The Specification of Array-Based Algorithms and the Automated Derivation of Parallel Implementations through Program Transformation

A

THESIS

submitted in fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in the

FACULTY OF SCIENCE

of

THE QUEEN'S UNIVERSITY OF BELFAST

by

Stephen Fitzpatrick B.A. (Oxon) M.Sc.

September 1994

Abstract

It is generally a difficult task to construct efficient implementations of numerical mathematical algorithms for execution on high-performance computer systems. The difficulty arises from the need to express an implementation in a form that reflects the nature of the computer system, rather than a form that reflects the computations performed by the algorithm.

This thesis develops the method of program transformation to derive automatically efficient implementations of algorithms from high-level, machine-independent specifications.

The primary system considered is the AMT DAP array processor, but sequential and vector systems are also considered.

The transformational method is further extended to automatically tailor implementations to use sparse programming techniques.

Acknowledgements

My thanks are due to Dr Terence Harmer for his patience and guidance through the four years this thesis has been in the making; to the members of the Department of Computer Science at QUB for a multitude of stimulating coffee breaks; and to my fellow candidates for their support (and for making my working hours look almost sane).

My thanks also to Dr James Boyle for hosting me at the Mathematics and Computer Science Division of Argonne National Lab., and to the lab. authorities (A.K.A. the University of Chicago and the U.S. Department of Energy) for the opportunity.

Personal thanks to AJ, ST and SS for keeping in touch through moods fair and foul.

The work reported in this thesis was supported by the Department of Education for Northern Ireland through a research studentship.

Contents

List of Figures

Chapter 1

Introduction

The theme of this thesis is the functional specification of numerical mathematical algorithms and the automated derivation through program transformation of efficient implementations for high-performance computer systems.

Numerical mathematical algorithms are algorithms typically based upon linear algebra or discrete approximations. Such algorithms dominate the field of scientific computing and typically have high computational requirements.

A functional specification is a high-level, machine independent definition of an algorithm in a functional programming language (for this thesis, I use a subset of the SML language; Lisp has also been used in related work). Functional programming languages permit an algorithm to be specified in a form close to its mathematical formulation.

An efficient implementation of an algorithm is typically very different from the algorithm's mathematical form: the form of an efficient implementation is typically dictated more by the particular computer system that will execute the implementation, than by the algorithm's mathematical form. This difference, combined with the awkward semantics of most implementation languages, is responsible for many of the problems arising in the construction and development of implementations, especially when an implementation is required for each of a set of computer systems.

Program transformations are rewrite rules, consisting of a pattern text and a replacement text[1]. The transformation system used for this thesis is the TAMPR system (Transformationally Assisted Multiple Program Realization), developed by Boyle at Argonne National Laboratory. TAMPR automatically applies transformations to a program, substituting all sections of the program that match the pattern texts with the corresponding replacement texts.

A derivation is a sequence of transformations, used to achieve radical changes in a program through many simpler changes. This thesis discusses several derivations that convert a functional specification of an algorithm into efficient implementations, each implementation being tailored for a particular computer system.

The primary system considered here is the AMT DAP array processor, programmed using the array-based Fortran dialect, Fortran Plus Enhanced; but I also consider sequential systems and a CRAY vector processor. In addition, much of the work can be used to produce implementations for multi-processor systems.

[1]Transformations are discussed in detail later, but for now it should be noted that the pattern and replacement texts are *not* just character strings; they are program trees defined in a formal grammar.

I also consider extensions to derivations that permit an implementation to be automatically tailored to use special programming techniques that take advantage of special forms of input data (sparse matrices).

1.1 The Problem

Over the last few decades, the computational power of computer systems has increased enormously, but the computational requirements of their users has always kept pace: it is seemingly a trivial matter for researchers to come up with important problems that demand the utmost performance from the most advanced systems. There are always vital algorithms that require maybe weeks or months to execute; as a result, practitioners are seemingly obsessed with efficiency of implementations (perhaps at the expense of the correctness of implementations) so that even minor improvements in performance are seized upon as important advances.

The majority of such algorithms are numerical mathematical algorithms; that is, algorithms which arise from mathematical analysis of problems in physics (e.g. quantum chromodynamics), chemistry (e.g. modelling of chemical interactions from first principles), engineering (e.g. aircraft design), modelling (e.g. of oil reservoirs in geology), and so on. Such algorithms are based upon methods that have been developed over centuries, to provide elegant analytic techniques that use simple but powerful notations and that have highly effective proof techniques (such as structural induction). Thus, many of the algorithms that consume the bulk of computational power on high performance systems are capable of being defined in an elegant manner that facilitates understanding, analysis and manipulation; many of the algorithms are indeed at least partially so defined.

However, to obtain the best performance from most high performance systems, it is usually necessary to construct computer programs at a level that reflects the nature of the system: high level algorithms expressed in mathematical terms must be distorted into low-level manipulations of system components such as storage, communication channels and processors. Such low-level manipulation is graceless, almost incapable of elegant expression or analysis.

Thus, a programmer must usually make the following choice: construct an abstract, modular implementation and pay the price in execution performance; or construct a low-level implementation that is tailored to the implementation system, and pay the price in difficulty of initial construction, in difficulty of maintaining the implementation as the algorithm and implementation techniques evolve, in difficulty of transferring the implementation to other computer systems, and in difficulty in ascertaining that the implementation is a correct implementation of the algorithm

Almost all practitioners in numerical mathematics choose the latter course: the performance criteria are simply too important and too demanding.

1.2 The Proposed Solution

In this thesis, I discuss a technique that aims to obviate the need for choosing between clarity and efficiency by separating what a programmer constructs from what is executed by a computer:

- a programmer constructs a *functional specification* of an algorithm, which is a clear, natural statement of the algorithm;
- he then uses the TAMPR transformation system to *automatically* apply *program transformations* to *derive* an efficient *imperative implementation*.

This separation of specification and implementation permits each to be designed according to its purpose: the specification is simple for a person to understand and analyse; the implementation executes efficiently.

Functional Specifications

A *functional specification* is an algorithmic specification written in a functional programming language.

- Functional programming languages can provide a conceptual basis that is similar to standard mathematics: primarily, algorithms in both mathematics and functional languages are based upon the evaluation of expressions. Functional languages can provide a syntax familiar to mathematicians. A functional specification is thus often a simple rewrite (conforming to a precise, formal notation) of a mathematical definition.
- A functional specification should be designed to be a clear, natural, simple statement of an algorithm. Although written in a programming language, a functional specification is *not* intended to be a practical implementation of an algorithm, so a programmer is free to employ whatever level of abstraction and modularity he feels is required to facilitate understanding. A specification should be independent of any consideration of the eventual implementation system. This independence results in higher-level specifications that are easier to understand than machine-based specifications. It also permits a single specification to be used to produce implementations for multiple systems.

