double *x,double *y)
{
        return sum(N,timesVec(N,x,y));
}
```

we find that this modularity causes inefficiency. A dot-product of two vectors of order 10^5 takes 1.21 seconds in C (gcc -O4) as opposed to 0.71 seconds in FSC[1]. Written more conventionally the C code acheives the same efficiency as FSC as in Section 10.2.

10.5 Gaussian Quadrature

We now consider an FSC version of the [Haskell] Gaussian quadrature function described in Chapter 4:

```
>       -- Haskell Gaussian Quadrature --
>       gq x w f (a,b) = pr * sum(zipWith f1 w x)
>        where f1 wi xi = wi * (f (pm+pr*xi) + f (pm-pr*xi))
>              pm = (b+a)/2.0
>              pr = (b-a)/2.0
```

The FSC version is almost identical to the Haskell version, differing only in that double-precision floating point numbers in FSC have `d` after them.

[1] To regain this performance from C the functions must be declared `static inline`.

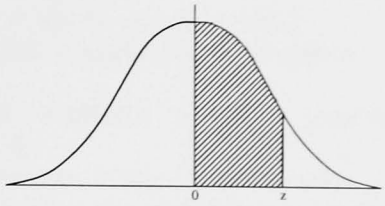


Figure 10.4: Area under a normal curve

```
>       -- FSC Gaussian Quadrature --
>       gq x w f a b = pr * sum(zipWith f1 w x)
>        where f1 wi xi = wi * (f (pm+pr*xi) + f (pm-pr*xi))
>              pm = (b+a)/2.0d
>              pr = (b-a)/2.0d
```

It is this function which we use in the next section to tabulate the values of the cumulative density function (CDF) of a normal distribution.

10.6 Normal CDFs : Extended Example

This section is an extended example detailing the process of compiling the code to compute the tabulated values for the cumulative density function (CDF) of the normal distribution with mean μ and standard deviation σ.

10.6.1 The Problem

The problem we choose to solve is to tabulate the area under the curve of a normal probability distribution function (PDF) from zero to various values of x (Fig. 10.4). We choose this example as the table should be familiar to the reader and, as the normal CDF does not admit a closed form, it is a motivating example. In an implementation we insist that the integrations be performed by a general integrating function, that the values of μ and σ be runtime dependent and that each integration be evaluated independently to simplify the definition.

10.6.2 The Mathematical Formulation

The values we wish to find may be defined as

$$\frac{1}{\sigma\sqrt{2\pi}} \int_0^x e^{-\frac{(y-\mu)^2}{2}} dy, \qquad x = 0, \delta, 2\delta, \ldots, 4 - \delta$$

The FSC code to compute this table is as follows usinf abscissae and weights taken from [1].

```
>       main :: DOUBLE^2 -> INT  -> IO (Array DOUBLE)
```

```
>       main  mu sigma N   = do IO return RESULT
>             where RESULT = normalCDF mu sigma N

>       normalCDF :: DOUBLE -> DOUBLE -> INT -> [DOUBLE]
>       normalCDF mu sigma N
>        = [ gq x w distribution 0.0d0 (4.0d0*i)/N | i in [0..N-1]]
>            where distribution x = normalPDF mu sigma x
>                  x = [0.2386191860831970d0,     -- abscissae
>                       0.661209386466265d0,
>                       0.932469514203152d0]
>                  w  = [0.4679139345726910d0,     -- weights
>                        0.3607615730481390d0,
>                        0.1713244923791700d0]

>       normalPDF :: DOUBLE -> DOUBLE -> DOUBLE -> DOUBLE
>       normalPDF mu sigma x
>        = exp((x-mu)^2/^2.0d0)/(sigma*sqrt(2.0d0*PI))
>              where PI = 3.14159265358979323846d0
```

This program was benchmarked against a C program written by the author and a C++ program kindly written by Dr. Misra [68]. These codes were hand-optimised for speed (for instance the C version does not compute $\sqrt{2\pi}$ but has it defined as 2.50663 in the code). These were run on tables ranging in size from 0 to 50,000 entries. The results of this benchmark are shown in Fig. 10.5.

10.6.3 Discussion

On this example, the FSC version performs remarkably well, running at just under double the speed of the C and C++ versions and the C version sneaks in under the C++ version as one would expect. The efficiency gain over the imperative versions is due to the ability to manipulate and optimise the code more freely in the absence of side effects. A C compiler will not try to inline a function when it is passed as a pointer rather than when it is simply called. To FSC functions are simply values like any other and, as such, may be freely in-lined and optimised further after the in-lining. The C code can be found in Appendix B.

10.7 Cyclic Reduction: Extended Example

Cyclic reduction [36] (or odd-even reduction) is a method of solving a tridiagonal system of equations which may be performed in parallel[2]. The variation of cyclic reduction we choose to implement

[2] Cyclic reductions were mentioned in Chapter 4 where a quadtree version was presented.

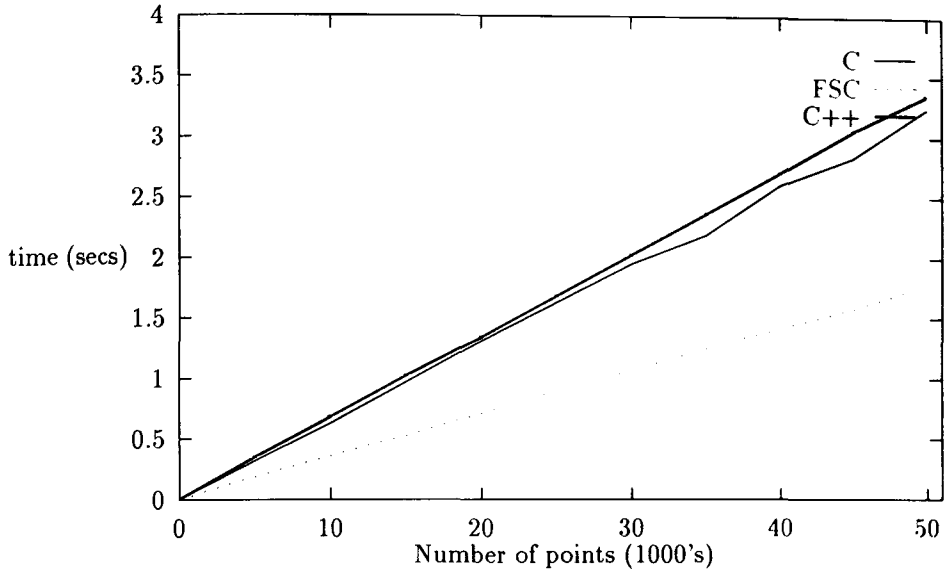


Figure 10.5: Timings (in-code) for CDF generation

proceeds as follows:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ \vdots \\ \vdots \\ y_n \end{bmatrix}$$

A tridiagonal system of equations $Ax = y$ is transformed into an equivalent pentadiagonal system with zero sub- and super-diagonals.

$$\begin{bmatrix} b_1^{(1)} & 0 & c_1^{(1)} & & \\ 0 & b_2^{(1)} & \ddots & \ddots & \\ a_3^{(1)} & \ddots & \ddots & \ddots & c_{(n-2)}^{(1)} \\ & \ddots & \ddots & \ddots & 0 \\ & & a_n^{(1)} & 0 & b_n^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1^{(1)} \\ \vdots \\ \vdots \\ \vdots \\ y_n^{(1)} \end{bmatrix}$$

This system is then reordered so that it is expressed as two sub-problems of half the dimension. That is, if we assume that n is even we renumber the equations in the order $1, 3, \ldots, n-1, 2, 4, \ldots, n$

and renumber the unknowns in a similar way.

$$\left[\begin{array}{cc|ccc} b_1^{(1)} & c_1^{(1)} & & & \\ a_3^{(1)} & \ddots & \ddots & & \\ & \ddots & b_{n-1}^{(1)} & & \\ \hline & & & b_2^{(1)} & \ddots \\ & & & \ddots & \ddots & c_{n-2}^{(1)} \\ & & & & a_n^{(1)} & b_n^{(1)} \end{array}\right] \left[\begin{array}{c} x_1 \\ \vdots \\ x_{n-1} \\ \hline x_2 \\ \vdots \\ x_n \end{array}\right] = \left[\begin{array}{c} y_1^{(1)} \\ \vdots \\ y_{n-1}^{(1)} \\ \hline y_2^{(1)} \\ \vdots \\ y_n^{(1)} \end{array}\right]$$

Since we now have two tridiagonal systems we may apply recursively and in parallel the above transformation process to the two non-zero quadrants of the matrix until we reach a set of trivial equations. The transformation step to create the pentadiagonal system described above is given below

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1} \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \beta_i a_{i+1} \\ c_i^{(1)} &= \beta_i c_{i+1} \\ y_i^{(1)} &= y_i + \alpha_i y_{i-1} + \beta_i y_{i+1} \\ \alpha_i &= -a_i / b_{i-1} \\ \beta_i &= -c_i / b_{i+1} \end{aligned}$$

though the more formal version is:

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1} && && i \in \{3, \ldots, n\} \\ b_i^{(1)} &= \begin{cases} b_1 + \beta_1 a_2 & \text{if } i = 1 \\ b_i + \alpha_i c_{i-1} + \beta_i a_{i+1} & \text{if } i < i < n \\ b_n + \alpha_n c_{n-1} & \text{if } i = n \end{cases} && i \in \{1, \ldots, n\} \\ c_i^{(1)} &= \beta_i c_{i+1} && && i \in \{1, \ldots, n-2\} \\ y_i^{(1)} &= \begin{cases} y_1 + \beta_1 y_2 & \text{if } i = 1 \\ y_i + \alpha_i y_{i-1} + \beta_i y_{i+1} & \text{if } i < i < n \\ y_n + \alpha_n y_{n-1} & \text{if } i = n \end{cases} && i \in \{1, \ldots, n\} \\ \alpha_i &= -a_i / b_{i-1} && && i \in \{2, \ldots, n\} \\ \beta_i &= -c_i / b_{i+1} && && i \in \{1, \ldots, n-1\} \end{aligned}$$

We may implement this algorithm in FSC as

```
> ...
> a'    = [alpha[i] * a[i-1]                                  | i in [3..N]   ]
> b'    = [b[1] + beta[1] * a[2]]
>       ++[b[i] + alpha[i]*c[i-1] + beta[i]*a[i+1] | i in [2..N-1] ]
>       ++[b[N] + alpha[N]*c[N-1]]
> c'    = [beta[i] * c[i+1]                                   | i in [1..N-2] ]
> y'    = [y[1] + beta[1] * y[2]]
>       ++[y[i] + alpha[i]*y[i-1] + beta[i]*y[i+1] | i in [2..N-1] ]
```

```
>       ++[y[N] + alpha[N]*y[N-1]]
> alpha = [~a[i]/b[i-1]                                  | i in [2..N]   ]
> beta  = [~c[i]/b[i+1]                                  | i in [1..N-1] ]
```

and we immediately see that there is a direct correlation. Interestingly, we could also implement an FSC version of the original presentation as

```
> ...
> array a'[i]    = alpha[i] * a[i-1]
> array b'[i]    = b[i] + alpha[i]*c[i-1] + beta[i]*a[i+1]
> array c'[i]    = beta[i] * c[i+1]
> array y'[i]    = y[i] + alpha[i]*y[i-1] + beta[i]*y[i+1]
> array alpha[i] = ~a[i]/b[i-1]
> array beta[i]  = ~c[i]/b[i+1]
```

We choose to use the former presentation rather than the latter as this it more efficient (the bounds of the array are explicitly given). All the implementations (including C++) relate to systems of 2^N equations where N is an integer. Our first code for a full cyclic reduction is:

```
> cyclic_reduction a  (b :> N) c     y
>  = if (N==2)
>    then Branch Leaf{y[1]/b[1]} Leaf{y[2]/b[2]}
>    else
>      Branch cyclic_reduction{2<:a1,b1,c1,y1}
>             cyclic_reduction{2<:a2,b2,c2,y2}
>        where
>           (a1,a2) = odd_even a'
>           (b1,b2) = odd_even b'
>           (c1,c2) = odd_even c'
>           (y1,y2) = odd_even y'

>           a1    = [alpha[i] * c[i-1]                          | i in [3..N] ]

>           b1    = [b[1] + beta[1] * a[2]]
>                 ++[b[i] + alpha[i]*c[i-1] + beta[i]*a[i+1] | i in [2..N-1] ]
>                 ++[b[N] + alpha[N]*c[N-1]]

>           c'    = [beta[i] * c[i+1]                           | i in [1..N-2] ]

>           y'    = [y[1] + beta[1] * y[2]]
>                 ++[y[i] + alpha[i]*y[i-1] + beta[i]*y[i+1] | i in [2..N-1] ]
>                 ++[y[N] + alpha[N]*y[N-1]]

>           alpha = [~a[i]/b[i-1]                               | i in [2..N]   ]

>           beta  = [~c[i]/b[i+1]                               | i in [1..N-1] ]
```

where the result is stored in a tree, to avoid repeated array copying. Once the tree is complete a result array may be generated so that copying is avoided as a post processing phase:

```
> FFT_tree_to_array :: [Double] -> INT -> INT -> Tree DOUBLE
> FFT_tree_to_array A n N (Leaf a)
>  = (A[m -> a],n+1)
>       where m = bitflip(n,N)
> FFT_tree_to_array A n N (Branch a b)
>  = FFT_tree_to_array A' n' N b
>       where (A',n') = FFT_tree_to_array A n N a
```

where `bitflip` is a function from the standard prelude to perform an FFT permutation:

```
> bitflip :: INT -> INT -> INT
> bitflip n N = (x | x <- bitXOR{x<<1,bitAND i 1};
>                          i <- i>>1;
>                          k <- k>>1 | (x,i,k) = (0,N,n))
```

Our second code reorders the results on the fly using the prelude function `riffle`

```
> riffle X Y = [ x;y | x in X dot y in Y]
```

which interleaves two arrays, to give

```
> cyclic_reduction a  (b :> N) c     y
>  = if (N==2)
>    then [y[1]/b[1],y[2]/b[2]]
>    else
>      riffle cyclic_reduction{2<:a1,b1,c1,y1}
>             cyclic_reduction{2<:a2,b2,c2,y2}
>        where
>           (a1,a2) = odd_even a'
>           (b1,b2) = odd_even b'
>           (c1,c2) = odd_even c'
>           (y1,y2) = odd_even y'

>           a1    = [alpha[i] * c[i-1]                          | i in [3..N] ]

>           b1    = [b[1] + beta[1] * a[2]]
>                 ++[b[i] + alpha[i]*c[i-1] + beta[i]*a[i+1] | i in [2..N-1] ]
>                 ++[b[N] + alpha[N]*c[N-1]]

>           c'    = [beta[i] * c[i+1]                           | i in [1..N-2] ]

>           y'    = [y[1] + beta[1] * y[2]]
>                 ++[y[i] + alpha[i]*y[i-1] + beta[i]*y[i+1] | i in [2..N-1] ]
```

```
>              ++[y[N] + alpha[N]*y[N-1]]

>          alpha = [~a[i]/b[i-1]                          | i in [2..N]   ]