Imperative Implementation

An *imperative implementation* is an implementation of an algorithm in an imperative programming language, usually Fortran or C.

- An imperative implementation for a particular implementation system is intended to be expressed in whatever form is necessary to obtain good execution performance from that system.
- An implementation is constructed for the compilation system, not for human readers, so no concern need be given to modularity or clarity.

Program Derivation

An imperative implementation is constructed from a functional specification by applying a *derivation*. A derivation is a sequence of *program transformations*, which are rewrite rules defined in a formal grammar.

- An implementation is derived from a functional specification by many applications of many transformations. A specification is gradually molded from functional form into imperative form, through several major intermediate forms.
- Each program transformation is typically simple enough for it to be self-evidently correct (i.e. that it preserves the meaning of a specification).
- Program transformations are formal objects and so there exists the possibility of formally proving the correctness of more complex transformations.
- Transformations are applied automatically, so if each transformation is known to be correct (and if the transformation system is trusted), then any number of applications of transformations is known to preserve the meaning of a specification. A derived implementation is thus known to be correct by construction.

- The derivation process itself is formal, so there exists the possibility that the process can be formally analyzed (to prove, for example, that the process terminates, or is complete).

One of the important aspects of the derivational approach is that the document that the programmer constructs (the functional specification) can and should be designed for human beings, while the document that is constructed automatically by derivation (the implementation) should be designed for processing by a compilation system. The functional specification should be simple for the programmer to construct initially and for him to maintain as the algorithm evolves; it should be simple for someone else to understand; and it should be amenable to formal analysis. The imperative implementation should be constructed in a form that produces efficient execution when processed by the compiler for a particular implementation system: that may mean that loops should have certain forms (to permit vectorization), or that array operations should be expressed in a whole-array fashion (for array architectures) — whatever is required by the implementation system.

Several further important aspects of the derivational approach are:

- A single functional specification can serve as the source from which multiple implementations are derived. The implementations may employ different techniques (for example, different data distributions on distributed memory systems) or may be tailored for particular implementation systems (one for sequential architectures, another for vector architectures, a third for array architectures, etc.).

  The derivational approach thus provides a means to construct an entire family of implementations, all automatically produced from the one specification.

- Much of the processing of a specification performed by a derivation is independent of the eventual implementation system. Thus, a derivation for one system is similar in structure to a derivation for another system, and many of the transformations are shared.

- Derivations can be structured so that various *implementation issues* are addressed one by one. For example, one stage of a derivation may optimize specifications by eliminating the modular structure; another may tailor an implementation for the target system; another may optimize storage reuse; another may perform the conversion from functional form to imperative form. Structuring a derivation reduces the intellectual complexity of the implementation process (in the usual divide-and-conquer manner).

- Each implementation issue represents a choice for the programmer, in that he must choose how to tackle the issue and encode that choice as a stage of the derivation; each choice is a transition from a more abstract to a more concrete form, as a specification is gradually molded into an imperative implementation. Structuring a derivation helps highlight the choices, drawing attention to the consequences of a choice for the final implementation.

- A derivation can be used to produce implementations from many specifications. A programmer's time and effort is thus expended in producing a method for implementation, rather than in producing a single implementation.

- As an algorithm evolves, it is a simple manner to change a functional specification to maintain consistency. Further, since derivation is performed automatically, it is relatively straightforward to produce updated implementations for the new specification.

- As implementation techniques evolve, a derivation (or perhaps just individual transformations) can be modified to correspond. The derivation is then re-applied to a specification to obtain an updated implementation.

Program derivation promises a means for a programmer to be able to construct high-level, abstract, readily understood specifications without sacrificing implementation performance. A single functional specification can be used as the source from which multiple efficient implementations are automatically derived by program transformation. Each implementation can be tailored to a particular implementation system to obtain the best performance from that system.

It remains an open question as to what extent this promise can be realized: it is quite possible that human programmers employ techniques that cannot readily be encoded (as transformations), and if so, the derivational approach would not be entirely general. However, I believe that much of what human programmers do when implementing algorithms can indeed be encoded, at least for certain important classes of algorithms. I certainly believe it is worth investigating what can be accomplished using the derivational approach; this thesis is a contribution in that direction.

1.3 Summary of Original Work

The original work reported in this thesis is:

- The development of an Array abstract data type suitable for specifying numerical mathematical algorithms and suitable for automated manipulation (the important functions of the ADT being the array constructor function `generate` and the array reduction function `reduce`).
- The identification of a family of intermediate forms facilitating the optimization of specifications, their tailoring for particular forms of input data and their implementation for multiple architectures.
- The development of derivation stages and transformations to create these intermediate forms.

1.4 Overview of Thesis

The central theme of this thesis is the application of the derivational approach to producing efficient implementations of numerical algorithms for the AMT DAP array processor. In addition, derivations are developed to produce implementations for sequential architectures and for the CRAY vector processor; these derivations are not as fully developed as the DAP derivation, but they are important to lend credence to the claim that implementations can be produced for multiple architectures.

In addition, I consider the tailoring of implementations when the data sets being manipulated are known to be sparse: transformations are added to derivations to optimize storage for such data sets and to reduce the number of computations performed. I consider this tailoring to be an important example of the flexibility of the derivation approach; the structure of derivations permits additional stages to be inserted to tackle particular implementation issues. In this manner, the use of sparse programming techniques is considered merely as another implementation issue, in contrast with traditional approaches which seem to consider the production of an implementation employing sparse programming techniques as being an entirely separate problem from that of producing a standard, ‘dense’ implementation.

The organization of the thesis is:

Chapter 2: Functional Specifications
I discuss in detail the styles typically employed by numerical mathematicians in defining algorithms, assessing the strengths and weaknesses of the styles. In particular, I emphasise that what gives mathematical definitions much of their elegance and ease of manipulation is being based upon expressions.

I then consider the use of a functional programming language for algorithm specifications; functional programming languages provide an *entirely* formal notation lacking in mathematical definitions (which often use natural language to describe important aspects of algorithms), but have the same expression-basis as mathematics and so share the same elegance and ease of manipulation. I discuss two examples of specifications to introduce the syntax of the specification language (a subset of the SML programming language).

Numerical algorithms are based primarily upon array-like data structures, but arrays are not traditionally well-supported in functional programming languages. So I discuss the provision of an array abstract data type.

Chapter 3: Implementing Specifications

Assuming a functional specification of an algorithm, the task then is to produce an efficient implementation. I have stated that the method used for this thesis is to automatically derive implementations, but I first briefly consider problems that must be addressed when implementing specifications and general methods that could be used for implementation, as this helps illustrate the advantages of the derivational approach.

Chapter 4: Derivations and Transformations

I then discuss details of the TAMPR transformation system: how transformations are constructed; how transformations are structured into sequences for application and how sequences of transformations are structured into derivations.

As an example derivation, I consider the conversion of a λ-calculus specification into Fortran77.

Chapter 5: AMT DAP 510 Derivation

The primary implementation system that I consider in this thesis is the DAP array processor. I discuss the hardware architecture and the programming system. I then detail a derivation to implement a functional specification in a DAP dialect of Fortran (Fortran Plus Enhanced).

Chapter 6: Extended Examples

I discuss the derivation of Fortran Plus Enhanced implementations of two examples of reasonably complex algorithms: a Conjugate Gradient algorithm (CG) for approximating the solution of simultaneous equations; and the POT algorithm for approximating the eigensystem of a real, symmetric matrix. Each stage of the derivation for Conjugate Gradient is considered. Only the specification and FPE implementation of POT are considered; the derived Fortran Plus Enhanced implementation of POT is compared with an independent, manually constructed implementation.

Chapter 7: Extending the Derivation for Other Architectures

The derivation that produces FPE implementations is then modified to produce sequential implementations. This derivation is then further modified to produce implementations tailored for the CRAY vector processor. Implementations of CG are discussed.

Chapter 8: Sparse Matrices

An additional stage is introduced into each of the DAP, sequential and CRAY derivations to introduce sparse programming techniques into implementations.

I first discuss the types of sparsity that will be considered: tridiagonal matrices and ‘row-wise’ matrices (which have a the same number of non-zero elements in each row). I then detail the method used for tridiagonal matrices and evaluate a tridiagonal implementation of CG (for sequential, CRAY and DAP). I then evaluate row-wise implementations (details of the row-wise method are given in appendix B).

Chapter 9: Related Work

Chapter 10: Conclusions

Appendix A: The Intermediate Forms
Each major intermediate form produced by the derivations is detailed.

Appendix B: The Sub-derivations
Each stage of the derivations is detailed.

Appendix C:Gram-Schmidt Orthonormalization
An example arising from the POT algorithm: the functional specification and a derived implementation of Gram-Schmidt orthonormalization.

1.5 Notational Conventions

Footnotes are used to add minor points of non-essential information to the body of the text; footnotes are indicated by Arabic numerals[2]. More extensive notes are collected at the end of each chapter and are indicated by Roman numerals. (i) Both footnotes and chapter notes can be ignored on a first reading.

References to other work are indicated by a number in square brackets, e.g. [2].

Mathematical expressions are denoted in italic font: $x + f(y)$.

Verbatim copies of other texts are indicated by a shaded background.

A verbatim copy of some text, usually a mathematical discussion of an algorithm. I am responsible for copying the text, but not for its contents or form.

Verbatim copies will usually contain a reference to the original source at the top of the reproduction.

Sections of programs and transformations are displayed in a fixed-width font: `x := x+1`. Larger sections are usually offset:

```
  DO 1 i=1,n
  A(i) = B(i)
1 CONTINUE
```

```
.sd.
     x+0 ==> x
.sc.
```

1.6 Chapter Notes

i (page 7)
Chapter notes are cross-referenced with the page to which the note refers.

[2]A footnote.

Chapter 2

Functional Specifications

In this chapter, I discuss using a functional programming language as a specification language for numerical mathematical algorithms and discuss array based abstract data types required in such algorithms. The discussion is as follows:

- Typical styles used by numerical mathematicians to define algorithms.
- Numerical mathematical algorithm definitions expressed in a functional programming language.
- The common data structures and their operations used in numerical mathematical operations; providing these data types and operations in a functional programming language.
- A general discussion of various aspects of functional specifications.

2.1 Introduction

The central theme of this thesis is the automated derivation of efficient implementations of algorithms. As implied by this statement, I view 'algorithm' and 'implementation' as different, thought not necessarily distinct, terms:

- In general, an algorithm is a method or procedure for solving a problem. In numerical mathematics, an algorithm is a method for computing some value ('the answer' to some problem).
- In general, an implementation of an algorithm is a means of causing some system to carry out that procedure. For this thesis, the 'system' is always a computer; an implementation is thus ultimately a sequence of machine level instructions, but for this thesis an implementation is a program expressed in some dialect of the Fortran programming language.

Thus, in general, I view an algorithm as a quite abstract entity and an implementation as a concrete entity designed to direct the execution of a computer. In addition, with present day hardware and compiler technology, a *practical* implementation of a numerical algorithm must generally be designed for:

- a specific computational architecture — e.g. sequential, parallel, vector or array;
- a specific computer system — e.g. CRAY Y-MP, AMT DAP or the Connection Machine;
- perhaps even for a specific software environment, either because there are several language systems (providing considerably different programming models) available on a given computer system — e.g. for Fortran Plus or Fortran Plus Enhanced (on the AMT DAP) — or because fine tuning of a program is required to obtain the best performance from a particular version of a compilation system (e.g. the optimizations performed by a vectorizing compiler may vary from one release of the compiler to the next).

Indeed, often a given algorithm has multiple implementations, each implementation being tailored for a specific system. Each implementation is a particular realization of the algorithm; the algorithm is an abstract entity, transcending particular computer systems, and perhaps computer systems in general.(i)

The theme of this thesis is the automated derivation of such highly tailored implementations of abstract algorithms. Clearly, any form of formal or automated manipulation, such as derivation, requires something concrete upon which to work: that is, the derivation of an implementation of an algorithm requires some definitive statement of the algorithm, or *algorithm definition*. The nature or character of an algorithm definition will obviously greatly affect the derivation process, so in this chapter I discuss at some length how algorithms can be defined. The algorithms considered in this thesis are all drawn from numerical mathematics, so I first discuss how numerical mathematicians typically define algorithms, and then discuss the form of definition used for this thesis.

2.2 Mathematical Definitions of Algorithms

Algorithm definitions written by mathematicians typically fall into three categories:

- High level analyses, presented mainly in English, with various expressions that are to be evaluated by the algorithm.
- Low level definitions, presented in an imperative programming language such as Fortran or Pascal.