>          beta  = [~c[i]/b[i+1]                          | i in [1..N-1] ]
```

10.7.0.1 Results

These codes were executed on systems of equations with up to 2^{14} elements and bench-marked against an existing C++ cyclic reduction code which had previously been run against Haskell and achieved a speed of at least 20 times that of Haskell [65]. The results of running this code can be seen in Fig. 10.6. These figures indicate that the C++ code runs at 9% and 27% faster than the array and tree versions respectively. Some improvement could be made to the efficiency of the FSC versions by sharing loops but for reasons of clarity we chose not to.

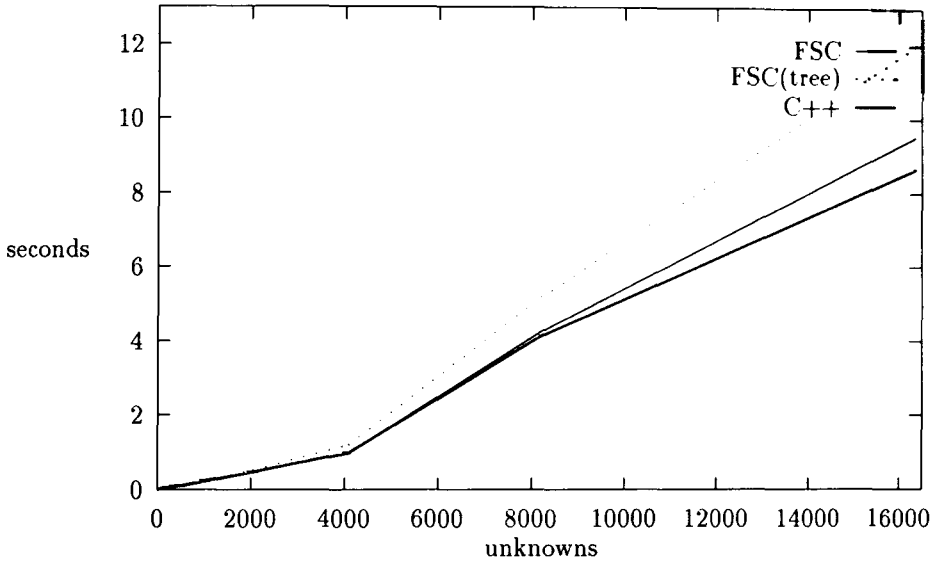


Figure 10.6: Timings (in-code) for cyclic reductions

10.7.0.2 Code Size

The amount of code which was required to write and implement these cyclic reductions ranged from 290 lines in C++ to 34 lines in FSC (the above function and an entry point). Although the C++ version is 9% faster than the FSC version, FSC is over 800% terser on this example, a fact to bear in mind when considering raw efficiency versus programming cost, and if array declarations were used this line count could fall below 20 without being obfuscated.

10.8 Summary

In this chapter we have presented various algorithms written in FSC. We recoded algorithms to demonstrate the tersity of the language and presented two case studies to demonstrate that FSC was capable of contending with C/C++ and in some cases beating it on efficiency! Since larger programs are constructed from the building blocks that we have shown run efficiently we believe that this vindicates our *proof of theory*. Further examples of FSC implementations of numerical exemplars are, however, provided in Appendix G for the interested reader.

Chapter 11

Related Work

In this chapter we survey the previous work covering areas introduced in Part II.

11.1 Use of Multi-Parameter Type Classes

- Cormack & Wright [23] present an overloading type system which treats overloaded functions in a similar manner to FSC. However, their system does not use implicit typing and all overloaded functions have their type parameters explicitly denoted at definition. They do not present a typechecking algorithm for their system.

- The GoFER language [54] has multi-parameter type classes but only provides conservative overloaded resolution, and therefore cannot support many numerical examples. For example, it is necessary (in GoFER) for the user to provide explicit typings on expressions such as

  ```
  ((((3 * 4.5) :: Double) * 4.8) :: Double) * 4 :: Double
  ```

  However, the type classes of GoFER are suitable for coding examples involving monads and monad transformers [3].

- Ophel & Duggan [72] provide a multi-parameter type class system on which many of the ideas in FSC are based. However, their system is restricted by the method they use for implementation (adding extra function parameters), which causes inefficiency and forces some syntactic forms to be constrained. Since FSC uses specialisation as its implementation method these restrictions are lifted.

- The latest version of GHC (3.01) has support for multi-parameter type classes as no doubt will Standard Haskell[51]. However, these are aimed at the same area as those of GoFER and, irrespective of their semantic power, would be too inefficient to use in our problem domain. GHC's inclusion of multi-parameter type classes is long overdue. It will be interesting to see how the Haskell community uses them.

11.2 Use of Transformation Systems

- Boyle, Fitzpatrick, Clint & Harmer [15, 14] use program transformations to translate code expressed as pure LISP and ML to FORTRAN. They make use of a program transformation system called TAMPR which uses transformations in a manner similar to FSC. The manner in which FSC differs with TAMPR is that FSC only attempts to perform source to source transformations and combine these in a less ad-hoc manner.

11.3 Use of Recursively Defined Monolithic Arrays

- Gao, Yates, Dennis & Mullin [37, 38] introduce a monolithic array constructor which offers a solution to the overhead of scheduling and synchronisation of recursively defined arrays and the copying of intermediate arrays during array construction. In their papers they give no implementation details and their array constructor resembles an explicitly ordered array declaration in FSC with poorer syntax. That is, information concerning the evaluation order of the elements is mandatory. Gao *et al.* also offer subscript analysis techniques for checking the validity of a specified ordering which (as mentioned in Chapter 7) may also be used to check the validity of the optional FSC array declaration orderings.

- Anderson & Hudak [2] present techniques for compiling Haskell array comprehensions based on the definition of *strict contexts*. Their techniques offer a number-theoretic subscript-analysis of Haskell arrays to allow safe-eager evaluation. FSC takes the alternative view that the user be allowed to specify an ordering of array evaluation if the default dynamic schedule is not efficient enough. However (as demonstrated in Chapter 7) FSC's default scheduling generates tags of finite size independent of the array expression rather than (possibly nested, large) closures, hence efficiency-loss is not such an issue.

Part III

Conclusions

Chapter 12

Conclusions

Numerical methods often have very elegant definitions which are easily understood. However it is sadly the case that often an efficient implementation of these methods is rather obscure. These implementations are difficult to verify formally and are also difficult to correct, modify or reason about.

12.0.1 Summary of Thesis

In this thesis we present a study of the principles underlying the design and implementation of a functional language specialised to numerical programming. We justify the design decisions we have taken via the use of empirical investigation of numerical functional programs and provide *proof of concept* of our ideas via results from experiments using an experimental-prototype implementation.

The techniques used in this thesis to investigate existing functional language usage are not new, although their application to this area is. We follow the same technique which led to the RISC revolution in microprocessor design, namely perform a quantitative analysis of commonly used features and provide a system which offers low-cost features.

We integrate the feel of Haskell into FSC such that numerical algorithms are specified in a manner which encourages formal verification as the implementation is often extremely close to the textbook definition.

Since the semantics of kernel FSC are simple[1] the ability to perform optimisation is great. We further enhance this innate ability by defining a transformational metalanguage which allows domain specific transformations to be defined, combined and reasoned about.

This combining is important as transformations may be verified in isolation prior to combination.

The prototype compiler is only the initial step in vindicating the work presented in this thesis. The compiler is not perfect (far from it), however we believe the basic ideas on which it is based are sound and that it can be considered as a *proof of concept*. That is, we may consider the results presented in this thesis as proof that purely-functional high performance computing in a Haskellesque style is indeed feasible.

[1] Compared with other languages.

12.1 Assessment

The importance of the work reported here is:

- It lessens the weight of evidence supporting the claim that functional languages are toys which cannot generate real-world performance.
- It supports the claim that functional languages may be implemented to run at comparable speeds to C/Fortran.
- It highlights the vast difference in performance of a system driven by pragmatic, rather than aesthetic, considerations.
- It presents a study of a functional language used in practice to program numerical methods and discusses the suitability of Haskell as a vehicle for expressing numerical programs. Empirical data regarding the level to which Haskellesque language features are used in numerical examples is also presented and as such, this thesis forms a useful source of reference for future language design.
- It offers and demonstrates ideas pertinent to functional language development such as recursive arrays and multi-parameter type classes and unifies ideas from the SISAL and Haskell languages such as I/O and array/list comprehensions.
- It shows that, for efficiency reasons, numerical functional programming languages must be strict and must operate over datatypes in a catamorphic manner. It also demonstrates that this *working with the grain* -style of algorithm is not always possible and thus arrays are necessary. This investigation has also provided a non-pivoting version of LU-factorisation which does not require all leading submatrices to be non-singular.
- It throws new light on earlier results such as conclusions arising from the SOR implementation of Wainwright & Sexton [100] where a catamorphic appoach was not used.

12.2 Discussion

We may draw an analogy between the designs of Haskell/FSC and the schools of mathematical thought of Intuitionism [6] /Classical logic. We liken FSC compilation to classic logic and proofs such as *reductio ad absurdum* whilst likening Haskell compilation to intuitionistic logic and constructive proof.

This analogy may be best understood by noticing that:

- Intuitionistic proofs are always valid in classical logic, similarly Haskell style compilation is always valid in FSC.
- Although not usable in constructive mathematics, classical proofs are watertight enough for most purposes. Similarly, Haskell-style compilation may be applied to FSC but not the converse. However FSC style compilation *gets you further* and is still formal in the same manner that classical logic and intuitionistic logic are both formal.

- Just as many classical proofs do not admit a constructive equivalent, many easily expressed FSC algorithms do not admit an efficient Haskell equivalent.

In short, FSC is a pragmatic tool with a mathematical impetus designed to perform a well-defined task. However, Haskell is an attempt at embedding a set (albeit small) of idealist principles within a language. As such, FSC out-performs Haskell with ease.

12.3 Further Work

12.3.1 The FSC compiler

The current implementation of the FSC compiler can be described as prototype at best. The results that it produces are good but it is far too inefficient to be used in anger (the cyclic reduction example presented earlier takes over an hour to compile on an Intel Pentium). This is due to the fact that it was not written with efficiency in mind and is implemented in Haskell which is known to be inefficient. Before further work is carried out it is suggested that the implementation be re-engineered in C in the style of GoFER and Hugs [54].

12.3.2 Transformations

The prototype FSC compiler currently does not include the transformational meta-language. As such, very little is known about its use in practice. This would be an interesting area of continued investigation. The transformational system was implemented as an off-line separate thread of investigation with its principles being tested via the implementation in terms of a simpler abstract syntax tree[2]. The behaviour-in-practice of this metalanguage is an area which is ripe for investigation.

12.3.3 Editing

Although not part of the language, the way in which algorithms are presented often allows code to be better understood.

With this in mind an area of further study would be an editor for FSC. This should allow

- literate commenting,
- diagrams, and
- user-defined hierarchical syntactic-sugaring.

12.3.4 User-Defined Hierarchical Sugaring

In mathematics we often invent syntax on the fly and also often give meaning to relative positioning and placement in a manner which, although ambiguous, is well understood within its context. For example

$$A^*, A^T, A^n, W^\perp, f^{(n)}$$

[2]This was not added to the compiler as it was proving unwieldy enough without it.

The most alarming of these being $f^{(n)}$ which depending on the situation may represent either $\underbrace{f \circ \cdots \circ f}_{n \text{ times}}$ or $\frac{d^n f(x)}{dx^n}$. To combat this we introduce the idea of hierarchical sugaring.

Overloading in FSC can be thought to form an abstraction, i.e. the function

```
f^0 = one
f^1 = f
f^n = f * f^(n-1)
```

is of type $\alpha \rightarrow \text{INT} \rightarrow \alpha$ for all α such that $* : \alpha \rightarrow \alpha \rightarrow \alpha$ exists and α has a multiplicative identity. This is opposed to using similar notation to mean very different things.

Hierarchical sugaring is the act of providing views for syntax which

- hides as much or as little information as required, and
- allows non-ASCII symbols to be used in programs.

12.3.5 Example: Quadratic Factorisation

We begin with the FSC function

```
quadFactor a b c = let p = ~b
                       q = sqrt(b^2 - 4a * c)
                   in
                       (p + q,p - q) / 2a
```

A sugaring of this might use the display-rule shown in Fig. 12.1, allowing the function to be written as

$$\text{quadSolve a b c} = (\ b \pm \text{sqrt}(b\hat{}2 - 4a * c))/2a$$

and with other rules written as

$$\text{quadSolve a b c} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

emulating textbook typesetting. User-defined hierarchical sugaring would also allow the visualisation of datatypes as in Figure 12.2.

12.3.6 Matrices

Perhaps a more motivating example is the sugaring of quadtrees as seen in Fig. 12.2. This emulates the shorthand that was used in Chapter 5. So far each of these diagramatic descriptions have been flat, and for simple examples this suffices. However, if we have multiple methods of performing the same operation we need a different solution. If these methods have exactly the same type then type-directed overloading is no use. For example in the following expression

```
area = integrate f (0,1)
```

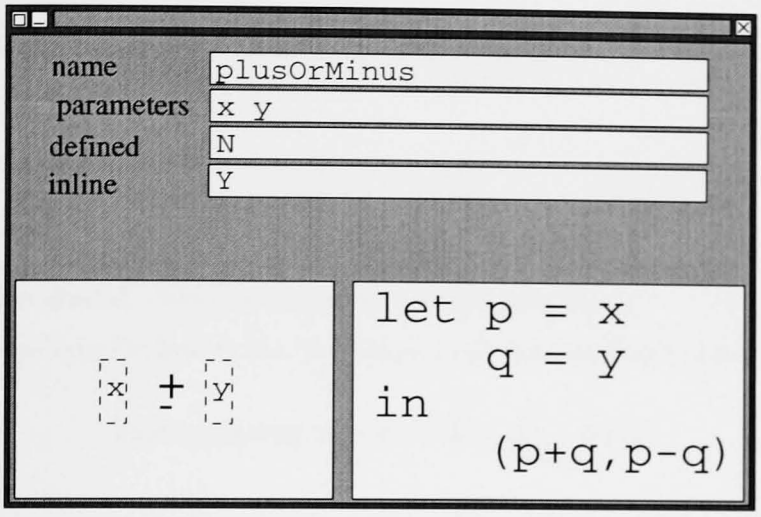


Figure 12.1: ± display rule

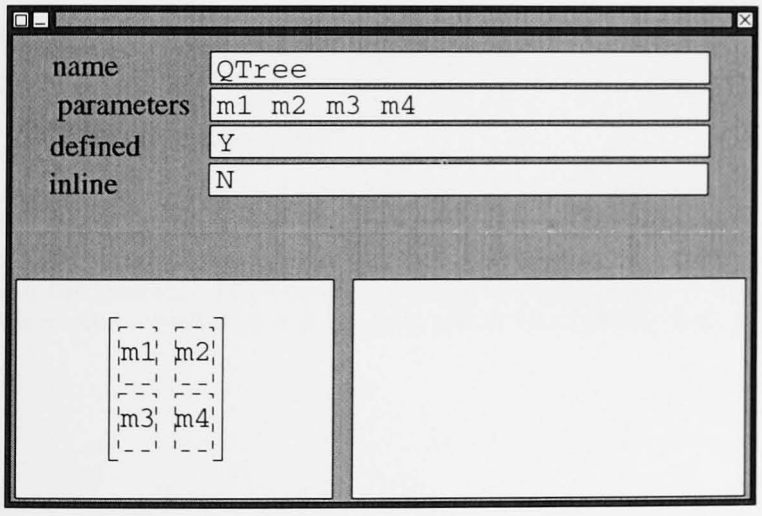


Figure 12.2: Quadtree display rule

it is impossible to overload the functions