- Formal high level definitions, presented in a mix of English text and expressions but in a more formal style than high level analyses.

2.2.1 High Level Analyses

In a high level analysis, an author gives an informal description in English (or some other natural language) of the process carried out by an algorithm, intermixed with expressions that the algorithm calculates at various stages. I discuss two examples of high level analyses:

Example 2.2.1-a: *Gaussian Elimination*

Gaussian Elimination is used to solve systems of equations of the form

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
\ldots & \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}$$

Such a system of equations can be represented in matrix form as $AX = B$ where A is the matrix of the coefficients a_{ij}, X is the column vector of the x_i and B is the column vector of the b_i. If we have m such systems in which A is the same, then the systems can be solved simultaneously by extending X and B to be $n \times m$ matrices.

An analysis of Gaussian Elimination is reproduced as figure 2.1.

[61] p 100

Consider Gaussian elimination for the matrix equation **AX = B** where **A** and **B** are of order $n \times n$ and $n \times m$ respectively, and whose solution X is of order $n \times m$. Table 1.3 [omitted] illustrates the hand solution procedure without pivot selection. It can be seen that the main part of the solution process (called the *reduction*) involves making a controlled series of modifications to the elements of **A** and **B** until **A** has been converted into upper triangular form. Let $a_{ij}^{(k)}$ be the coefficient in the i, j position at the k-th reduction step where $i \geq k$ and $j \geq k$, and let $b_{ij}^{(k)}$ represent the element b_{ij} at this stage where $i \geq k$. The k-th step in the elimination can be represented algebraically by

$$\begin{aligned}
a_{ik}^{(k+1)} &= 0 && (k < i \leq n) && (4.1a) \\
a_{ij}^{(k+1)} &= a_{ij}^{(k)} - \frac{a_{ij}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}} && (k < i \leq n, k < j \leq n) && (4.1b) \\
b_{ij}^{(k+1)} &= b_{ik}^{(k)} - \frac{a_{ik}^{(k)} b_{kj}^{(k)}}{a_{kk}^{(k)}} && (k < i \leq n, 1 < j \leq m) && (4.1c)
\end{aligned}$$

On a computer the elements evaluated according to equations (4.1) may overwrite their predecessors at stage k, and hence the reduction can be carried out in the array storage which initially holds the matrices **A** and **B**. Since the eliminated elements play no part in the subsequent elimination there is no need to implement equation (4.1a).

Figure 2.1: Analysis of Gaussian Elimination

Example 2.2.1-b: *Cellular Automata/Characteristics Algorithm for Hyperbolic Partial Differential Equations*

The solutions of hyperbolic partial differential equations are often required in models of fluid flow, where the fluid is subject to conservation laws. Two popular techniques for the calculation of solutions are Cellular Automata and the Method of Characteristics. The former can be viewed as a discrete approximation to molecular dynamics; the latter uses properties of conservation laws to calculate the motion and behaviour of certain characteristics (or invariants) of the fluid, from which the behaviour of the fluid itself can be derived. The algorithm considered here is a combination of these techniques. (For brevity, I will normally refer to this algorithm as "the Cellular Automata algorithm".)

An analysis of the Cellular Automata algorithm is reproduced as figure 2.2.

Discussion of High Level Analyses

The following are general characteristics of high level definitions:

- In general, the discussion proceeds from the mathematical or physical problem, through an outline of the method used to solve the problem, to details of interesting or novel aspects of the method. This "top down" approach has the advantage of introducing the algorithm through, and relating the algorithm to, concepts with which the reader is probably already familiar.

  For example, the Cellular Automata analysis can be summarized as follows:

  - a discussion of the physical problem modelled by the algorithm;
  - an outline of the algorithm: the authors indicate that they use an iterative technique and give an almost anthropomorphic description ("… we follow the characteristics …") of what to do on each iteration;
  - what the algorithm calculates ("… determine the field … compute u(x,t) … ") and how the calculation is performed ("Our approach to the solution of problem A … "), defined using quantities associated with the problem;
  - details of the method and a description of the data manipulated by the algorithm.
- Often the algorithm is partially described by providing a worked example for a small problem with the statement that the process for larger problems follows by extension.
- Often a description consists of a basic, simplified algorithm, followed by extensions or modifications to the algorithm. There may not be a single concise definition of the final algorithm; indeed, the modifications may be applicable under different circumstances, so the description is actually of a family of algorithms.
- It is usually difficult to extract a concise definition of the algorithm, as this requires sifting the discussion for details and taking into account any extensions described. If the reader is not familiar with the problem area (he is, say, a computer scientist rather than a physicist), this task may prove difficult.

Some disadvantages of high level analyses are:

- Not all details of the algorithm may be present in the discussion, since the discussion may be designed to highlight the most important or novel aspects of the algorithm. Indeed, it is common for a discussion to gloss over operations that would be major sections of an implementation, either because the operations are well known or because the method used to perform an operation is irrelevant in the context of the main algorithm under discussion.

[37]

[Discussion of the physical problem omitted.]

Our method of computation is based on the use of the method of characteristics to compute the entropic solution of P as follows:

- We start with a discretization of the initial conditions.
- For each time step, we follow the characteristics, some of which may intersect and some of which do not.
- Those pairs of characteristics that intersect, causing shocks, are removed from further time steps.

. . .

Our approach to solving HPDEs with the method of characteristics may be divided into two parts. At each time step we

A) determine the field of characteristics $x(x, t)$, and

B) compute $u(x, t)$ for a given t.

The computation of A for each characteristic may be done in parallel, but may be difficult when the characteristics intersect. For a single conservation law, the computation of B is trivial, and so we are concerned here only with solving A.

. . .

Our approach to the solution of Problem A is to discretize the space direction into a one dimensional grid of cells, and then have each processor correspond to a particular cell. We begin, by computing a discretization of the initial conditions, Specifically, we define a grid in the (x, t) plane. We then have a set of rectangular cells with the bottom side of the cells at $t = 0$. The horizontal side is the space interval, and the vertical side is the time interval. The ratio between the two sides is limited by the CFL condition [reference to literature for details of CFL condition]. Each characteristic begins in the center of the cell. Our parallel algorithm works by assigning a processor to each of the cells in the discretization.

. . .

For the first time step, each cell contains the characteristic issued from within that cell. At each time step, we compute the characteristic's advance:

$$x \leftarrow x + f'(u)\Delta t$$

As characteristics advance, they may continue in the cell they are in or exit from the cell. New characteristics may also enter a cell. During a time step, cells communicate only with their east and west neighbors to determine which new characteristics, if any, enter their cell at the next time step. The result of this computation for a given time step is a set of cells, each of which contains the characteristic(s) in the cell, if any.

Our algorithm works with two types of data:

- Numerical data describing the characteristic [details omitted]
- Boolean data describing the state of each cell [details omitted]

. . .

Analyzing the different possibilities for three neighbouring cells, we see that a shock occurs if:

- either two or more characteristics intersect in the same cell, or
- a pair of characteristics each exit their cells and cross.

Figure 2.2: Analysis of Cellular Automata algorithm

- Aspects of the algorithm may be defined *solely* in terms of concepts from the problem. For example, in the cellular automata analysis, the authors state how to initialize the data: "For the first time step, each cell contains the characteristic issued from within that cell". They do not expand on how this relates to the data manipulated by the algorithm; if this information is required, it is necessary to examine an imperative implementation of the algorithm provided by the authors.
- The clarity of the analysis can be highly dependent upon the author's command of language.
- A natural language description of a process can be very verbose.
- As analyses are informal, they are amenable to neither formal proof techniques nor machine processing.

Some advantages of high-level analyses are:

- An author can employ the usual teaching aids (diagrams, examples, analogies, judicious simplifications) to facilitate understanding an algorithm.
- Those sections of the definition that employ formal notation are based on evaluating expressions; for example, the equations defining the elimination step in the Gaussian Elimination definition. The importance of this is that expressions have a very simple semantic basis: the meaning of an expression is simply its value. The expression-based sections of the definition are thus usually simple to understand and to reason about.
- Generally, the definition does not detail the manipulation of storage (the operations are *referentially transparent*): both of the above algorithms are based on the repeated application of some operation, but each iteration creates a new set of values rather than updating a set of state variables. The introduction of state would greatly complicate a definition since, for example, the value of an expression could depend on other expressions evaluated, on how many times an expression is evaluated or on the order of evaluation.

  However, sometimes an author does address the use (and reuse) of storage (e.g. Gaussian Elimination: "the elements evaluated ... may overwrite their predecessors") because in-place updating can be essential for the efficient execution of an algorithm.

In summary:

- a high-level analysis is more a discussion about an algorithm than a definition of the algorithm;
- the formal sections of the discussion are expression-based and so have simple semantics;
- the informality of most of an analysis precludes automatic manipulation.

2.2.2 Low Level Definition

Many authors who provide a high level analysis as described above, follow the analysis with an implementation of the algorithm in an imperative programming language such as Fortran or Pascal, or some ad-hoc pseudo-language. For example, the Cellular Automata and Gaussian Elimination analyses given above were followed by imperative implementations, reproduced as figures 2.3 and 2.4.

[37]

```
processor$_i$
  { Initialization }
   read u
  $sign \leftarrow$ sign of $f'(u)$ { $sign = 0$ if $f'(u) < 0$, else $sign = 1$ }
  $state \leftarrow 1,\ x \leftarrow tx \leftarrow 0.5,\ tsign \leftarrow sign$
  $slope \leftarrow |\ f'(u)\ |$
  $outlft \leftarrow outrht \leftarrow inlft \leftarrow inrht \leftarrow 0$
  foreach $timestep$ do { Main loop, for number of time steps }
    if ($state\ ==\ 1$) then { for cells that contain a characteristic }
      $tu \leftarrow u,\ tslope \leftarrow slope$
      $x \leftarrow x + slope$
      if ($x > 1$) then { Characteristic has left the cell }
        $state \leftarrow 0,\ u \leftarrow empty$
        $tsign \leftarrow sign,\ tx \leftarrow x - 1,\ x \leftarrow 0.0$
        if ($sign\ ==\ 0$) then $outlft \leftarrow 1$ $endif$
        if ($sign\ ==\ 1$) then $outrht \leftarrow 1$ $endif$
      endif
    endif
    if ($east(outlft)\ ==\ 1$) then $inrht \leftarrow 1$ endif
    if ($west(outrht)\ ==\ 1$) then $inlft \leftarrow 1$ endif
    $shock \leftarrow state + inlft + inrht$ { calculate shock }
    if ($shock > 1$) then { shock, more than one characteristic in cell }
      $state \leftarrow 0$, $u \leftarrow empty$
    elseif ($inlft\ ==\ 1\ and\ outlft\ ==\ 1$) or
      ($inrht\ ==\ 1\ and\ outrht\ ==\ 1$) then { shock, crossing characteristics }
      $state \leftarrow 0$, $u \leftarrow empty$
    else {no shock}
      if ($inlft\ ==\ 1$) then
        $x \leftarrow west(tx),\ slope \leftarrow west(tslope),\ sign \leftarrow west(tsign)$
        $u \leftarrow west(tu),\ state \leftarrow 1$
      endif
      if ($inrht\ ==\ 1$) then
        $x \leftarrow east(tx),\ slope \leftarrow east(tslope),\ sign \leftarrow east(tsign)$
        $u \leftarrow east(tu),\ state \leftarrow 1$
      endif
    endif
  $outlft \leftarrow ourrht \leftarrow inlft \leftarrow inrht \leftarrow 0$
  end foreach
```

Figure 2.3: Implementation of Cellular Automata algorithm

[61] p 101

```
  KK = K+1
  DO 1 I = KK,N
  X = A(I,K)
  IF (X .EQ. 0.0) GO TO 1
  X = X/A(K,K)
  DO 2 J = KK,N
2 A(I,J) = A(I,J)-A(K,J)*X
  DO 1 J = 1,M
  B(I,J) = B(I,J)-B(K,J)*X
1 CONTINUE
```

Figure 2.4: Implementation of the reduction step of Gaussian Elimination

There are several reasons why an author may provide an imperative implementation:

- The implementation is usually simple to use in a program, so the reader is saved the task of implementing the algorithm himself.
- There may be aspects of the implementation itself that are pertinent to the author's discussion. For example, in both of the above algorithms, the authors were concerned with being able to update variables on each iteration; details such as the use of temporary variables and the order of execution are thus important.
- The authors may wish to emphasize the implementation of an algorithm for a particular computational model. For example, the implementation of the Cellular Automata algorithm was designed for a data parallel model.
- If the author is reporting on the execution characteristics of an algorithm, it is important to indicate what was actually executed.

However, here I am considering the suitability of such low level implementation as algorithm definitions, as discussed below.