```
gaussianQuadrature :: (Double -> Double) -> (Double,Double) -> Double
simpsonsRule       :: (Double -> Double) -> (Double,Double) -> Double
```

onto the same identifier `integrate` as their types are identical and as such, a compiler would not know which one to choose. However, we may create hierarchical representations of them which may be partially viewed.

12.3.7 Hierarchical viewing and information hiding

Using the technique from the last section, it is simple to imagine sugaring Gaussian quadrature and Simpson's rule as

$$\texttt{gaussianQuadrature f (a,b)} \equiv \int_{GQ_a}^{b} f(x)dx$$

and

$$\texttt{simpsonsRule f (a,b)} \equiv \int_{Simpson_a}^{b} f(x)dx$$

respectively. If the two sugarings could be defined in such a way that the subscript information was defined at a lower level, this information could be hidden by the editor until needed and the presentation of resulting algorithms could be simplified to

$$\text{area} = \int_0^1 f(x)\text{dx}$$

12.4 Concluding Remarks

With the current prototype implementation of FSC we have proven that the style of programming advocated by functional programming may indeed be used for the efficient implementation of numerical methods. This fact holds independently of the efficiency of our prototype compiler. The principles on which our prototype implementation stands are independent of FSC and it would be interesting to see how others would interpret the results from Part I of this thesis to form a language.

Appendices

Appendix A

FSC: Syntax and Kernel semantics

A.1 Lexical Structure

A.1.0.1 Lexical Program Structure

program	→	{*lexeme*|*whitespace*}
lexeme	→	*tyid*|*conid*|*varsym*|*consym*|*literal*|*special*|*reservedop*|*reservedid*
literal	→	*integer*|*float*|*double*|*character*|*string*|*boolean*
special	→	(|)|,|;|[|]|_|`|{|}
whitespace	→	*whitestuff*{*whitestuff*}
whitestuff	→	*whitechar*|*comment*
whitechar	→	*space*|*newline*|*tab*
comment	→	--{*any*}*newline*
any	→	*graphic*|*space*|*tab*
graphic	→	*large*|*small*|*digit*
	|	!|"|#|$|%|&|'|(|)|*|+
	|	,|-|.|/|:|;|<|=|>|?|@
	|	[|\\|]|^|_|`|{|}|~
small	→	a|b|...|z
large	→	A|B|...|Z
digit	→	0|1|...|9

Characters not in the category *graphic* or *whitestuff* are not valid and should result in a lexing error.

Program	*prog*	→	*binds*	
Bindings	*binds*	→	$bind_1; \ldots; bind_n$	$n \geq 1$
	bind	→	*var* = *expr*	
Expression	*expr*	→	$expr_1\ expr_2$	Application
		|	λ *var* . *expr*	Lambda abstraction
		|	case *expr* of *alts*	Case expression
		|	if $expr_1$ then $expr_2$ else $expr_3$	Conditional
		|	let *bind* in *expr*	Local definition
		|	letrec *binds* in *expr*	Local recursion
		|	*con*	Constructor
		|	*var*	Variable
		|	*literal*	
Literal values	*literal*	→	*Integer*	
		|	*Float*	
		|	*Character*	
		|	*Double*	
		|	*Boolean*	
Alternatives	*alts*	→	$calt_1; \ldots; calt_n$	$n \geq 1$
Constructor alt	*calt*	→	*con* $var_1 \ldots var_n$ -> *expr*	$n \geq 0$

Figure A.1: Syntax of the FSC kernel

$$\mathcal{P}[\![program]\!] : Val$$
$$\mathcal{P}[\![prog]\!] = \mathsf{EVAL}[\![\ \texttt{letrec}\ prog\ \texttt{in main}\ ]\!]\ \rho_{init}$$

$$\mathsf{EVAL}[\![expr]\!] : Env \to Val$$

$$\begin{array}{lcl}
\mathsf{EVAL}[\![k]\!]\ \rho & = & \mathcal{K}[\![k]\!] \\
\mathsf{EVAL}[\![x]\!]\ \rho & = & \rho\ x \\
\mathsf{EVAL}[\![e_1\ e_2]\!]\ \rho & = & (\mathsf{EVAL}[\![e_1]\!]\ \rho)\ (\mathsf{EVAL}[\![e_2]\!]\ \rho) \\
\mathsf{EVAL}[\![\lambda x.e]\!]\ \rho & = & strict(\lambda x_{new}.\mathsf{EVAL}[\![e]\!]\ (\rho \oplus x \mapsto x_{new})) \\
\mathsf{EVAL}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\ \rho & = & \mathsf{EVAL}[\![\texttt{case}\ e_1\ \texttt{of True -> }e_2\texttt{;} \\
 & & \quad \texttt{False -> }e_3]\!]\ \rho \\
\mathsf{EVAL}[\![\texttt{let}\ x = e\ \texttt{in}\ b\ ]\!]\ \rho & = & \mathsf{EVAL}[\![(\lambda x.b)\ e]\!]\ \rho \\
\mathsf{EVAL}[\![\texttt{letrec}\ binds\ \texttt{in}\ e\ ]\!]\ \rho & = & \mathsf{EVAL}[\![e]\!]\ (\rho \oplus fix(\lambda\rho'.\mathcal{B}[\![binds]\!](\rho \oplus \rho'))) \\
\mathsf{EVAL}[\![c]\!]\ \rho & = & \lambda\epsilon_1 \ldots \lambda\epsilon_n.\langle c, \epsilon_1, \ldots, \epsilon_n\rangle
\end{array}$$

$$\mathsf{EVAL}[\![\texttt{case}\ x\ \texttt{of}\ c_1\ x_{11}\ldots x_{1a_1}\texttt{->}\ e_1\texttt{;}\ldots\texttt{;}\ c_n\ x_{n1}\ldots x_{na_n}\texttt{->}\ e_n]\!]\ \rho$$

$$\begin{array}{lcl}
= \ \textbf{case}\ \mathsf{EVAL}[\![e]\!]\ \rho\ \textbf{of} & & \\
\quad [\![\ \bot & \to & \bot \\
\quad [\![\ \langle c_1, \epsilon_{11}, \ldots, \epsilon_{1a_1}\rangle & \to & \mathsf{EVAL}[\![e_1]\!]\ (\rho \oplus \{x_{11} \mapsto \epsilon_{11}, \ldots, x_{1a_1} \mapsto \epsilon_{1a_1}\}) \\
\quad [\![\ \ldots & & \\
\quad [\![\ \langle c_n, \epsilon_{n1}, \ldots, \epsilon_{na_n}\rangle & \to & \mathsf{EVAL}[\![e_n]\!]\ (\rho \oplus \{x_{n1} \mapsto \epsilon_{n1}, \ldots, x_{na_n} \mapsto \epsilon_{na_n}\}) \\
\quad \textbf{end} & &
\end{array}$$

$$\mathcal{B}[\![binds]\!] : Env \to Env$$
$$\mathcal{B}[\![x_1 = e_1\texttt{;}\ldots\texttt{;}x_n = e_n]\!]\ \rho = \{x_i \mapsto \mathsf{EVAL}[\![e_i]\!]\ \rho | i \in \{1, \ldots, n\}\}$$

$$strict : (\ Val \to Val\ ) \to Val \to Val$$
$$strict\ f\ x = f\ x\ \text{, if } x \neq \bot$$
$$strict\ f\ \bot = \bot$$

Figure A.2: Denotational semantics of the FSC kernel

_access	::	*Array* α	$\rightarrow$	*Int*	$\rightarrow$	α		
_cons	::	α	$\rightarrow$	*Array* α	$\rightarrow$	*Array* α		
_snoc	::	α	$\rightarrow$	*Array* α	$\rightarrow$	*Array* α		
_head	::	*Array* α	$\rightarrow$	α				
_tail	::	*Array* α	$\rightarrow$	*Array* α				
_nil	::	*Array* α						
_last	::	*Array* α	$\rightarrow$	α				
_init	::	*Array* α	$\rightarrow$	*Array* α				
_append	::	*Array* α	$\rightarrow$	*Array* α	$\rightarrow$	*Array* α		
_replace	::	*Array* α	$\rightarrow$	*Int*	$\rightarrow$	α	$\rightarrow$	*Array* α
_fill	::	*Int*	$\rightarrow$	*Int*	$\rightarrow$	α	$\rightarrow$	*Array* α
_size	::	*Array* α	$\rightarrow$	*Int*				
_liml	::	*Array* α	$\rightarrow$	*Int*				
_limh	::	*Array* α	$\rightarrow$	*Int*				
_subarray	::	*Array* α	$\rightarrow$	*Int*	$\rightarrow$	*Int*	$\rightarrow$	*Array* α
_setindex	::	*Array* α	$\rightarrow$	*Int*	$\rightarrow$	*Array* α		
_isEmpty	::	*Array* α	$\rightarrow$	*Bool*				

Figure A.3: Intrinsic array operations

A.1.1 Identifiers and Operators

tyid $\rightarrow$ $(small\{small|large|digit|'|\_\})_{\text{reservedid}}$
varid $\rightarrow$ *tyid*|*conid*
conid $\rightarrow$ *large*$\{$*small*|*large*|*digit*|'|_$\}$
reservedid $\rightarrow$ `all|array|at|by|case|class|cross|datatype|do|dot|else`
| `for|if|ininfix|infixr|infixl|instance|let|of|ordered`
| `prefix|repeat|return|returns|suffix|then|type|with|when`
| `where|until|unless`

varsym $\rightarrow$ $\{symbol\}_{\text{reservedop}}$
symbol $\rightarrow$ `!|#|$|%|&|*|+|.|/|<|=|>|^|?|@|\|~||`
reservedop $\rightarrow$ `..|::|=>|=|@|\||<-|->|:`

A.1.2 Boolean Literals

boolean $\rightarrow$ `True|False`

$$\_access\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, j \mapsto e_j, \cdots, n \mapsto e_n]\ j = e_j$$

$$\_cons\ a\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n] = [(i-1) \mapsto a, i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n]$$

$$\_snoc\ a\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n] = [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n, (n+1) \mapsto a]$$

$$\_head\ [i \mapsto e_i, \cdots, n \mapsto e_n] = e_i$$

$$\_tail\ [i \mapsto e_i, (i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n] = [(i+1) \mapsto e_{i+1}, \cdots, n \mapsto e_n]$$

$$\_nil = []$$

$$\_last\ [i \mapsto e_i, \cdots, n \mapsto e_n] = e_n$$

$$\_init\ [i \mapsto e_i, \cdots, (n-1) \mapsto e_{i+1}, n \mapsto e_n] = [i \mapsto e_i, \cdots, (n-1) \mapsto e_{n-1}]$$

$$\_append\ [i \mapsto e_i, \cdots, n \mapsto e_n]\ [j \mapsto e_j, \cdots, m \mapsto e_m] = [i \mapsto e_i, \cdots, n \mapsto e_n, (n+1) \mapsto e_j, \cdots, ((m-j)+n+1) \mapsto e_n]$$

$$\_replace\ [i \mapsto e_i, \cdots, j \mapsto e_j, \cdots, n \mapsto e_n]\ j\ a = [i \mapsto e_i, \cdots, j \mapsto a, \cdots, n \mapsto e_n]$$

$$\_fill\ i\ n\ a = [i \mapsto a, \cdots, \cdots, n \mapsto a]$$

$$\_liml\ [i \mapsto e_i, \cdots, n \mapsto e_n] = i$$

$$\_limh\ [i \mapsto e_i, \cdots, n \mapsto e_n] = n$$

$$\_size\ [i \mapsto e_i, \cdots, n \mapsto e_n] = (n-i)+1$$

$$\_subarray\ [i \mapsto e_i, \cdots, j \mapsto e_j, \cdots, k \mapsto e_k, \cdots, n \mapsto e_n]\ j\ k = [j \mapsto e_j, \cdots, k \mapsto e_k]$$

$$\_setindex\ [i \mapsto e_i, \cdots, n \mapsto e_n]\ j = [j \mapsto e_i, \cdots, (n-i)+j \mapsto e_n]$$

$$\_isEmpty\ A = \_primIntIntEqual\ (\_size\ A)\ 0$$

Figure A.4: Intrinsic array operation identities

A.1.3 Numeric Literals

$$
\begin{array}{lcl}
\textit{integer} & \rightarrow & \textit{digit}\{\textit{digit}\} \\
\textit{float} & \rightarrow & \textit{integer.integer}[(\texttt{e}\,|\,\texttt{E})[\texttt{-}\,|\,\texttt{+}]\textit{integer}] \\
 & | & \textit{integer}(\texttt{e}\,|\,\texttt{E})[\texttt{-}\,|\,\texttt{+}]\textit{integer} \\
 & | & \textit{integer.} \\
\textit{double} & \rightarrow & \textit{integer.integer}[(\texttt{d}\,|\,\texttt{D})[\texttt{-}\,|\,\texttt{+}]\textit{integer}] \\
 & | & \textit{integer}(\texttt{d}\,|\,\texttt{D})[\texttt{-}\,|\,\texttt{+}]\textit{integer}
\end{array}
$$

A.2 Expressions

In this section, we describe the syntax and semantics of FSC *expressions*, including their translation into the FSC kernel where appropriate.

exp	→	*aexp* :: *type*	(expression type signature)
	|	exp_0	
exp_0	→	`let` {*decls*[;]} `in` *exp*	(let expression)
	|	\\$apat_1$ ... $apat_n$`->` *exp*	(lambda abstraction $n \geq 1$)
	|	`if` *exp* `then` *exp* `else` *exp*	(conditional)
	|	`case` *exp* `of` {*alts*[;]}	(case expression)
	|	`do` *type computations*	(do (IO) expressions)
	|	*iteration*	(iteration/array expressions)
	|	*fexp*	
fexp	→	*fexp aexp*	(function application)
	|	*fexp* `[`$args_{array}$`]`	(array access)
	|	*fexp* {$args_{exp}$}	(bracketed application)
	|	*aexp*	
aexp	→	*var*	(variable)
	|	*con*	(constructor)
	|	*literal*	
	|	()	(unit)
	|	(*exp*)	(parenthesised expression)
	|	($exp_1, \ldots, exp_k$)	(tuple $k \geq 2$)

The FSC grammar is simplified by handling the parsing of operators from outside the grammar. Table A.1 shows how examples are initially parsed and then transformed to prefix expressions.

The expression	parses as	and is transformed to
`f x + g y`	`((((f x) +) g) y)`	`(+ (f x)) (g y)`
`n!`	`(n !)`	`(! n)`
`(x+)`	`BRACK(x +)`	`(+ x)`

Table A.1: Operator parsing

A.2.1 Variables, Constructors and Operators

var	→	*varid* | *varsym* |(*op*)	(variable)
con	→	*conid* | *consym*	(constructor)
consym	→	*varsym*	
op	→	*varid* | *varsym* | ‘ *varid* ‘	(operator)

A.2.2 Curried Applications and Lambda Abstractions

$$\begin{array}{lcl} exp & \rightarrow & \backslash apat_1 \ \ldots \ apat_n \texttt{ -> } exp \\ fexp & \rightarrow & fexp\ aexp \\ & | & fexp\ \{args_{exp}\} \end{array}$$

A.2.3 Operator Applications

$$\begin{array}{lcll} exp & \rightarrow & exp_1\ op_{infix}\ exp_2 & \text{(infix operator application)} \\ & | & exp\ op_{suffix} & \text{(suffix operator application)} \end{array}$$

A.2.4 Sections

$$\begin{array}{lcl} aexp & \rightarrow & (exp\ op) \\ & | & (op\ exp) \end{array}$$

A.2.5 Conditionals

$$exp \rightarrow \texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3$$

A.2.6 Arrays

$$iteration \rightarrow [e_1, \ldots, e_n] \qquad (k \geq 0)$$

Arrays are of the form $[e_1, \cdots, e_n]$, where $n \geq 0$; the empty array is written `[]`. *Arrays* are the predominant datatype in FSC and hence much work has been done to facilitate their use. Since FSC is based heavily around pattern matching there is a facility to pattern match against array arguments, with many analogies being drawn with *lists* in languages such as Haskell. Standard operations for constructing arrays are shown in Fig. A.5.

A.2.7 Tuples

$$aexp \rightarrow (e_1, \ldots, e_n) \qquad (n \geq 0)$$

Operation	Translation
`A +> a`	append a to the end of A
`a <+ A`	append a to the front of A
`A ++ B`	concatenate A and B
`l <: A`	set the lower bound of A to l
`A :> u`	set the upper bound of A to u
`[]`	The empty array

Figure A.5: Intrinsic operations over arrays

A.2.8 Unit Expressions and Parenthesised Expressions

$$\begin{array}{lcl} \textit{aexp} & \longrightarrow & (e) \\ & | & () \end{array}$$

A.2.9 Arithmetic Sequences and Strides

$$\begin{array}{lcll} \textit{iteration} & \longrightarrow & [e_1 .. e_2] & \text{(range)} \\ & | & [e_1 : e_2 \{: e_3\}] & \text{(stride)} \end{array}$$

A.2.10 Array Comprehensions

$$\begin{array}{lcll} \textit{iteration} & \longrightarrow & [\textit{retexp}_1, \ldots, \textit{retexp}_n \mid \textit{range}\ [\mid \textit{inits}]\ ] & n \geq 1 \\ & | & [\textit{exp}_1; \ldots; \textit{exp}_n \mid \textit{range}\ [\mid \textit{inits}]\ ] & n \geq 1 \\ \textit{retexp} & \longrightarrow & [\textit{aexp}\ \texttt{of}\ ]\ \textit{exp}[(\texttt{when}|\texttt{unless})\ \textit{exp}] & \\ \textit{inits} & \longrightarrow & \textit{init}_1; \ldots; \textit{init}_n & n \geq 1 \\ \textit{init} & \longrightarrow & \textit{pat} = \textit{exp} & \\ \textit{range} & \longrightarrow & \textit{updates}\ (\texttt{while}|\texttt{until})\ \textit{exp} & \\ & | & (\texttt{while}|\texttt{until})\textit{aexp}\ \textit{updates} & \\ & | & \{\textit{updates};\}\ \textit{inrange} & \\ & | & \texttt{all}\ \textit{var} & \\ \textit{updates} & \longrightarrow & \textit{update}_1; \ldots; \textit{update}_n & n \geq 1 \\ \textit{update} & | & \textit{pat} \texttt{<-}\ \textit{exp} & \\ \textit{inrange} & \longrightarrow & \textit{pat}\ \texttt{in}\ \textit{exp}\ [\texttt{at}\ \textit{var}]\ [(\texttt{dot}|\texttt{cross}|;)\ \textit{inrange}] & \end{array}$$

A.2.11 Value Comprehensions

iteration	→	($retval_1, \ldots, retval_n$ | *range* [| *inits*])	$n \geq 1$
retval	→	*exp*[(**when**|**unless**) *exp*]	
inits	→	$init_1; \ldots; init_n$	$n \geq 1$
init	→	*pat* = *exp*	
range	→	*updates* (**while**|**until**) *exp*	
	|	(**while**|**until**)*aexp updates*	
	|	{*updates*;} *inrange*	
	|	**all** *var*	
updates	→	$update_1; \ldots; update_n$	$n \geq 1$
update	|	*pat* <- *exp*	
inrange	→	*pat* **in** *exp* [**at** *var*] [(**dot**|**cross**|;) *inrange*]	

A.2.12 For Expressions

iteration	→	**for** [**initial** {*inits*}] | $range_{for}$ **returns** ($retexp_1, \ldots, retexp_n$)	$n \geq 1$
retexp	→	[*aexp* **of**] *exp*[(**when**|**unless**) *exp*]	
inits	→	$init_1; \ldots; init_n$	$n \geq 1$
init	→	*pat* = *exp*	
$range_{for}$	→	**repeat** {*updates* }(**while**|**until**) *exp*	
	|	(**while**|**until**)*aexp* {**repeat** *updates*}	
	|	{*updates*;} *inrange*	
	|	**all** *var*	
updates	→	$update_1; \ldots; update_n$	$n \geq 1$
update	|	*pat* <- *exp*	
inrange	→	*pat* **in** *exp* [**at** *var*] [(**dot**|**cross**|;) *inrange*]	

A.2.13 Case Expressions

exp	→	**case** *exp* **of** { *alts*[;]}	
alts	→	$alt_1; \ldots; alts_n$	$(n \geq 1)$
alt	→	*pat* -> *exp* [**where** {*decls* [;]}]	
	|	*pat gdexp* [**where** { *decls* [;]}]	
gdpat	→	*gd* -> *exp* [*gdpat*]	
gd	→	| *exp*	

A.2.14 Do Expressions

exp	→	`do` *domain*`{`*actions*`[;]}`	
actions	→	$action_1 ; \ldots ; action_n$	$n \geq 1$
action	→	`return` *exp*	(unit)
	|	*pat* `<-` *exp*	(bind)
	|	*exp*	
domain	→	*type*	(domain choice)

A.2.15 Expression Type-Signatures

exp → *aexp* `::` *type*

A.2.16 Pattern Matching

A.2.16.1 Patterns

pat	→	*apat*	
	|	*con fpats*	
apat	→	*var*[@*apat*]	(as pattern)
	|	*con*	(arity *con=0*)
	|	*literal*	
	|	_	(wildcard)
	|	()	(unit pattern)
	|	(*pat*)	(parenthesised pattern)
	|	$(pat_1, \ldots, pat_k)$	(tuple pattern, $k \geq 2$)
	|	$[pat_1, \ldots, pat_k]$	(array pattern, $k \geq 0$)
	|	$[pat_1, \ldots, pat_k)$	(left edge array pattern, $k \geq 0$)
	|	$(pat_1, \ldots, pat_k]$	(right edge array pattern, $k \geq 0$)
fpats	→	$\{pat_1, \ldots, pat_k\}$[*fpats*]	(application pattern, $k \geq 1$)
	|	*apat*[*fpats*]	

A.3 Declarations and Bindings

module	→	**module** *modid* [*exports*]**where** *body*	
	|	*body*	
body	→	{[*impdecls*;][[*fixdecls* ;]*topdecls* [;]]}	
	|	{*impdecls* [;]}	
topdecls	→	$topdecl_1; \ldots; topdecl_n$	$(n \geq 1)$
topdecl	→	**type** *simple* = *type*	
	|	**datatype** *simple* = *constrs*	
	|	**class**[*context* =>]*class* [**where** {*cbody* [;]}]	
	|	**instance**[*context* =>]*tycls inst* [**where** {*valdefs* [;]}]	
	|	*transformation*	
	|	*decl*	
decls	→	$decl_1; \ldots; decl_n$	$(n \geq 0)$
decl	→	*arraydecl*	
	|	*valdef*	

A.3.0.2 Syntax of Types

type	→	*btype*[^*digit*][-> *type*]	
btype	→	$atype_1 \ldots atype_k$	(arity *tycon* = *k*, $k \geq 1$)
	|	*atype*	
atype	→	*tyvar*	
	|	*tycon*	(arity *tycon* = *0*)
	|	()	(unit type)
	|	(*type*)	(parenthesised type)
	|	$(type_1, \ldots, type_k)$	(tuple type $k \geq 2$)
	|	[*type*]	(array type)

A.3.0.3 Syntax of Class Assertions and Contexts

context	→	*class*	
	|	$(class_1, \ldots, class_n)$	$(n \geq 1)$
class	→	$tycls(type_1, \ldots, type_n)$	$(n \geq 1)$
tycls	→	*conid*	

A.3.1 User-Defined Datatypes

$$
\begin{array}{lcll}
\mathit{topdecl} & \rightarrow & \texttt{datatype}\ \mathit{simple} = \mathit{constrs} & \\
\mathit{simple} & \rightarrow & \mathit{tycon}\ \mathit{tyvar}_1 \ldots \mathit{tyvar}_k & (\text{arity } \mathit{tycon} = k,\ k \geq 0) \\
\mathit{constrs} & \rightarrow & \mathit{constr}_1 \mid \ldots \mid \mathit{constr}_n & (n \geq 1) \\
\mathit{constr} & \rightarrow & \mathit{con}\ \mathit{atype}_1 \ldots \mathit{atype}_k & (\text{arity } \mathit{con} = k,\ k \geq 0)
\end{array}
$$

A.3.1.1 Type Synonym Declarations

$$
\begin{array}{lcll}
\mathit{topdecl} & \rightarrow & \texttt{type}\ \mathit{simple} = \mathit{type} & \\
\mathit{simple} & \rightarrow & \mathit{tycon}\ \mathit{tyvar}_1 \ldots \mathit{tyvar}_k & (\text{arity } \mathit{tycon} = k,\ k \geq 0)
\end{array}
$$

A.3.1.2 Class Declarations

$$
\begin{array}{lcll}
\mathit{topdecl} & \rightarrow & \texttt{class}\ [\mathit{context} \texttt{=>}]\mathit{class}\ [\texttt{where}\ \{\ \mathit{cbody}[;]\}] & \\
\mathit{cbody} & \rightarrow & \mathit{csigns} & \\
\mathit{csigns} & \rightarrow & \mathit{csign}_1; \ldots; \mathit{csign}_n & (n \geq 1) \\
\mathit{csign} & \rightarrow & \mathit{vars} :: [\mathit{context} \texttt{=>}\ \mathit{type} & \\
\mathit{vars} & \rightarrow & \mathit{var}_1, \ldots, \mathit{var}_n & (n \geq 1)
\end{array}
$$

A.3.1.3 Instance Declarations

$$
\begin{array}{lcll}
\mathit{topdecl} & \rightarrow & \texttt{instance}\ [\mathit{context} \texttt{=>}]\mathit{tycls}\ \mathit{inst}\ [\texttt{where}\ \{\ \mathit{valdefs}\ [;]\ \}] & \\
\mathit{inst} & \rightarrow & \mathit{type} & \\
\mathit{valdefs} & \rightarrow & \mathit{valdef}_1; \ldots; \mathit{valdef}_n & (n \geq 1)
\end{array}
$$

A.3.2 Nested Declarations

A.3.2.1 Type Signatures

$$
\begin{array}{lcll}
\mathit{decl} & \rightarrow & \mathit{vars} :: \mathit{type} & \\
\mathit{vars} & \rightarrow & \mathit{var}_1, \ldots, \mathit{var}_n & (n \geq 1)
\end{array}
$$

A.3.2.2 Function and Pattern Bindings

$$
\begin{array}{lcl}
\mathit{decl} & \rightarrow & \mathit{valdef} \\
 & | & \mathit{arraydecl} \\
 & & \\
\mathit{valdef} & \rightarrow & \mathit{lhs} \texttt{ = } \mathit{exp}\ [\texttt{where}\ \{\ \mathit{decls}[;]\}] \\
 & | & \mathit{lhs}\ \mathit{gdrhs}\ [\texttt{where}\ \{\ \mathit{decls}[;]\}] \\
 & & \\
\mathit{lhs} & \rightarrow & \mathit{pat} \\
 & | & \mathit{funlhs} \\
\mathit{funlhs} & \rightarrow & \mathit{var}\ \{\ \mathit{apat}_1, \ldots, \mathit{apat}_k\}\{(\ \{\ \mathit{apat}_1, \ldots, \mathit{apat}_k\}|\mathit{apat})\} \\
 & | & \mathit{apat}\ \mathit{apat}\ \{\mathit{apat}\} \\
 & & \\
\mathit{gdrhs} & \rightarrow & \mathit{gd} \texttt{ = } \mathit{exp}[\mathit{gdrhs}] \\
\mathit{gd} & \rightarrow & |\,\mathit{exp}
\end{array}
$$

A.3.3 Array Declarations

$$
\begin{array}{lcll}
\mathit{arraydecl} & \rightarrow & \texttt{array}\ [\mathit{var}_1[[\mathit{bnd}]..\mathit{bnd}][\mathit{ordering}][\mathit{mutexp}]\texttt{where}\ ] & \mathit{vardecls} \\
\mathit{vardecls} & \rightarrow & \{\mathit{vardecl}; \ldots; \mathit{vardecl}\} & \\
\mathit{vardecl} & \rightarrow & \mathit{var}_1[p_1, \ldots, p_n]\,\mathit{grhs}[\texttt{where}\ \{\mathit{decls}[;]\}] & \\
 & | & \mathit{var}_1[p_1, \ldots, p_n] \texttt{ = } \mathit{exp}[\texttt{where}\ \{\mathit{decls}[;]\}] & \\
\mathit{ordering} & \rightarrow & \texttt{ordered}\ \mathit{ords} & \\
\mathit{mutexp} & \rightarrow & \texttt{overwrites}\ \mathit{var}_2 & (\mathit{var}_1 \neq \mathit{var}_2) \\
\mathit{ords} & \rightarrow & \mathit{ord}\ \{(\texttt{then}|\texttt{and})\mathit{ords}\} & \\
\mathit{ord} & \rightarrow & \texttt{by}\ \mathit{var}\ \texttt{in}\ \mathit{exp}\ \mathit{iteration} & \\
 & | & \mathit{iteration} & \\
 & & & \\
\mathit{bnd} & \rightarrow & (\mathit{exp}_1, \ldots, \mathit{exp}_n) & (n \geq 1) \\
 & \rightarrow & \mathit{exp} &
\end{array}
$$

A.4 Modules

A.4.1 Module Implementations

$$
\begin{array}{lcll}
\mathit{module} & \rightarrow & \texttt{module}\ \mathit{modid}\ [\mathit{exports}]\texttt{where}\ \mathit{body} & \\
 & | & \mathit{body} & \\
\mathit{body} & \rightarrow & \{[\mathit{impldecls}\ ;][[\mathit{fixdecls}\ ;]\mathit{topdecls}\ [;]]\} & \\
 & | & \{[\mathit{impldecls}\ [;]\} & \\
 & & & \\
\mathit{modid} & \rightarrow & \mathit{conid} & \\
\mathit{impdecls} & \rightarrow & \mathit{impdecl}_1; \ldots; \mathit{impdecl}_n & (n \geq 1) \\
\mathit{topdecls} & \rightarrow & \mathit{topdecl}_1; \ldots; \mathit{topdecl}_n & (n \geq 0)
\end{array}
$$

A.4.1.1 Export Lists

$$
\begin{array}{lcll}
\mathit{exports} & \longrightarrow & (\mathit{export}_1, \ldots, \mathit{export}_n) & (n \geq 1) \\
\\
\mathit{export} & \longrightarrow & \mathit{entity} & \\
 & | & \mathit{modid} & \\
\\
\mathit{entity} & \longrightarrow & \mathit{var} & \\
 & | & \mathit{tycon} & \\
 & | & \mathit{tycon}(..) & \\
 & | & \mathit{tycon}(\mathit{con}_1, \ldots, \mathit{con}_n) & (n \geq 1) \\
 & | & \mathit{tycls} & \\
 & | & \mathit{tycls}(..) & \\
 & | & \mathit{tycls}(\mathit{var}_1, \ldots, \mathit{var}_n) & (n \geq 0)
\end{array}
$$

A.4.1.2 Import Declarations

$$\mathit{impdecl} \longrightarrow \texttt{import}\ \mathit{modid}$$

A.4.1.3 Fixity Declarations

$$
\begin{array}{lcll}
\mathit{fixdecls} \longrightarrow \mathit{fix}_1; \ldots; \mathit{fix}_n & & & (n \geq 1) \\
\mathit{fix} & \longrightarrow & \texttt{infix}\ [\mathit{integer}]\mathit{vars} & \\
 & | & \texttt{infixl}\ [\mathit{integer}]\mathit{vars} & \\
 & | & \texttt{infixr}\ [\mathit{integer}]\mathit{vars} & \\
 & | & \texttt{prefix}\ [\mathit{integer}]\mathit{vars} & \\
 & | & \texttt{suffix}\ [\mathit{integer}]\mathit{vars} & \\
\\
\mathit{vars} & \longrightarrow & \mathit{var}_1, \ldots, \mathit{var}_n &
\end{array}
$$

Appendix B

normalCDF: C implementation

The code that follows is the C code to tabulate a cumulative density function of a normal distribution referred to in Chapter 10.

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>

static void normalCDF          (int n);
double calculate               (int n,int i);
static inline double normalPDF(double x);
static inline double integrate(double (*f)(double),double a,double b);

double MU;
double SIGMA;
double pi;
clock_t t1,t2;
double sc;

/* Takes 3 command line arguments
** MU double
** SIGMA double
** n     int
** N is the number of points you wish to evaluate
** i.e. in Schaum book N = 400
*/

int main(int argc,char* argv[])
{
```

```
    int n = argc;
    pi = 3.14159265358979323846;

    if (argc == 4)   {
        n =  atoi(argv[3]);
        MU = atof(argv[1]);
        SIGMA = atof(argv[2]);
        sc = (SIGMA * 2.50663 ); /* 2.50663 = sqrt(2 * pi)); */
        normalCDF(n);
    }
    else {
        printf("Error: 3 arguments expected %d received ",argc);
    }
    return 0;
}


static void normalCDF(int n)
{
    int i;
    double* out;
    double z;
    out = (double*) malloc((unsigned) n * sizeof(double));
    t1 = clock();
    z = 4.0 / ((double) n);
    for (i = 0;i <n;i++) {
        out[i] = integrate(normalPDF,0.0,(double) i * z);
        /* printf("%lf\n",out[i]); */
    }
    t2 = clock();
    printf("clocks %lf\n",
         ((double)(t2 - t1))/((double)CLOCKS_PER_SEC));
}


static inline double normalPDF(double x) {
    register double q = x - MU;
    register double y = (q*q)/(-2.0);
    return exp(y)/sc;
}

/* symmetric n-point Gaussian quadrature.*/

static inline double integrate(double (*f)(double),double a,double b)
```

```
{
    double xi;
    double total;
    total = (b-a)/2.0;
    b     = (b+a)/2.0;
    a     = total;
/* unroll loop */
    xi    = a*0.960289856497536;
    total = 0.101228536200367 * ((*f)(b+xi) + (*f)(b-xi));
    xi    = a*0.796666477413627;
    total += 0.222381034453374 * ((*f)(b+xi) + (*f)(b-xi));
    xi    = a*0.525532409916329;
    total += 0.313706645877887 * ((*f)(b+xi) + (*f)(b-xi));
    xi    = a*0.183434642495650;
    total += 0.362683783378362 * ((*f)(b+xi) + (*f)(b-xi));
    return a * total;
}
```