Discussion of Low-level Definitions

There is one major advantage for low-level definitions when considered as an algorithm definition:

- As the definition is written in a programming language, the definition is entirely formal. The existence of compilers for such languages demonstrates that such definitions can be processed automatically.

However, there are several major disadvantages:

- Widely used imperative languages tend to have unwieldy semantics, inhibiting analysis of and manipulation of definitions.
- Imperative programming tends to require the programmer to think in terms of the machine he is programming, rather than in terms of the data the algorithm is manipulating. For example, to subtract one (scaled) column of a matrix from another in the Fortran implementation of Gaussian Elimination, the programmer must use a loop which processes one element at a time, reflecting the sequential execution order and single word operations of typical von Neumann computers.

  Such machine-based thinking not only inhibits the initial development of the definition, it also inhibits understanding of, analysis of and manipulation of (e.g. for parallelization) the definition. Some languages do provide abstraction mechanisms that permit the programmer to ignore, to a greater or lesser extent, the details of the machine but programmers may avoid using such abstraction mechanisms as their use generally degrades execution performance.
- Imperative programs generally use subroutines to decompose complex operations (and software libraries almost always provide operations as subroutines). Unfortunately, subroutines have wide interfaces: that is, the inputs to, the outputs from and the effects of a subroutine cannot easily be localized. For example, data is often provided to subroutines through global variables and subroutines often return values or store state information by altering variables that are passed to them as arguments or by altering global variables. The total effect of a subroutine application may include altering any of the variables in its scope.

  Further, subroutines are not *combining forms*: the effects of one subroutine cannot be combined with the effects of another by direct composition (i.e. by directly applying one subroutine to another). Their combination must be mediated by variables in which the effects of the individual subroutines are accumulated one at a time.

In contrast, pure functions can be directly composed. For example, compare the combination of matrix multiplication and addition in functional form and in subroutine form:

$A * B + C * D \quad \rightarrow$

```
CALL MMMULT(A,B,T1)
CALL MMMULT(C,D,T2)
CALL MMADD(T1,T2,T3)
```

Combining subroutines through sequencing has several disadvantages:

- The subroutine sequence bears little resemblance to the mathematical expression: the structure of the expression is lost. This inhibits understanding the computations being performed by the subroutine sequence and inhibits maintaining the sequence.
- The effects produced by the sequence are not localized:
  - The three subroutine invocations may be separated by any number of program statements, any of which may alter the value of the 'mediating' variables `T1` and `T2`.
  - Further, suppose one of the intervening statements is an application of some subroutine `S`. On examining the implementation, the only way to determine whether the application of S makes use of the values assigned to `T1` and `T2` is to examine the definition of S, and the definitions of any subroutines called by S, and the definitions of any subroutines called by those subroutines, and so on.
  - The mediating variables retain their values beyond the point in the implementation where the values are required. Thus, on examining an implementation, there is again the possibility that program statements following this point make use of the values.

It may be argued that the careful arrangement of the scope of variables can ameliorate the above problems. However, the programming language may not permit or may discourage the use of local scope: for example, Fortran does not permit the size of arrays to be determined during program execution, so a common programming technique is to allocate a large, global 'workspace' array from which portions are allocated as needed. Each and every statement in the program can alter the contents of any portion of the array in a completely arbitrary manner.

As a result of the above, an imperative implementation of an algorithm is a poor definition of the algorithm as it bears little resemblance to any extant high level analysis of the algorithm and is difficult to understand and to manipulate.

2.2.3 Formal High Level Definitions

In a formal, high level definition, an author uses natural language to prescribe the operations that are to be performed to calculate a required value. In contrast to a high level analysis, the English is formal to the extent that there is little room for ambiguity, but in contrast to a low level definition, the operations prescribed are drawn from the problem domain of the algorithm rather than being limited to the operations provided by programming languages such as Fortran; for example, in a linear algebra algorithm the operations may be matrix addition or multiplication, rather than single-array-element assignments. Examples of high level definitions follow.

Example 2.2.3-a: *Jacobi's Algorithm*

Jacobi's algorithm, figure 2.5, is used to solve systems of linear equations (see section 2.2.1).

[11] pp 130–131

Let A be an $n \times n$ matrix, let b be a vector in $\Re^n$, and consider the system of linear equations

$$Ax = b$$

where x is an unknown vector to be determined.

...

Starting with some initial vector $x(0) \in \Re^n$, evaluate $x(t)$, $t = 1, 2, \ldots$, using the iteration

$$x_i(t+1) = -\frac{1}{a_{ii}} \left[\sum_{j \neq i} a_{ij} x_j(t) - b_i \right]$$

Figure 2.5: Definition of Jacobi's algorithm

Example 2.2.3-b: *Conjugate Gradient*

The Conjugate Gradient algorithm, figure 2.6 is used to solve systems of linear equations (see section 2.2.1).

[80] p152

```
To solve $Ax = b$, where $A$ is a positive definite symmetric $n \times n$ matrix:
    Set an initial approximation vector $x_0$,
    calculate the initial residual $r_0 = b - Ax_0$,
    set the initial search direction $p_0 = r_0$;
    then, for $i = 0, 1, \ldots$,
      (a) calculate the coefficient $\alpha_i = p_i^T r_i / p_i^T A p_i$,
      (b) set the new estimate $x_{i+1} = x_i + \alpha_i p_i$,
      (c) evaluate the new residual $r_{i+1} = r_i - \alpha_i A p_i$,
      (d) calculate the coefficient $\beta_i = -r_{i+1} A p_i / p_i^T A p_i$,
      (e) determine the new direction $p_{i+1} = r_{i+1} + b_i p_i$,
    continue until either $r_i$ or $p_i$ is zero.
```

Figure 2.6: Definition of the Conjugate Gradient algorithm

Example 2.2.3-c: *POT*

The POT (Parallel Orthogonal Transformations) algorithm, figure 2.7, is used to calculate the eigenvalues and eigenvectors of real, symmetric matrices. An eigenvector/eigenvalue pair of a matrix A satisfy the *characteristic equation* $As = \lambda s$ where s is the eigenvector and λ the corresponding eigenvalue. In general, if A is of order N, there are N such pairs, so the characteristic equation for the full eigensystem can be written $AS = S\Lambda$ where S is a matrix of order N and Λ is a diagonal matrix also of order N; the columns of S are the eigenvectors of A and the diagonal elements of Λ are the corresponding eigenvalues.