Appendix C

Lambda Calculus

C.1 Introduction

The development of functional languages has been most influenced by the work of Church on the lambda calculus. This work was motivated by the desire to create a calculus (a syntax for terms and a set of rewrite rules for transforming them) of anonymous functions that captured the computational aspects of functions rather than considering functions as merely sets of argument/result pairs.

In the lambda calculus all functions are presented in prefix form. For example,

$$(+\ 2\ 3)$$

denotes the expression $(2+3)$. If we wish to evaluate an expression we select a reducible expression, or redex, in our main expression and reduce it. In this example the reduction would be:

$$(+\ 2\ 3) \Rightarrow 5$$

If our expression was

$$(+\ (*\ 1\ 2)\ 3)$$

then $(+(*12)3)$ would not be a redex since $+$ must be applied to two numbers before we can reduce it. However, $(*\ 1\ 2)$ is a redex and so we may reduce the expression:

$$(+\ (*\ 1\ 2)\ 3) \Rightarrow (+\ 2\ 3) \Rightarrow 5$$

If an expression contains more than one redex then we have a choice of which to reduce first. This issue is discussed later.

C.2 Definitions

To denote function application we use juxtapositioning and write

$$f\ x$$

to denote the function f applied to the argument x. We may also define new functions using lambda abstractions:

$$\lambda x.(+\ x\ 1)$$

This is read as

λ		x	.	$(+$	x		$1)$
The function	of	x	which	adds	x	to	1

C.3 The Pure Untyped Lambda Calculus

In the pure untyped lambda calculus we have identifiers and lambda expressions where an expression e in Exp can be either an identifier x, an abstraction $\lambda x.e$ or an application $e_1\ e_2$

Identifier	x		
Expression	e	::=	x
		|	$e_1\ e_2$
		|	$\lambda x.e$

In our previous example the expression $\lambda x.(+\ x\ 1)\ 2$ would be an example of an application of the abstraction $\lambda x.(+\ x\ 1)$ to the expression 2 where x is an identifier[1].

The expression $\lambda x.e$ is an example of an abstraction, the mechanism by which the notion of a function is captured without the need to bind a name to it. Informally this expression can be read as "the function which when applied to x, returns e". Lambda abstractions can often be found in actual functional programs. For example the following two Haskell definitions are equivalent:

```
> f = \x -> x * x
> g x = x*x
```

where $\lambda x.e$ would be written as `\x -> e`.

The expression $(e_1\ e_2)$ is an application, the mechanism by which the notion of function application is captured. By convention function application is assumed to be left associative and so we can write $((e_1\ e_2)\ e_3)$ as $(e_1\ e_2\ e_3)$. This process was carried out above where we wrote $(+\ 2\ 3)$ rather than $((+\ 2)\ 3)$.

C.4 Rewrite Rules

The rewrite rules of the λ-calculus depend on the notion of the substitution of an expression e_1 for all free occurrences of an identifier x in an expression e_2, which we write[2] as $[e_1/x]e_2$. So

$$[5/x](+\ x\ 1) \Rightarrow (+\ 5\ 1).$$

However, when performing these substitutions the scope of a variable must be respected in the same sense as mentioned earlier. As a result of this complication the definition of substitution, although

[1] Technically this expression is not taken from the pure untyped lambda calculus but from the lambda calculus with constants. Also dealing with built in functions such as addition would require the use of δ rules.

[2] In some texts the notation $e_2[e_1/x]$ is also used.

conceptually simple, is somewhat laborious. To define substitution we must first make the distinction between free and bound variables occurring inside an expression.

An occurrence of the variable v inside the expression e is said to be ***bound*** if it occurs within a sub-expression of e with the form $\lambda v.e_1$, and is *free* otherwise. Or, more formally, the set of free variables of an expression e is defined as

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\ FV(\lambda x.e) & = & FV(e) - \{x\} \end{array}$$

and x is free in e if (and only if) $x \in FV(e)$. For example

$$\begin{array}{lcl} FV(\lambda x.\lambda y.x\ y\ y) & = & \{\} \\ FV(\lambda x.x\ y\ y) & = & \{y\} \\ FV(x\ y\ y) & = & \{x, y\} \end{array}$$

We now can define the substitution $[e_1/x]e_2$ inductively [48] over identifiers, applications and abstractions as

$$[e/x_i] \quad x_j \quad = \begin{cases} e, & \text{if } i = j \\ x_i, & \text{otherwise} \end{cases}$$

$$[e_1/x] \quad (e_2\ e_3) \quad = ([e_1/x]e_2)([e_1/x]e_3)$$

$$[e_1/x_i] \quad (\lambda x_j.e_2) \quad = \begin{cases} \lambda x_j.e_2, & \text{if } i = j \\ \lambda x_j.[e_1/x_i]e_2, & \text{if } i \neq j \wedge x_j \notin FV(e_1) \\ \lambda x_k.[e_1/x_i]([x_k/x_j]e_2), & \text{otherwise} \\ & \text{where } k \neq i, k \neq j, x_k \notin FV(e_1) \cup FV(e_2) \end{cases}$$

The last rule resolves name conflicts by making a name change if necessary. This may be more easily understood as

$$\begin{array}{lcll} [E/x](\lambda x.F) & = & \lambda x.F & \\ [E/x](\lambda y.F) | y \neq x & = & \lambda y.[E/x]F, & \text{if } x \text{ does not occur free in } E \\ & = & \lambda y.[E/x]([z/y]F), & \text{otherwise} \\ & & & \text{where } z \text{ is a new variable name} \\ & & & \text{which does not occur free in } E \text{ or } F. \end{array}$$

The following example, taken from [48], demonstrates application of all three rules:

$$[y/x](\lambda y.x)(\lambda x.x)x \equiv (\lambda z.y)(\lambda x.x)y$$

C.5 Reductions and Conversions

We define three simple rewrite rules on lambda expressions. These rules define steps that may be used in simplifying terms which we write in the form $e_1 \rightarrow e_2$.

1. α-conversion (renaming):

$$\lambda x_i.e \rightarrow \lambda x_j.[x_j/x_i]e, \text{where } x_j \notin FV(e).$$

2. β-reduction (application):

$$(\lambda x.e_1)e_2 \rightarrow [e_2/x]e_1.$$

3. η-reduction (regarding functions with the same external behaviour equal):

$$\lambda x.(e\ x) \rightarrow e, \text{if } x \notin FV(e).$$

Informally, for a function f:

- α-conversion says that the definition $f(x) = x + 1$ is the same as $f(y) = y + 1$.
- β-reduction says that $f(e) \rightarrow e + 1$
- η-reduction says that we can regard the function $\lambda x.f(x)$ to be the same as f, i.e. if a function takes an argument and simply passes it to another function then these two functions are equivalent.

An example of each rule is :

$$\begin{array}{lll} (\lambda x.+\ x\ 1) & \xrightarrow{\alpha} & (\lambda y.+\ y\ 1) \\ (\lambda x.f(x)) & \xrightarrow{\eta} & f \\ (\lambda x.+\ x\ 1)\ 2 & \xrightarrow{\beta} & (+\ 2\ 1) \end{array}$$

We write $e_1 \xrightarrow{*} e_2$ if e_2 can be derived from zero or more β/η-reductions or α-conversions. We also have $\beta-$ and $\eta-$ conversions which are the same as $\beta-$ and $\eta-$ reductions other than they can happen in both directions and hence are denoted $e_1 \leftarrow e_2$.

C.6 Normal Forms and Confluence of Reductions

A lambda expression is said to be in normal form if it cannot be reduced further by using β- or η-reduction. This is often what we think of as the value of an expression or the result of a computation.

Note: some expressions have no normal form such as

$$(\lambda x.(x\ x))\ (\lambda x.(x\ x))$$

where the reduction process is non-terminating. If normal form does exist we would wish to be able to find it and also would like it to have a unique value. The Church Rosser theorems give positive results in both these cases.

- **CHURCH-ROSSER THEOREM I**
  If $e_0 \overset{*}{\leftrightarrow} e_1$ then there exists an e_2 such that $e_0 \xrightarrow{*} e_2$ and $e_1 \xrightarrow{*} e_2$. In other words if e_0 and e_1 are intra-convertible then there exists a third term (possibly the same as e_0 or e_1) to which they can both be reduced.

- **COROLLARY**
  No lambda expression can be converted to two distinct normal forms (ignoring differences due to α-conversion).

  What this means is that a normal form is unique (up to renaming of variables) and how we arrive at it does not matter (the order of evaluation is irrelevant). We say that $\rightarrow$ is *confluent* or $\rightarrow$ has the *Church Rosser property* if for all e_0, e_1, e_2 such that $e_0 \stackrel{*}{\rightarrow} e_1$ and $e_0 \stackrel{*}{\rightarrow} e_2$ there exists e_3 such that $e_1 \stackrel{*}{\rightarrow} e_3$ and $e_2 \stackrel{*}{\rightarrow} e_3$.

So we know that the normal form of an expression is unique but the question still remains whether it is always possible to find it. To answer this we define two reduction strategies.

C.7 Order of Reduction

A *normal-order* reduction is a sequential order reduction in which whenever there is more than one redex (reducible expression), the left-most one is chosen first. An *applicative-order* reduction is a sequential reduction in which the left-most-innermost redex is chosen first.

- **CHURCH-ROSSER THEOREM II**

  If $e_0 \stackrel{*}{\rightarrow} e_1$ and e_1 is in normal form then there exists a normal order reduction from e_0 to e_1.

  What this means is that if a normal form of an expression exists then we can always find it by using normal order reduction.

As it turns out, applicative order reduction is not always adequate, for instance using applicative order reduction to reduce $(\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x))$ would result in $(\lambda x.x\ x)$ (the inner-most left-most redex) being applied to $(\lambda x.x\ x)$ yielding $(\lambda x.x\ x)$ with which to replace $(\lambda x.x\ x)(\lambda x.x\ x)$ and so

$$\begin{array}{ll} & (\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x)) \\ \rightarrow & (\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x)) \\ \rightarrow & \quad \ldots \text{ad infinitum} \end{array}$$

However, using normal order reduction to reduce $(\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x))$ would result in $\lambda x.y$ (the left-most redex) being applied to $(\lambda x.x\ x)(\lambda x.x\ x)$ yielding y.

$$\begin{array}{ll} & (\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x)) \\ \rightarrow & \quad\quad\quad y \end{array}$$

C.8 Strictness and Laziness

The normal-order reduction rules of the lambda calculus are the most general in that they can guarantee to produce the normal form of an expression if one exists. Given this, it is natural to consider using normal-order reduction as the computational basis of a programming language.

Unfortunately a naive implementation of this is hopelessly inefficient. To see why consider the following examples [48]:

$$\begin{array}{ll} & (\lambda x.(+\ x\ x))(*\ 5\ 4) \\ \rightarrow & (+(*\ 5\ 4)(*\ 5\ 4)) \\ \rightarrow & (+\ 20\ (*\ 5\ 4)) \\ \rightarrow & (+\ 20\ 20) \\ \rightarrow & 40 : 4 \text{ reductions} \end{array}$$

where the multiplication $(*54)$ is done twice. In other circumstances this could be an arbitrarily large computation which is computed as many times as there are occurrences of the formal parameter. For this reason normal order reduction is often associated with a call-by-name strategy of parameter passing. A solution to this is to use a reduction rule other than normal-order such as applicative-order reduction where evaluation of the above expression would result in the following

$$\begin{array}{ll} & (\lambda x.(+\ x\ x))(*\ 5\ 4) \\ \rightarrow & (\lambda x.(+\ x\ x))\ 20 \\ \rightarrow & (+\ 20\ 20) \\ \rightarrow & 40 : 3 \text{ reductions} \end{array}$$

where the argument is evaluated before the application of the lambda abstraction. Thus applicative-order reduction is often associated with a call-by-value strategy of parameter passing. However, the disadvantage of this is that it may perform more reductions than is necessary in order to reach normal-form or, as mentioned earlier, may never reach normal-form. Despite this apparent drawback many functional languages, such as ML, Hope and pure Lisp, use a version of applicative-order semantics and enjoy quite efficient implementations using much of the same call-by-value compiler technology used in imperative languages [48].

The efficiency problem with normal order reduction is based around the fact that the lambda calculus is normally described as *string reduction* which prevents any sharing from occurring. However, if we describe it as a *graph reduction* [99] then we may implement sharing by the use of pointers:

$$\begin{array}{ll} & (\lambda x.(+\ x\ x))(*\ 5\ 4) \\ \rightarrow & \text{let } \chi = (*\ 5\ 4) \text{ in}(+\ \backslash\ \backslash) \\ \rightarrow & \text{let } \chi = 20 \text{ in}(+\ \backslash\ \backslash) \\ \rightarrow & 40 : 3 \text{ reductions} \end{array}$$

which takes the same number of steps as the applicative-order reduction sequence. This form of reduction in which re-computation is avoided is called call-by-need, or *lazy evaluation*. The key features are that it possesses the full power of normal-order reduction and arguments are evaluated at most once. From a *number of reductions* point of view it is more efficient than applicative-order reduction in that "at most once" may amount to no computation. However, this apparent efficiency is amortised against the cost of implementing lazy graph reduction on conventional hardware which seem more suited to call-by value strategies. Representing unevaluated parts of the graph involves implementing closures or thunks whose cost is often non-trivial.

Appendix D

Polymorphic Type Inference

D.1 Polymorphic Type Inference

The purpose of this appendix is to serve as an introduction to polymorphic type inference used in the basic type system which Haskell, Miranda and FSC have in common. This is not presented in a rigorous 'from the ground up' style approach. Rather, we assume the reader has an understanding of the notion of *types* and how these are used in practice in programming languages but has had little experience with notions such as *polymorphism* or *polymorphic types*. This appendix is organised as follows:

- Section D.2 reviews the basic concepts of, and notation for, types.
- Section D.3 introduces the idea of polymorphism.
- Section D.4 shows informally how the principal type of an expression is inferred.
- Section D.5 makes this process more concrete, clarifies the rules of type inference and presents these rules as an abstract algorithm.

This is presented in terms of a *very* simple applicative language which is the *extended lambda calculus*.

D.2 Notation for Types

We begin by defining three classes of type:

- Ground types
- Constructed types
- Type variables

Ground types are types such as *Integer*, *Real*, *Character*, etc. Constructed types are types made from other types such as functions, lists and tuples, and type variables play the same role as variables in mathematics, i.e. a type variable α can stand for a type in the same way that a variable n stands for a number. We use Greek characters to represent type variables. The notation

$$x :: \alpha$$

means that x is of type α. Other examples include

$$\begin{array}{ll} 10 & :: \ \mathit{Integer} \\ \texttt{'c'} & :: \ \mathit{Character} \end{array}$$

where 10 is of ground type *Integer* and `'c'` is of ground type *Character*.

D.2.1 Constructed Types

Constructed types consist of a type-forming operator and a list of arguments. Some examples of constructed types are

Empty	::	*List* α	list of type α	former = List
truncate	::	*Real* $\rightarrow$ *Integer*	function from integers to reals	former = ($\rightarrow$)
*Leaf*1	::	*Tree Integer*	a tree of integers	former = Tree

It is quite common for some of these formers to be syntactically sugared in functional languages, for example List(α) is often written as [α].

D.3 Polymorphism

Many of the functions defined in a functional program are to some extent indifferent to the types of their arguments. The archetypal example being the identity function `id`

```
> id x = x
```

which can be applied to any type i.e.

```
> id 1          -- = 1
> id 'a'        -- = 'a'
> id "hello"    -- = "hello"
> id (1,'a')    -- = (1,'a')
```

In a sense `id` is *indifferent to the type of its argument* although it always returns a value of the *same* type as that of its argument. For the examples above the types of `id` would be

```
> id 1          -- id::INT          -> INT
> id 'a'        -- id::CHAR         -> CHAR
> id "hello"    -- id::STRING       -> STRING
> id (1,'a')    -- id::(INT,CHAR) -> (INT,CHAR)
```

We capture this generality of typing by saying that the type of `id` is $\forall\alpha : \alpha \rightarrow \alpha$. In this example α is known as a *generic*, or *schematic*, variable and it is often the case that we take all variables to be implicitly schematic and omit the $\forall$ clause. So the type of `id` is written as

$$\texttt{id} :: \alpha \rightarrow \alpha$$

D.3.1 The Functions `length` and `map`

Often we want to define functions which work on all types which are more useful than the above identity function. An example is `length`:

```
> length []     = 0
> length (x:xs) = plus 1 (length xs)
```

This function can be applied to lists of any element type and returns the length as an integer. Hence the type of `length` is

$$\texttt{length} :: [\alpha] \texttt{->} \texttt{INT}$$

Other useful polymorphic functions are `map` and `foldr` with definitions as follows:

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

> foldr f z []     = z
> foldr f z (x:xs) = f x (foldr f z xs)
```

whose types are

$$\begin{array}{lll} \texttt{map} & :: & (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \texttt{foldr} & :: & (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta \end{array}$$

D.4 Type Inference

In order to explain the manner in which principal types are inferred we consider the following example:

$$\text{let } f\ x = x \text{ in } f\ 3$$

which is translated into

$$\text{let } f = \lambda x.x \text{ in } f(3)$$

Informally the process of type inference is

- infer type of $\lambda x.x$
- bind f to this type
- infer type of $f(3)$ under the above inferences

where the type of $\lambda x.e$ is found as follows:

- create a non-polymorphic type variable τ_1 and bind it to x
- infer the type τ_2 of e under the above
- unify τ_1 with the type of x to give τ_3
- type of $\lambda x.x$ is inferred as $\tau_3 \rightarrow \tau_2$
- make τ_1 polymorphic

giving the type of $\lambda x.x$ as $\forall\sigma : \sigma \rightarrow \sigma$. The type of $f(e)$ is now inferred as follows:

- create a new type variable τ
- infer the type of f $(\sigma_1 \rightarrow \sigma_2)$
- infer the type of 3 (σ_3)
- unify $\sigma_1 \rightarrow \sigma_2$ and $\sigma_3 \rightarrow \tau$ to give $\tau_3 \rightarrow \tau_4$
- type of $f(e)$ is τ_4

D.5 Type Rules

The rules we use to infer types in the above style are often written as abstract algorithms. For instance, the statement:

$$A.x : \tau \vdash x : \tau$$

reads "from the set of assumptions A and the assumption that x has type τ we may deduce that x is of type τ", and the rule:

$$\frac{A \vdash f : \sigma \rightarrow \tau \qquad A \vdash e : \sigma}{A \vdash (f\ e) : \tau}$$

reads "If, from the set of assumptions A, we may deduce that f has type $\sigma \rightarrow \tau$ and we may also deduce that e has type σ then we may deduce from A that $f(e)$ has type τ". The type rules for standard Hindley-Milner type inference are given in Fig. C.2. with Robinson's unification algorithm given in Fig. C.1.

Unification	
$unify_\theta(v_1, v_2)$	succeeds; v_1 and v_2 are bound to the same variable in θ.
$unify_\theta(\chi_1[s_{11}, \ldots, s_{1m}], \chi_2[s_{21}, ts, s_{2n}])$	succeeds if $\chi_1 = \chi_2 \wedge m = n$ $\wedge unify_\theta(s_{11}, s_{21}) \wedge \ldots \wedge unify_\theta(s_{1m}, s_{2n})$, and fails otherwise.
$unify_\theta(v, \chi[s_1, \ldots, s_n])$	succeeds binding v to $\chi[s_1, \ldots, s_n]$ in θ if v does not occur in $s_1 \ldots s_n$, and fails otherwise
$unify_\theta(\chi[s_1, \ldots, s_n], v)$	succeeds in a similar manner to the definition above

Figure D.1: Robinson's unification algorithm

Type rules		
TAUT	$A.x : \tau \vdash x : \tau$	
COND	$\dfrac{A \vdash e_0 : bool \quad A \vdash e_1 : \tau \quad : A \vdash e_2 : \tau}{A \vdash (if\ e_0\ then\ e_1\ else\ e_2) : \tau}$	
ABS	$\dfrac{A.x : \sigma \vdash e : \tau}{A \vdash (\lambda x.e) : \sigma \rightarrow \tau}$	
LET	$\dfrac{A \vdash e_0 : \sigma \quad A.x : \sigma \vdash e_1 : \tau}{A \vdash (let\ x = e_0\ in\ e_1) : \tau}$	
GEN	$\dfrac{A \vdash e : \tau}{A \vdash e : \forall \alpha.\tau}$	(α not free in A)
SPEC	$\dfrac{A \vdash a : \forall \alpha.\tau}{A \vdash e : [\sigma/\alpha]\,\tau}$	
COMB	$\dfrac{A \vdash e_0 : \sigma \rightarrow \tau \quad A \vdash e_1 : \sigma}{A \vdash (e_0\ e_1) : \tau}$	

Figure D.2: Type inference rules

Appendix E

Monads and Imperative Functional Programming

The concept of a monad originally stems from category theory where it is defined as a triple (M, η, μ) where M is a functor and $\eta : MM \rightarrow M, \mu : I \rightarrow M$ are natural transformations such that the three monad laws are satisfied:

$$\begin{aligned} \mu \circ \eta &= id \\ \mu \circ M\eta &= id \\ \mu \circ M\mu &= \mu \circ \mu \end{aligned}$$

In functional programming terms a functor is any function *map* together with a type constructor M such that *map* takes functions $\alpha \rightarrow \beta$ into functions $M\,\alpha \rightarrow M\,\beta$ and satisfies

$$\begin{aligned} map\ id &= id \\ map\,(g \circ f) &= map\,g \circ map\,f \end{aligned}$$

and a natural transformation can be thought of as a polymorphic function which reshapes a data structure without affecting the constituents of its arguments. It turns out that the well-known functions

$$\begin{aligned} concat &:: [[\alpha]] \rightarrow [\alpha] \\ zip &:: ([\alpha], [\beta]) \rightarrow [(\alpha, \beta)] \\ take\ n &:: [\alpha] \rightarrow [\alpha] \\ fst &:: (\alpha, \beta) \rightarrow \alpha \end{aligned}$$

are all natural transformations.

E.0.1 Monads in Functional Programs

In Haskell the triple (M, η, μ) is written as $(map, unit, join)$

$$\begin{aligned} map &= (\alpha \rightarrow \beta) \rightarrow (M\,\alpha \rightarrow M\,\beta) \\ unit &:: \alpha \rightarrow M\,\alpha \\ join &= M\,(M\,\alpha) \rightarrow M\,\alpha \end{aligned}$$

and, as before, must satisfy the three monad laws for (*map*, *unit*, *join*) to form a monad. However, the manner in which monads are used in functional programs leads to an alternative, though equivalent, definition via an abstract datatype $\mathcal{M}$ with two operations *unit* and *bind*

$$\begin{array}{lll} unit & :: & \alpha \to \mathcal{M}\alpha \\ bind & :: & \mathcal{M}\alpha \to (\alpha \to \mathcal{M}\beta) \to \mathcal{M}\beta \end{array}$$

satisfying

$$\begin{array}{llrcl} (1) & \text{Left unit :} & (unit\ a)\ `bind`\ k & = & k\ a \\ (2) & \text{Right unit :} & m\ `bind`\ unit & = & m \\ (3) & \text{Associative :} & m\ `bind`\ \lambda a.(k\ a\ `bind`\ h) & = & (m\ `bind`\ k)\ `bind`\ h \end{array}$$

with the following equivalences holding

$$\begin{array}{lcl} map\ f & = & \lambda x.bind\ x\ (unit \circ f) \\ join & = & \lambda x.bind\ x\ (map\ id) \\ bind\ x\ f & = & (join \circ map\ f)\ x \end{array}$$

Before proceeding we introduce an auxiliary operation

$$\begin{array}{lcl} f \star g & = & \lambda x.bind\ (f\ x)\ g \\ & = & join \circ map\ g \circ f \end{array}$$

which allows us to reformulate the above laws as:

$$\begin{array}{llrcl} (1) & \text{Left unit :} & unit \star x & = & x \\ (2) & \text{Right unit :} & x \star unit & = & x \\ (3) & \text{Associative :} & f \star (g \star h) & = & (f \star g) \star h \end{array}$$

which is a lot clearer. In the rest of this appendix monads will be built using *unit* and *bind*, and reasoned about using Kleisli composition $\star$. In categorical terms this is known as a *Kleisli* category with $\mathcal{M}$ -resultric functions as morphisms, *unit* as identity and $\star$ as composition. We denote this category as $\mathcal{A}_\star^{\mathcal{M}}$ and denote the monad $(\mathcal{M}, unit_{\mathcal{M}}, bind_{\mathcal{M}})$ as $\mathcal{A}^{\mathcal{M}}$.

E.0.2 Monads In Practice

The reason monads are usually defined in terms of *unit* and *bind* rather than *unit* and $\star$ is due to the use of these operations when modelling computational behaviour in a purely functional manner.

Consider the problem of uniquely numbering the leaves of a binary tree. In Haskell this could be implemented by explicitly passing state as

```
> data Tree a = Leaf a
>             | (Tree a) :^: (Tree a)
```

```
> number (Leaf a)          n = (Leaf (a,n),n+1)
> number (left :^: right) n = let
>                                 (nleft,n1)  = number n  left
>                                 (nright,n2) = number n1 right
>                             in
>                                 (nleft :^: nright,n2)
```

However, if we form a monad with an extra operation *get* defined as:

```
> type M a = Int -> (Int,a)

> bind x f = \a -> let (i,r) = x a in f r i
> unit x   = \a -> (a,x)
> get      = \a -> (a+1,a)
```

we may model this passing of state

```
> number (Leaf a) = get `bind` \x ->
>                   unit (Leaf (a,x))

> number (x :^: y) = number x `bind` \x ->
>                    number y `bind` \y ->
>                    unit (x :^: y)
```

and the computation is structured and less likely to suffer from *silly errors* introducted by explicitly passing state. Other computational behaviours which monads model include exceptions, I/O and non-determinism. Examples of use in these areas may be found in [98].

Appendix F

Finite Element Code

This Appendix contains the FSC finite element code from Chapter 10

```
-------------------------------------------------------------------
-------------------------------------------------------------------
--
--  Direct translation of functions from Dwyer's
--  Functional programming for finite elements
--
-------------------------------------------------------------------
-------------------------------------------------------------------

   PHI :: DOUBLE -> INT -> INT -> [DOUBLE] -> DOUBLE
   PHI x j 1 X = (X[j+1] - x) / (X[j+1] - X[j])
   PHI x j _ X = (x - X[j])   / (X[j+1] - X[j])

-------------------------------------------------------------------


   DPHIDX :: DOUBLE -> INT^2 -> [DOUBLE] -> DOUBLE
   DPHIDX x j 1 X = ~1/(X[j+1] - X[j])
   DPHIDX x j _ X =  1/(X[j+1] - X[j])

-------------------------------------------------------------------

   FINT :: DOUBLE -> INT^3 ->[DOUBLE] -> DOUBLE
   FINT x j i k X | k == 3    = 2/x^2 * PHI{x,j,i,X}
                  | otherwise = DPHIDX{x,j,i,X} * DPHIDX{x,j,k,X} * x

-------------------------------------------------------------------
```

```
GQUAD :: DOUBLE^2 -> INT^3 -> [DOUBLE]^3 -> DOUBLE
GQUAD b a j i k X xs ws
     = sum[pr*w*FINT{x*pr+pm,j,i,k,X}| w in ws dot x in xs]
                                where pm = (b+a)/2
                                      pr = (b-a)/2
```

```
mkL :: [DOUBLE]^3->INT->[DOUBLE]
mkL X xs ws N
  = [   GQUAD X[j+1] X[j]   j     2 2 X xs ws +
        GQUAD X[j+2] X[j+1] (j+1) 1 1 X xs ws ;
        GQUAD X[j+2] X[j+1] (j+1) 1 2 X xs ws |
    j in [1..N-2]] +> GQUAD X[N] X[N-1] (N-1) 2 2 X xs ws
```

```
mkB :: [DOUBLE]^3->INT->DOUBLE^2->DOUBLE
mkB X xs ws N bc1 bc2
 = ~GQUAD X[2] X[1] 1 2 3 X xs ws
   -GQUAD X[3] X[2] 2 1 3 X xs ws
   -bc1 * GQUAD X[2] X[1] 1 1 2 X xs ws
      <+ [ ~GQUAD X[j] X[j-1] (j-1) 2 3 X xs ws
           - GQUAD X[j-1] X[j] j 1 3 X xs ws | j in [3..N-1] ] +>
   ~GQUAD X[N] X[N-1] (N-1) 2 3 X xs ws - bc2
```

```
LP :: [DOUBLE] -> [DOUBLE]
LP (1 <:L:> N) = let array L'[1..N] ordered [1..N] where
                                  L'[1] = L[1]
                                  L'[i] | (i is even) = L[i]
                                        | otherwise = L[i] - L[i-1]^2/L'[i-2]
                 in L'
```

```
BP  :: [DOUBLE]^2 -> [DOUBLE]
BP L' (1 <: B :> N) =  let array B'[1..N] ordered [1..N] where
                                    B'[1] = B[1]
                                    B'[j] = B[j]-L'[2j-2]/L'[2j-3]*B'[j-1]
                       in B'
```

```
A :: INT -> [DOUBLE]^3 -> DOUBLE^2 -> [DOUBLE]
A N X x_g w_g BC1 BC2 = BSUB L' B' where
                        L' = LP L
                        B' = BP L' B
                        L  = mkL X x_g w_g N
                        B  = mkB X x_g w_g N BC1 BC2
```

Appendix G

FSC Code Examples

In this appendix we give numerical examples implemented in FSC

G.1 Jacobi Iteration

```
> jacobi :: [[DOUBLE]] -> [DOUBLE]^2 -> [DOUBLE]
> jacobi A b x = x' where
>   array x'[i] =  (b[i]-sum[A[i,j]*x[j] when i!=j| all j])/ A[i,i]
```

G.2 Gauss-Seidel Iteration

```
> GaussSeidel :: [[DOUBLE]] -> [DOUBLE]^2 -> [DOUBLE]
> GaussSeidel A b (1 <: x :> N) = x' where
>  array x'[1..N] where
>   x'[i] =  (b[i]-sum[A[i,j]*x'[j]|j in [1..i-1]]
                  -sum[A[i,j]*x[j]|j in [i+1..N]])/ A[i.i]
```

G.3 Gaussian Quadrature

```
> gq :: [DOUBLE]^2 -> (DOUBLE->DOUBLE)->DOUBLE^2-> DOUBLE
> gq x w f a b = pr * sum[f1 w_i x_i | w_i in w dot x_i in x]
>         where f1 wi xi = wi * (f (pm+pr*xi) + f (pm-pr*xi))
>               pm = (b+a)/2.0d
>               pr = (b-a)/2.0d
```

G.4 Matrix-Vector Operations

```
> mmult :: [[DOUBLE]]^2 -> [[DOUBLE]]
```

```
> mmult M1 M2 = M3 where
>  array M3[i,j] = sum[ M1[i,j]* M2[k,j] | all k]

> dotproduct :: [DOUBLE]^2 -> DOUBLE
> dotproduct X Y = sum(zipWith (*) X Y)

> MatrixVectorMult :: [[DOUBLE]]->[DOUBLE]->[DOUBLE]
> MatrixVectorMult A x = [ dotprod row x | row in A]

> MatrixPlus :: [[DOUBLE]]^2 -> [[DOUBLE]]
> MatrixPlus A B = C where
>  array C[i,j] = A[i,j] + B[i,j]
```

G.5 LU Decomposition

```
LU_Decompose :: [[DOUBLE]] -> [[DOUBLE]]
LU_Decompose (1 <: A :> N) = LU where
  array  LU[..(N,N)] where
    LU[i,j] | i>j  = (A[i,j]-sum[LU[i,k] * LU[k,j]|k in [1..(j-1)]])/LU[j,j]
            | i<=j = (A[i,j]-sum[LU[i,k] * LU[k,j]|k in [1..(i-1)]]
```

G.6 Newton's Method for Systems of Non-Linear Equations

In this section we present an example of Newton's method extended to deal with a system of n non-linear equations in n unknowns.

G.6.1 Outline of Method

We consider the system of n equations in n unknowns

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned}$$

which is normally written $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ where $\mathbf{f} = [f_1, f_2, \ldots, f_n]^T$ and $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ [107]. We assume a solution exists, that is, we assume there is an $\mathbf{a}$ such that $\mathbf{f}(\mathbf{a}) = \mathbf{0}$. The following code is for a partucular example where

$$\begin{aligned} f_1(x, y) &= \cos(2x) - \cos(2y) - 0.4 \\ f_2(x, y) &= \sin(2y) - \sin(2x) - 1.2 \end{aligned}$$

```
main :: [DOUBLE] -> [DOUBLE]
main init = iterate (NEWTON_IT LU_SOLVE J F) norm 0.000d0 init

f_1,f_2 :: [DOUBLE] -> DOUBLE
f_1 [x,y] = cos(2x) - cos(2y) - 0.4d0;
f_2 [x,y] = sin(2y) - sin(2x) - 1.2d0;

F :: [DOUBLE] -> [DOUBLE->DOUBLE] -> [DOUBLE]
F X = [f x | x in X dot f in Fs]
                where Fs = [f_1,f_2]

J :: [DOUBLE] -> [[DOUBLE]]
J [x,y] = [
            [ ~2sin(2x), 2sin(2y)],
            [ ~2cos(2x), 2cos(2y)]
          ]

norm :: [DOUBLE]-> DOUBLE
norm X = sum[x^2 |x in X]

iterate :: ([DOUBLE]->[DOUBLE])->([DOUBLE]->DOUBLE)
        -> DOUBLE -> [DOUBLE] -> [DOUBLE]
iterate scheme norm tol init
 = (x | while (norm x < tol) x <- scheme x | x = init)

NEWTON_IT SOLVE J F x = SOLVE (J x) (J x * x - F x)

LU_SOLVE :: [[DOUBLE]] -> [DOUBLE]-> [DOUBLE]
LU_SOLVE A b = LU_Subst M b
               where M = LU_Decomp A

LU_Subst :: [[DOUBLE]] -> [DOUBLE] -> [DOUBLE]
LU_Subst LU b = solution where
                 solution = LU_BkSubst LU y
                 y        = LU_FwdSubst LU b

LU_FwdSubst :: [[DOUBLE]] -> [DOUBLE] -> [DOUBLE]
LU_FwdSubst L (1 <: b :> N) = y where
 array y[1..N] where
       y[i] = (b[i] - sum[L[i,k] * y[k]|k in [i..(i-1)]]) / L[i,i]