Discussion of Formal High Level Definitions

The advantages of a formal high level definition are:

[97]

The POT algorithm for the computation of the eigensystem of a real symmetric matrix A, of order N, consists essentially of the following iterative cycle. Given an $N * N$ matrix R of orthonormal approximations to the eigenvectors of A:

1. Form a transformed matrix B, of order N, using the sequence of algebraic operations:

$$V := A * R, B := R^T * V$$

2. Construct a transformation matrix T, of order N, whose columns form an approximation to the set of right eigenvectors of the symmetric matrix B. This construction is described in [reference omitted].
3. Compute a new set of eigenvector approximations R, which is represented in matrix form by the algebraic matrix product

$$R := V * T.$$

4. Orthonormalise the set of vectors R.

The cycle ceases whenever the off-diagonal components of B are zero to a predetermined precision. The diagonal of B then yields the required eigenvalues and the columns of the final matrix R yield the required eigenvectors.

Figure 2.7: Definition of the POT algorithm

- Definitions are based upon operations that are natural to the problem domain. (In the above examples, matrix and vector operations.) This facilitates understanding and reasoning about the definition by invoking any body of results that have been established for the problem domain.
- The definition is not based on any particular computational model. (Alternatively, the definition is based on a computational model that reflects the nature of the algorithm rather than the nature of any particular physical computer system.)
- Definitions are expression based: that is, computation proceeds through the evaluation of expressions (though natural language is used to direct aspects of evaluation). This basis has several advantages:
  - An expression can be viewed as having a very simple meaning: its value — for example, the meaning of the expression $5 + 3$ is simply 8.(ii)
  - Expressions are referentially transparent: that is, in a given context of name bindings, two expressions that look the same have the same meaning/value. This property may seem rather obvious, but it does not hold in state based definitions (such as the low level imperative definitions discussed earlier).
  - The order in which the sub-expressions of an expression are evaluated does not affect the value of the expression. For example, when evaluating the expression $f(g(x), h(y))$, the applications of g and h may be performed in any order, including in parallel.
- One method for decomposing complex operations is to bind an expression to a name; for example, the binding of `A*R` to `V` in POT. This decomposition method has very simple semantics: the name is simply an abbreviation for the expression — occurrences of the name can be replaced by the expression (or by the value of the expression).

- A second method for decomposition is to define functions. A function definition has a simple intuitive meaning, as a parameterized abbreviation for an expression. For example, given the definition $f(x) = x + 2$, the meaning of the application $f(3)$ is the expression $3 + 2$.(iii) (The example algorithms do not explicitly define functions, but they assume standard functions such as vector addition, matrix-vector multiplication, and so on.)

  Functional decomposition has several advantages as a decomposition technique:

  - Functions have narrow interfaces: (a) the factors that influence the value computed by a function are explicitly passed to the function as parameters;(iv) (b) the value computed by a function is returned directly by the function (rather than, say, through global variables); (c) the application of a function *cannot* have any other 'effects', since there is no concept such as a global state that could manifest such effects.
  - As a consequence of a narrow interface, the definition of a function is isolated from its applications. Thus, it is possible to analyze or reason about a function definition with no regard to any other factors such as the context in which it is to be applied.
  - Applications of a function are isolated from the definition of the function. It may be possible to reason about an expression involving an application of a function based solely on the properties of the value computed by the function, with no regard to how the computation was performed. (e.g. the value computed may have the property that it is the solution to a set of simultaneous equations. This property may be useful regardless of which function was used to solve the equations.)

    Clearly, any decomposition technique provides isolation, but the narrow interface of function definitions enhances the degree of isolation, since the only possible 'effect' of a function application is to return a value.
  - The arguments to a function are expressions and a function application is itself an expression. This uniformity allows traditional recursive proof techniques to be employed simply.[1] It also maintains a simple conceptual basis.
  - A further consequence of uniformity is that functions are combining forms: the effects of one function f can be combined with the effects of another g by directly combining the functions, e.g. as $f(g(x))$ or $f \circ g$.[2]

There are two main disadvantages in using high-level definitions to define algorithms:

- Someone who wishes to use the algorithm must undertake the task of implementing the definition himself.
- Implementing the definition may not be straightforward if, as is usually the case with the type of algorithm considered here, the implementation language is an imperative language.

In summary, high-level definitions make good algorithm definitions (though are perhaps not ideal for someone who requires a pre-built implementation).

[1]In contrast, the arguments to a subroutine in state-based languages are expressions, but the subroutine application is itself a statement, a different class of entity.

[2]Contrast this with the required use of variables in state-based definitions to carry the effects of one subroutine to another, with the resulting non-locality.

2.2.4 Conclusions

- A high level analysis can provide a good basis for understanding an algorithm, but lack of formality may lead to ambiguity and prohibit automatic processing.
- A low level definition in an imperative programming language may be formal but is generally unsuitable as an algorithm definition as such languages usually have complex semantics, are normally based on a particular programming model and are difficult to understand and to manipulate.
- A formal high level definition provides precision, has simple semantics and avoids irrelevant details. However, lack of *complete* formality prohibits automatic processing.

2.3 Algorithm Definitions in a Functional Programming Language

There are three main reasons for writing a definition of an algorithm:

- The definition is the basis of practical implementations of the algorithm: any implementation of the algorithm must satisfy the definition and derivations of implementations of the algorithm begin with the definition.
- A person may read the definition to assist his understanding of the algorithm.
- The definition may be analysed to investigate the properties of the algorithm.

If a single document is to be useful for the above purposes, it should possess certain characteristics:

- The definition should be formal to avoid ambiguity and to allow formal manipulation (in particular, to allow automated processing).
- The semantic basis of the definition should be simple, as this assists understanding and (automated) manipulation of the definition.
- The definition should decompose complex operations to reduce the intellectual complexity of the algorithm.
- The definition should be expressed in manner that is close to the problem domain of the algorithm, as this assists understanding by relating the algorithm to the physical or mathematical problem and by using operations with which the reader is probably already familiar.
- The definition should not be biased toward any particular computational architecture as any such bias tends to greatly impede implementation for other architectures.

Of the styles of definition discussed in the previous section, high level analysis can be discounted for the above purposes as its almost complete informality virtually precludes formal manipulation. Low level definitions in imperative programming languages are also ill suited because they tend to assume a very specific computational architecture, more closely related to computer hardware systems than numerical mathematics; they use a poor decomposition method; and they have unwieldy semantics.

High level formal definitions are partly suited: the expression based sections of the definitions have simple semantics, good decomposition techniques, are closely related to numerical mathematics and are not based upon any particular implementation system. Formal definitions do though employ natural language to describe important aspects of algorithms, and so are not entirely suited.