LU_BkSubst :: [[DOUBLE]] -> [DOUBLE] -> [DOUBLE]
LU_BKSubst U (1 <: y :> N) = x where
```

```
array x[1..N] where
      x[j] = y[j] - sum[U[j,k] * x[k] | k in [j+1..N]]
```

Appendix H

FSC Standard Prelude

```
--  PRIMITIVES FILE
--

--------------- FIXITIES / PRIORITIES -----------------------------------
-- infixl 1000 [ ] ARRAY INDEXING
-- infixl 1000 [ |-> ]  ARRAY REPLACEMENT
-------------------------------------------------------------------------
infixr 950  .                       -- COMPOSITION
infixl 925 ^                        -- EXPONENTIATION
infixl 900  /                       -- DIVISION
infixl 890 |/|                      -- VECTOR DIVISION
infixl 800  *, %                    -- MULTIPLICATION / MODULUS
infixl 790 |*|                      -- VECTOR MULTIPLICATION
infix  700 :+                       -- COMPLEX CONSTRUCTOR
infixl 700  +                       -- ADDITION
infixl 690 |+|                      -- VECTOR ADDITION
infixl 600  -                       -- SUBTRACTION
infixl 590 |-|                      -- VECTOR SUBTRACTION
infixl 500 <:, <+                   -- LOWER BOUND / APPEND LEFT
indixr 490 :>, +>                   -- UPPER BOUND / APPEND RIGHT
infixl 500 ++                       -- CATENATION
infixl 400 ==, !=, <, <=, >=, >     -- EQUALITY / ORDERING
infixl 300 &&                       -- LOGICAL AND
infixl 200 ||                       -- LOGICAL OR
suffix !                            -- FACTORIAL
prefix ~                            -- NEGATION
infix <<                            -- SHIFT LEFT
infix >>                            -- SHIFT RIGHT
infix is                            -- PREDICATE CONNECTIVE
infix div                           -- INTEGER DIVISION
```

```
-- Boolean Functions -----------------------------------------------------

(&&), (||)      :: BOOL -> BOOL -> BOOL
False && x       = False
True  && x       = x

False || x       = x
True  || x       = True

not             :: BOOL -> BOOL
not True         = False
not False        = True

-- Factorials -----------------------------------------------------------

0! = 1
n! = product [i | i in [1..n]]

-- Some standard functions ----------------------------------------------
-- component projections for pairs:

fst             :: (a,b) -> a
fst (x,_)        = x

snd             :: (a,b) -> b
snd (_,y)        = y

-- identity function
id              :: a -> a
id    x          = x

-- constant function
const           :: a -> b -> a
const k _        = k

-- function composition
(.)             :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x        = f (g x)

-- predicate connective -------------------------------------------------

(is) :: a -> (a->BOOL) -> BOOL
X is P = P X

-- error is applied to a string, returns any type, and is everywhere
-- undefined.  Operationally, the intent is that its application
```

```
-- terminates execution of the program and displays the argument
-- string in some appropriate way.

error :: String -> a

-- Standard types, classes and instances -------------------------------------

class NEQ(a) where
    (==) :: a -> a -> Bool
    (!=) :: a -> a -> Bool
A == B = not (A != B)

class ORD(a) where
    (<), (<=), (>=), (>) :: a -> a -> Bool

class PLUS(a,b,c) where
   (+) :: a -> b -> c;

class BY(a,b,c) where
   (/) :: a -> b -> c;

class TIMES(a,b,c) where
   (*) :: a -> b -> c;

class NEGATE(a) where
   (~) :: a -> c;

class VECPLUS(a,b,c) where
   (|+|) :: a -> b -> c;

class VECMINUS(a,b,c)  where
   (|-|) :: a -> b -> c;

class VECBY(a,b,c) where
   (|/|) :: a -> b -> c;

class VECTIMES(a,b,c) where
   (|*|) :: a -> b -> c;

class ADDID(a) where
    zero :: a

class MULID(a) where
    one :: a

class BOUNDS(a) where
    maxVAL :: a
```

```
    minVAL :: a

class TO_INT where
    int :: a -> INT;

class TO_DOUBLE where
    double :: a -> DOUBLE;

class TO_FLOAT where
    float :: a -> FLOAT;

class MATH(a) where
    pi,e                :: a
    exp, log, sqrt, ln  :: a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

-- Boolean type ----------------------------------------------------------

datatype BOOL = False | True

-- Complex Type ----------------------------------------------------------

datatype Complex a = a :+ a

real (re :+ im) = re
imag (re :+ im) = im

instance PLUS(a,b,c)  => PLUS(Complex a,Complex b, Complex c) where
     (x:+y) + (x':+y') = (x+x') :+ (y+y')

instance TIMES(a,b,c) => TIMES(Complex a,Complex b,Complex c) where
     (x:+y) * (x':+y') = (x*x'-y*y') :+ (x*y'+y*x')

-- ETC ---

-- Standard functions ----------------------------------

bitFlip   :: INT -> INT -> INT
(<<)      :: INT -> INT -> INT
(>>)      :: INT -> INT -> INT
bitAND    :: INT -> INT -> INT
bitOR     :: INT -> INT -> INT
bitXOR    :: INT -> INT -> INT
```

```
riffle                          :: Array a -> Array a -> Array a
riffle X Y                       = [ x ; y  | x in X dot y in Y]

deal                            :: Array a -> INT -> Array (Array a)
deal (lo <: X :> hi) N           = [X[lo+i:hi:N]|i in [0..N-1]]

odd_even                         :: Array a -> (Array a,Array a)
odd_even    (lo <: X :> hi)       =  (X[lo:hi:2],X[lo+1:hi:2])

liml,liml                        :: Array a -> INT
bound                            :: Array a -> (INT,INT)
liml   (lo <: X      )           = lo
bounds (lo <: X :> hi)           = (lo,hi)
limh   (      X :> hi)           = hi

head,last                        :: Array a -> a
head (a <+ A)                    = a
last (A +> a)                    = a

tail, init                       :: Array a -> Array a
tail (a <+ A)                    = A
init (A +> a)                    = A

take,drop                        :: INT -> Array a -> Array a
take N (lo <: X      )           = X[lo:lo+N-1]
drop N (lo <: X :> hi)           = X[lo+N:hi]

splitAt                          :: INT -> Array a -> (Array a,Array a)
splitAt N X                      = (take n X, drop n X)

filter                           :: (a ->  BOOL) -> Array a -> Array a
filter p X                       = [x when x is p | x in X]

mapcat                           :: (a -> Array b) -> Array a -> Array b
mapcat f                         = concat.(map f)

map                              :: (a -> b) -> Array a -> Array b
map f X                          = [ f x | x in X];


-------------------------------------------------------------------
-------------------------------------------------------------------
--
--  NOTE: Application of reduce should construct a parallel
--        reduction function
--
-------------------------------------------------------------------
```

```
-----------------------------------------------------------------

reduce                          :: (a -> a -> a) -> a -> Array a -> a;
reduce (*) id []                 = id
reduce (*) id [x]                = x
reduce (*) id [x,y]              = x * y
reduce (*) id (x <+ X +> y)      = x * (reduce (*) id X) * y



length                          :: Array a -> INT
length (lo <: X :> hi)           = hi - lo + 1

foldl                           :: (a -> b -> a) -> a -> Array b -> a
foldl  f z []                    = z
foldl  f z (a <+ A)              = foldl f (f z a) A

foldr                           :: (a -> b -> b) -> b -> Array a -> b
foldr  f z []                    = z
foldr  f z (A +> a)              = foldr f (f z a) A

foldl1                          :: (a -> b -> a) -> Array b -> a
foldl1 f   (x <+ X)              = foldl f x X

foldr1                          :: (a -> b -> b) -> Array a -> b
foldr1 f   (X +> x)              = foldr f x X

scanl, scanr                    :: (a -> a -> a) -> Array a -> Array a;
scanl (*) []                     = []
scanl (*) (x <+ X)               = x <+ [x * y | y in scan (*) X]

scanr (*) []                     = []
scanr (*) (X +> x)               = [x * y | y in scan (*) X] +> x

accumulate                       :: PLUS(b,a,b) => b -> Array a -> b
accumulate frame                  = foldl (+) frame

      -- for all associative operations (+)
      -- accumulate zero [ ... ] = sum [ ... ]

sum                             :: PLUS(a,a,a),ADDID(a)  => Array a -> a
sum                              = reduce (+) zero

sums                            :: PLUS(a,a,a) => Array a -> Array a
sums                             = scanl (+)

product                         :: TIMES(a,a,a),MULID(a) => Array a -> a
```

```
product                          = reduce (*) one

products                        :: TIMES(a,a,a) => Array a -> Array a
products                         = scanl (*)

maximum, minimum                :: ORD(a),BOUNDS(a) => Array a -> a
maximum                          = reduce max minVAL;
minimum                          = reduce min maxVAL;


min,max                         :: ORD(a) => a -> a -> a
min x y | x < y                  = x
        | otherwise              = y

max x y | x > y                  = x
        | otherwise              = y

imin,imax                       :: ORD(a) -> (a,INT) -> (a,INT) -> (a,INT)

imin (x,i) (y,j) | x < y         = (x,i)
                 | otherwise     = (y,j)

imax (x,i) (y,j) | x > y         = (x,i)
                 | otherwise     = (y,j)

imaximum, iminimum               :: ORD(a),BOUNDS(a) => Array a -> (a,INT)
imaximum                         = reduce imax (maxVAL,maxINT)
iminumum                         = reduce imin (minVAL,minINT)

value                           :: Array a -> a
value (X +> x)                   = x
value []                         = error "EMPTY LIST in call to value"

values                          :: Array a => Array a
values                           = id

and, or                         :: Array BOOL -> BOOL
ands, ors                       :: Array BOOL -> Array BOOL

ands                             = scanl  (&&)
and                              = reduce (&&) True

ors                              = scanl  (||)
or                               = reduce (||) False

reverse                         :: Array a -> Array a
reverse  (lo <: X :> hi)         = X[lo:hi:~1]
```

```
concat                          :: Array^2 a -> Array a
concat                           = reduce (++) []

transpose                       :: Array^2 a -> Array^2 a
transpose A                      = B where array B[i,j] = A[j,i]

indexZipWith, zipWith           :: (a->b->c) -> Array a -> Array b -> Array c
indexZipWith f X Y               = z where array z[i] = f X[i] Y[i]

zipWith f X Y                    = [f x y | x in X dot y in Y]


------- I/O ------------------------------------------------------------------

putchar                         :: CHAR                    -> IO()
getchar                         ::                            IO CHAR
getstring                       ::                            IO STRING
putstring                       :: STRING                  -> IO()
openfile                        :: STRING                  -> IO FILE_PTR
closefile                       :: FILE_PTR                -> IO()
fgetchar                        :: FILE_PTR                -> IO CHAR
fputchar                        :: FILE_PTR    -> CHAR     -> IO ()
fgetstring                      :: FILE_PTR                -> IO STRING
fputstring                      :: FILE_PTR    -> STRING   -> IO ()

showInt                         :: INT    -> STRING;
showDouble                      :: DOUBLE -> STRING;
showFloat                       :: FLOAT  -> STRING;

printInt                        :: INT                  -> IO();
printDouble                     :: DOUBLE               -> IO();
printFloat                      :: FLOAT                -> IO();
printBool                       :: BOOL                 -> IO();

argv                            :: Array STRING
argc                            :: INT

print                           :: SHOW(a) => a -> IO()
print                            = putstring.show
```

Appendix I

Domain Theory

I.1 Introduction

Our discussion of the lambda calculus in Appendix C was purely syntactic. We defined a set of rewrite rules with which we could reduce an expression by replacing sub-expressions textually. This observation applies equally to all programming languages in that the syntax alone is not powerful enough to explain the effect of executing a program. Domain theory and denotational semantics (discussed in Appendix I) have been constructed to give meaning to syntactic expressions[1] and hence recursive functional programs. We take the meaning or value of a program to be taken from some set or *domain* with well understood mathematical properties. If we consider a very simple language made up from the following tokens:

$$\mathrm{T} \quad \mathrm{F} \quad \vee \quad \wedge \quad \neg$$

although we can determine what are grammatically correct expressions, such as $\neg$T or T $\wedge$ F, we have not given a meaning to these tokens. Using the usual interpretation we would expect these expressions to mean the same thing - the truth value *false*.

I.2 Terminology

If we take this interpretation of the above language we say that the token T *denotes* the truth value *true*. The important distinction is that our tokens are concrete and the values they denote are mathematical abstractions.

I.3 Domains Versus Sets

Initially it may seem simple to define the domain for the meanings of syntactic expressions as the set of booleans {*true*, *false*} and functions defined on this. However, with this approach we almost

[1] There are many alternatives for describing behaviour to denotational semantics such as operational semantics. However, these systems will not be considered here as denotational semantics suits our purpose.

immediately run into difficulties with inconsistency. Consider the definition of the following function:

$$\begin{array}{lcl} f & :: & Truthvalue \rightarrow Truthvalue \\ f\ x & = & \neg f(x) \end{array}$$

If we consider *Truthvalue* to be the set $Bool = \{true, false\}$ and $\neg$ to be the negation operator we cannot find any function $f :: Bool \rightarrow Bool$ to satisfy our equation representing f. For example, if $f(true) = true$ then $f(true) \neq \neg f(true)$, contradicting the equation. If $f(true) = false$ we see that the result is a similar contradiction. In practice applying f to any element in the set *Bool* would cause the computation to loop forever and there is no value in *Bool* to represent this idea of looping forever. If we now extend *Bool* by adding the new element $\perp$, pronounced "bottom", and

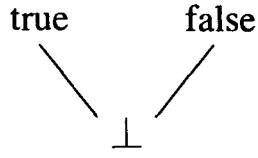


Figure I.1: The boolean domain

refer to this extended set as $Bool_\perp$[2] we can regain consistency by making f a well defined function. This requires that we extend the definition of $(\neg)$ such that $\neg\perp = \perp$ and the definition of f which satisfies its equations is $\forall x \in Bool_\perp, f(x) = \perp$. The bottom element contains less information than the elements of *Bool* as it represents an undefined value, or non-termination. By defining a partial ordering by information content on $Bool_\perp$ in this way, the set becomes what is known as a *domain*. In fact for functions defined only on sets which do not contain compound data items such as lists this domain construction is sufficient to define a consistent semantics. The boolean domain is shown in Fig. H.1.

I.4 Summary

This is all the domain theory needed for the topics discussed in this thesis as the strict semantics of FSC prevent the creation of partial objects. Rather than bog the reader down with unnecessary theory, further introduction can be found in [32], or for the more adventurous [88].

[2]For any set S, $S_\perp$ denotes the set $S \cup \{\perp\}$.

Appendix J

Denotational Semantics

J.1 Introduction

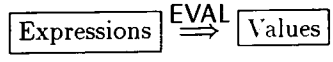
In this appendix we give a very brief introduction to denotational semantics by considering the denotational semantics of the lambda-calculus presented in Appendix C. Denotational semantics differs from other ways of looking at functions in that, rather than considering a function as a sequence of state transitions over time (as in operational semantics) it considers them as a fixed set of associations between arguments and their corresponding values.

In Appendix C we saw how an expression may be evaluated via the repeated application of α, β and η rules. These provide a purely syntactic treatment of the process of evaluation, i.e. given an expression they describe which conversion rules may be applied without any reference to the meaning of these expressions. By itself the lambda-calculus is merely a formal system for manipulating syntactic symbols but why do we suppose that this system models the idea of an abstract function?

Denotational semantics is the necessary stepping stone relating this rewrite system to our intuitive ideas of abstract functions.

J.2 The EVAL Function

The purpose of denotational semantics is to give a value to every expression in a language. An expression is viewed as a purely syntactic object. However, a value is an *abstract* mathematical object. We express the semantics of a language as a (mathematical) function, EVAL, from expressions to values



and we can write equations such as:

$$\mathsf{EVAL}[\![2+2]\!] = 4$$

which is read as 'the meaning of the expression $2+2$ is the abstract value 4' with the use of $[\![]\!]$ emphasising that the argument is a syntactic object. We may regard the expression $2+2$ *denoting*

the value 4, hence the term *denotational semantics.*

J.3 Lambda Calculus

To give meaning to expressions in the lambda-calculus we need to be able to give meaning to variables. It is immediately obvious that in order to do this we need to be able to access the context or scope of a variable. This scope is taken into account by giving EVAL an extra parameter, ρ, called its *environment*, a function mapping variable names to their values. Thus

$$\mathsf{EVAL}[\![x]\!]\,\rho = \rho(x), \qquad x \in Id$$

Applications may be treated similarly:

$$\mathsf{EVAL}[\![E_1\ E_2]\!]\,\rho = (\mathsf{EVAL}[\![E_1]\!]\,\rho)\,(\mathsf{EVAL}[\![E_2]\!]\,\rho)$$

However, lambda abstractions require a little more thought. What should the value of $\mathsf{EVAL}[\![\lambda x.E]\!]\rho$ be? Since it is a function we define its value in terms of an arbitrary argument a:

$$(\mathsf{EVAL}[\![\lambda x.E]\!]\,\rho)\,a = \mathsf{EVAL}[\![E]\!]\,\rho[x \rightarrow a]$$

where the notation $\rho[x \rightarrow a]$ denotes the environment ρ extended with the binding of variable x to value a. More formally, this is:

$$\begin{array}{lcl} \rho[x \rightarrow a]\,x & = & a \\ \rho[x \rightarrow a]\,y & = & \rho\,y, x \neq y \end{array}$$

This completes the denotational semantics for the lambda calculus. This can be extended naturally to cover constants and built-in primitive functions such as the integers, reals, booleans and the standard operators over them. The collection of all the possible values which EVAL can produce turns out to be a domain (see Appendix I) and we may write the following for the function f from Appendix I defined as $f(x) = \neg f(x)$:

$$\mathsf{EVAL}[\![f\ x]\!]\,\rho = \bot$$

This allows us to talk about the properties of functions in a very terse manner, for instance, we may encapsulate the fact that (a+b) is undefined if either a or b are undefined as:

$$\begin{array}{lcll} \mathsf{EVAL}[\![+]\!]\,a\,b & = & a + b, & \text{if } a \neq \bot \wedge b \neq \bot \\ \mathsf{EVAL}[\![+]\!]\,a\,b & = & \bot, & \text{otherwise} \end{array}$$

J.4 Summary

This is a very brief and superficial introduction to denotational semantics meant to be no more than a compact summary of the material required to understand the presentation of some of the sections in this thesis. For a fuller treatment the reader is again referred to [88].

Appendix K

Mathematical Notation

$\mathbf{A} = [a_{ij}]$	matrix having element a_{ij} in row i, column j
$\mathbf{AB}$	matrix multiplication
$\mathbf{a} = [a_1, a_2, \ldots, a_n]$	row n-vector having a_i as the i^{th} component
$\mathbf{A}^T$	transpose of $\mathbf{A}, = [a_{ji}]$
$\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$	column n-vector having b_i as the i^{th} component
$\mathbf{I}$	identity matrix of any size ($i_{jj} = 1$ and $i_{kj} = 0, k \neq j$
$\|\mathbf{A}\|$	determinate of square matrix $\mathbf{A}$
$\mathbf{A}^{-1}$	inverse of $\mathbf{A}$
$\frac{\partial v}{\partial x}$	derivative of v with respect to x
∇	$\nabla v = \frac{\partial v}{\partial x} i + \frac{\partial v}{\partial y} j + \frac{\partial v}{\partial z} k$
$f'(x)$	derivarive of $f(x)$ with respect to x
$f^k(x)$	k^{th} derivarive of $f(x)$ with respect to x
$< \mathbf{x}, \mathbf{y} >$	inner product (usually $\mathbf{x}^T\mathbf{y}$)

References

[1] J.E. Akin. *Application and Implementation of Finite Element Methods*. Academic Press, 1982.

[2] Steve Anderson and Paul Hudak. Compilation of Haskell array comprehensions for scientific computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1990.

[3] Chris Angus. Constructing configurable applications by constructing monads. Technical Report 577, University of Newcastle upon Tyne, June 1997.

[4] Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures, data structures for parallel computing. Technical Report (SCG Memo 269), MIT Lab. for Computer Science, February 1987.

[5] J.W. Backus, F.L. Bauer, J. Green, C. Katx, J. McCarthy, P. Naur (Ed.), A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, H.H. Wegstein, A. van Wijngaarten, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[6] John D. Barrow. *What is Mathematics*. Oxford University Press, 1988.

[7] Robert Bernecky and Tom DeBoni. Sisal development workshop (minutes of meeting). Published on `comp.lang.functional`, 1997.

[8] Stephen J. Bevan. Functional library of numerical functions implemented in Haskell, 1992. `ftp://ftp.cs.chalmers.se/pub/haskell/library/bevan`.

[9] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[10] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[11] E. Bodewig. *Matrix Calculus*. North Holland, 1956.

[12] A.D. Booth. *Numerical Methods*. Butterworths, 1957.

[13] Rick Booth. *Inner Loops, A source-book for fast 32-bit Software Development*. Addison-Wesley, December 1996.

[14] James M. Boyle, Maurice Clint, Stephen Fitzpatrick, and Terence J. Harmer. The construction of numerical mathematical software for the AMT DAP by program transformation. In *Proceedings of CONPAR VAPP V, LNCS 634*, pages 761–767. Springer-Verlag, September 1992.

[15] J.M. Boyle and T.J. Harmer. A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, 2(1):81–126, January 1992.

[16] D.S. Burnett. *Finite Element Analysis-From Concepts to Applications.* Adison-Wesley, 1987.

[17] R.M. Burstall, D.B. MacQueen, and Sannella D.T. Hope: An experimental applicative language. In *The 1980 LISP Conference, Stanford University, Santa Clara Univ. The USP Co.*, pages 136–143, 1980.

[18] David Cann. Retire Fortran? a debate rekindled. *Communications of the ACM*, Aug 1992.

[19] David C. Cann. *SISAL 1.2: A Brief Introduction and Tutorial.* Computing Research Group, Lawrence Livermore National Laboratory, 1.2 edition.

[20] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2), 1987.

[21] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[22] J.W. Cooley and J.W. Turkey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, April 1965.

[23] C.V. Cormack and A.K. Wright. Type-dependent parameter inference. In *ACM SIGPLAN conference on programming language design and implementation*, pages 127–136, June 1990.

[24] J.C. Diaz and K. Shenoi. Domain decomposition and Schur complement approaches to decoupling the well equations in reservoir simulation. *SIAM Journal on Scientific Computing*, 16(1):29–39, 1995.

[25] Iain S. Duff. Data structures, algorithms and software for sparse matrices. Technical Report CSS158, Computer Science and Systems Division, AERE Harwell, Oxfordshire, OX11 ORA, 1984.

[26] J.F. Dwyer. Functional programming for finite elements. *Computers and Structures*, 33(6):1343–1348, 1989.

[27] V.N. Faddeeva. *Computational Methods for Linear Algebra.* Dover,New York, 1959.

[28] V.N. Faddeeva and D.K. Faddeev. *Computational methods of Linear Algebra.* W.H. Freeman & Co., 1963.

[29] Graeme Fairweather. *Finite Element Galerkin Methods for Differential Equations.* MARCEL DEKKER INC, 1978.

[30] FAQ. `comp.lang.functional frequently asked questions`. web reference, 1996.

[31] John Feo, Patrick Miller, Stephen Skedzielewski. Scott Denton, and Cindy Solomon. Sisal 90. In A. P. Wim Bohm and John T. Feo, editors. *High Performance Functional Computing*. `ftp://sisal.llnl.gov/pub/hpfc/hpfc95.html`, pages 35–47, April 1995.

[32] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Adison-Wesley, 1988.

[33] Stephen Fitzpatrick. *The specification of Array-Based Algorithms and the Automated Derivation of Parallel Implementations though Program transformation*. PhD thesis, Department of Computer Science, September 1994.

[34] L. Fox. Practical methods for the solution of linear equations and the inversion of matrices. In *Contributions to the Solution of systems of Linear Equations and the Determination of Eigenvalues*, pages 1–54, 1954.

[35] L. Fox. *An Introduction to Numerical Linear Algebra*. Carendon, 1964.

[36] Len Freeman and Chris Phillips. *Parallel numerical algorithms*. Prentice Hall international series in computer science. New York,Prentice Hall, 1992.

[37] G.R. Gao, Robert Kim Yates, Jack B. Dennis, and Lenore R. Mullin. An efficient monolithic array constructor. Technical Report ACAPS Technical Memo 19, McGill University, School of Computer Science, June 1990.

[38] G.R. Gao, Robert Kim Yates, Jack B. Dennis, and Lenore R. Mullin. A strict monolithic array constructor. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, 1990.

[39] Curtis F. Gerald and Patrick O. Wheatley. *Applied Numerical Analysis*. Adison Wesley, 1994.

[40] G. Golub and C.van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, 1983.

[41] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Some issues in a functional implementation of a finite element algorithm. In *FPCA '93. Conference on Functional Programming Language and Computer Architecture*. ACM Press, June 1993.

[42] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Sparse matrix representations in a functional language. *Journal of Functional Programming*, 6:1–28, 1996.

[43] J. Hammes, S. Sur, and W. Böhm. On the effectiveness of functional language features: Nas benchmark FT. *Journal of Functional Programming*, 7(1):103–123, January 1997.

[44] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language - a case study: the fast Fourier transform. In G. Hains and L. M. R. Mullin, editors, *2nd Arrays, functional languages, and parallel systems (ATABLE)*, pages 52–66, June 1992. Publication 841, Dept. d'informatique et de recherche opirationelle, Univ. de Montreal, Canada.

[45] Pieter H Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Meemster, Emmanuel Chailloux, Christine H Flood, Wolfgang Grieskamp, John H G Van Groningen, Kevin Hammond, Bogumil Hausman, Melody Y Ivory, Richard E Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D Lins, Sandra Loosemore, Niklas Rojemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[46] F.B. Hildebrand. *An Introduction to Numerical Analysis*. McGraw-Hill, 1956.

[47] A.S. Housholder. *Principles of Numerical Analysis*. McGraw-Hill, 1953.

[48] P. Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3), september 1989.

[49] P. Hudak and J. H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[50] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.

[51] John Hughes. Announcing the standard haskell effort. `www.cs.chalmers.se/ rjmh/Haskell`.

[52] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[53] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[54] M.P. Jones. Introduction to GoFER 2.20. Technical report, Oxford programming research group, 1991.

[55] M.P. Jones. Hugs, the Haskell user's Gofer system, 1995.

[56] Simon L Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, March 1993.

[57] Frank Ayres Jr. *Modern Algebra*. McGraw-Hill, 1965.

[58] B.W. Kernigan and D.M. Ritchie. *The C programming language*. Prentice Hall, 1998.

[59] D. Knuth. An empirical study of Fortran programs,. *Software - Practice and Experience*, 1:105 – 133, 1971.

[60] Donald E. Knuth. *Literate Programming*. Lecture Notes Number 27. Leland Stanford Junior University, 1992.

[61] Konstantinos Ksickis. Numerical algorithms in a functional language, December 1994. MSc dissertation, Department of Computing Science,University of Newcastle upon Tyne.

[62] Junxian Liu, Paul Kelly, and Stuart Cox. Functional programming for finite element analysis. Technical report, Department of Computing, Imperial College, 1993.

[63] M. Marcus. *Basic Theorems in Matrix theory.* National Bureau of Standards Applied Mathematical Series, No 57, U.S. Government Printing Office, 1960.

[64] John H. Mathews. *Numerical Methods for Mathematics, Science, and Engineering.* Prentice-Hall, 1992.

[65] John A. McCrory. Numerical algorithms in Haskell and C++, a comparative study. May 1995. BSc dissertation, Department of Computing Science, University of Newcastle upon Tyne.

[66] Bertrand Meyer. *Eiffel, the language.* Prentice Hall, 1991.

[67] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT, Aug 1990.

[68] Manoj Misra. *Performance modelling of replication protocols.* PhD thesis, The University of Newcastle upon Tyne, 1997.

[69] A.R. Mitchell and R. Wait. *Finite Element Analysis and Applications.* Wiley, 1985.

[70] R. Morrison, A. Dearle, R.C.H. Connor, and A.L. Brown. An ad hoc approach to the implementation of polymorphism. *Transactions on Programming Languages and Systems*, 13(3):342–372, 1991.

[71] R.S. Nikhil, K. Pingali, and Arvind. Id nouveau. Technical Report Computation Structures Group Memo 265, Laboratory for Computer Science, MIT, 1986.

[72] John Ophel and Dominic Duggan. Multi-parameter parametric overloading. Submitted for publication , `http://nuada.uwaterloo.ca/ dduggan/papers/`, 1995.

[73] Rex L. Page and Brian D. Moe. Experience with a large scientific application in a functional language. In *FPCA '93. Conference on Functional Programming Language and Computer Architecture.* ACM Press, June 1993.

[74] G.M. Papadopoulos. *Implementation of a general purpose dataflow multiprocessor.* PhD thesis, Laboratory for Computer Science, MIT, August 1988.

[75] L.C. Paulson. *ML for the working programmer.* Cambridge University Press, 1993.

[76] M.C. Pease. An adaption of the fast Fourier transform for parallel processing. *Journal of the ACM*, 15(2):252–264, April 1968.

[77] Simon L. Peyton Jones and John Lauchbury. Declarative systems architecture: a quantitative approach. EPSRC case for support, Department of Computer Science, University of Glasgow, March 1992.

[78] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*. ACM Press, Sept 91.

[79] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical recipes in Pascal the art of Scientific computing*. Cambridge, England , New York , Cambridge University Press, 1989.

[80] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 2(3):23–49, January 1965.

[81] Walter Rudin. *Principle of Mathematical Analysis*. McGraw-Hill, 1964.

[82] J. Sargeant. United functions and objects, draft language description. Technical report, Computer Science Department, University of Manchester, 1992.

[83] J. Sargeant. United functions and objects, an overview. Technical report, Computer Science Department, University of Manchester, 1993.

[84] K.E. Schauser, D.E. Culler, and S.C. Goldstein. Separation constraint partitioning - a new algorithm for partitioning non-strict programs into sequential threads. In *Principles of Programming Languages*, January 1995.

[85] K.E. Schauser and S.C. Goldstein. How much non-strictness do lenient programs require? In *FPSC '95 SUGPLAN-SIGARCH-WG2.9 Conference on Functional Programming Languages and Computer Architecture*, June 1995.

[86] Stephen K. Skedzielewski, John T. Feo, and Scott M. Denton. *SISAL 90 User's Guide*. Lawrence Livermore National Laboratory, 1995.

[87] Murray R. Spiegel. *Mathematical Handbook of Formulas and Tables*. Schaum's Outline Series in Mathematics. McGraw-Hill Book Company, 1968.

[88] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to programming language theory*. MIT Press, 1977.

[89] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[90] Stephen J. Sullivan and Benjamin G. Zorn. Numerical analysis using nonprocedural paradigms. *ACM Transactions on Mathematical Software*, 21(3):267–298, September 1995.

[91] Simon Thomson. *Haskell, The craft of functional Programming*. Addison-Wesley, 1996.

[92] K.R. Traub. *Implementation of Non-strict functional programming languages*. MIT Press, 1991.

[93] G. Tremblay and G.R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 119–133, April 1995.

[94] D.A. Turner. *The SASL language manual, University of St Andrews*, 1976.

[95] D.A. Turner. Recursion equations as a programming language. *Functional programming and its applications*, pages 1–28, 1982.

[96] D.A. Turner. An overview of Miranda. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 1–16, 1990.

[97] P. Wadler and S. Blott. How to make *ad hoc polymorphism less* ad hoc. In *16th Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[98] Philip Wadler. Monads for functional programming. Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag, Aug 92.

[99] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda calculus.* PhD thesis, Department of Computer Science, 1971.

[100] Roger L. Wainwright and Marian Sexton. A study of sparse matrix representations for solving linear systems in a functional language. *Journal of Functional Programming*, 2(1):61–72, January 92.

[101] Joan R. Westlake. *A Handbook of Numerical Matrix Inversion and Solution of Linear Systems.* John Wiley & Sons INC., 1968.

[102] H.S. Wilf. Matrix inversion by the annihilation of rank. *J. Soc. Indust. Appl. Math.*, 7:149–151, 1959.

[103] H.S. Wilf. Matrix inversion by the method of rank annihilation. In *Mathematical Methods for Digital Computers.* Wiley, 1960.

[104] David S. Wise. Matrix algorithms using quadtrees. Technical Report 357, Computer Science Department, Indiana University, 1992.

[105] D.S. Wise. Matrix algebra and applicative programming. In *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science No. 274. Springer-Verlag, 1987.

[106] D.S. Wise. Undulant block pivoting and integer-preserving matrix inversion. Technical Report 418, Computer Science Department, Indiana University, 1994.

[107] Chris Woodford. *Solving Linear and Non-linear equations.* ELLIS HORWOOD, 1992.

[108] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Frontier Series. ACM Press, 1991.