What is required of a definition language is that is possess the above mentioned *desiderata* of high-level definitions, while also being entirely formal. These are precisely the virtues usually claimed for functional programming languages:

- The central concept in functional languages is that computation is denoted using expressions, providing a simple semantic basis.
- The primary means of decomposing operations is to introduce pure functions.
- Functional programming languages support the construction of programs in an abstract, machine-independent manner.
- The expression-basis of functional languages makes them conceptually similar to mathematics. In addition, many modern functional languages allow a syntax that is close to traditional mathematical syntax.
- Functional programming languages are entirely formal.

Thus, the definition language that I use in this thesis is a functional programming language: a pure[3] subset of the Standard Meta-Language (SML [99]) functional programming language. The Lisp language was used as the definition language in earlier versions of this work and is still used for related work. Ideally, a programmer should be relatively free to use whatever functional language he wishes; it should not be difficult to extend the work reported here to other functional programming languages such as Miranda or Haskell.

In addition to simple semantics and naturalness of expression (for numerical algorithms), functional programming languages have the advantage as definition languages of being executable: this can be of use in constructing definitions and in developing algorithms, for ‘rapid prototyping’: an algorithm designer can freely experiment with his algorithm without devoting hours (or days) to the minutiae of an imperative implementation, with all the potential that has for making implementation mistakes which could easily invalidate whatever conclusions the designer draws from his experiments.

2.4 An Example Functional Definition: POT

In this section, I introduce the definition language through several example definitions. The first example is the functional definitions of the POT algorithm: the mathematical definition is repeated as figure 2.8; the functional definition is figure 2.9.

This example exhibits many of the main syntactic features of the definition language:

- Function definitions are introduced by the keyword `fun`, so the above definition defines two functions, `POT` and `POTstep`.
- The type of an identifier follows the identifier, separated from it by a colon. For example, the first formal argument to `POT` is `A`, a matrix with real elements.
- The type following the formal argument list in a function definition is the type of an application of the function (the ‘return’ type); so an application of `POT` has the type `(real matrix*real matrix)` — that is, a pair of real matrices. The type of the function itself (rather than an application of the function) is `real matrix->real matrix*real matrix`.
- The notation

[3]SML and many other functional languages provide some constructs that are based upon state manipulation, as are common in imperative languages. In this work, I avoid all such constructs to maintain a simpler semantic basis.

[97]

The POT algorithm for the computation of the eigensystem of a real symmetric matrix A, of order N, consists essentially of the following iterative cycle. Given an $N * N$ matrix R of orthonormal approximations to the eigenvectors of A:

1. Form a transformed matrix B, of order N, using the sequence of algebraic operations:

$$V := A * R, B := R^T * V.$$

2. Construct a transformation matrix T, of order N, whose columns form an approximation to the set of right eigenvectors of the symmetric matrix B. This construction is described in [8].
3. Compute a new set of eigenvector approximations R, which is represented in matrix form by the algebraic matrix product

$$R := V * T.$$

4. Orthonormalise the set of vectors R.

The cycle ceases whenever the off-diagonal components of B are zero to a predetermined precision. The diagonal of B then yields the required eigenvalues and the columns of the final matrix R yield the required eigenvectors.

Figure 2.8: Definition of the POT algorithm

```
fun POTstep(A:real matrix,R:real matrix)
    :(real matrix*real matrix)
  = let
      val V:real matrix = A*R
      val B:real matrix = transpose(R)*V
      val T:real matrix = transform(B)
      val R′:real matrix = V*T
    in
      (A,orthonormalise(R′))
    end

fun POT(A:real matrix):(real matrix*real matrix)
  = iterate(POTstep,(A,UnitMatrix(shape(A))),IsOk)
```

Figure 2.9: Functional definition of the POT algorithm

`let` *definitions* `in` *expression* ... `end`

denotes a *local expression*. Local expressions are used to introduce definitions that are valid only within the local expression itself. For example, `POTstep` defines the four values `V`, `B`, `T` and `R′`. The value of a local expression is the value of the *body expression* occurring after the `in` keyword: `(A,orthonormalise(R′))`.

- Common arithmetic operations are normally denoted using infix or prefix operators. Many operators are overloaded: that is, the one operator is used to indicate a set of related functions (such as integer addition, real addition, matrix addition, etc.) and the language system uses type information to resolve the overloading (i.e. to determine which component function should be applied in a given expression).

  Overloading provides the programmer with a much simpler interface to common operations; for example, the programmer specifies "addition" and the language system decides whether he means integer addition, real addition, matrix addition, and so on.[4] Overloading is a quite natural tool for definitions, since people generally infer meaning based upon context.

- The multiplication operator, `*`, when applied to vectors and matrices, indicates algebraic multiplication rather than elementwise multiplication.

- The expression `(A,orthonormalise(R′))` is a pair, or a 2-tuple, of real matrices. Components of tuples are extracted using the prefix `#n` operators, where n is an integer. For example, `#1(A,orthonormalise(R′))` is `A`.

Comparing the mathematical definition of POT with the functional definition:

- The mathematical definition introduces a name, A, for the matrix under consideration. While it would be possible to introduce a corresponding global name in the functional definition, it is normally considered better style to define a formal argument for each function, increasing the modularity of the definition.

- The mathematical definition states that the matrix is real, symmetric and of order N. In general, there are various properties of the data sets an algorithm manipulates that may be useful in a definition. In the above example, the type associated with matrix identifiers provides only the information that the elements are real numbers.[5] Other information can be introduced into a definition by assertions or by program transformation.

- The mathematical definition states that the algorithm employs an "iterative cycle" and then defines

  i. what is to be done on each iteration;

  ii. an initial value which which to begin the cycle;

  iii. a terminating condition.

  The functional definition employs the function `iterate` to indicate repetition. The arguments to `iterate` are:

  i. the function, `POTstep`, that is to be repeatedly applied;

  ii. the initial value `(A,UnitMatrix(shape(A)))`;

[4]Widespread use of overloading does require a programmer to be more explicit with types, though, since overloading inhibits the ability of the type system to infer types.

[5]Including the size of a matrix in the type information would require dynamic typing, but SML is statically typed.

iii. a boolean function, `IsOK`, which is applied after each iteration and which determines whether another iteration is to be applied.

- In the mathematical definition, each iteration consists of the calculation of the four matrices V,B,T and R and the orthonormalization of R. In the functional definition, the iteration function POTstep calculates these four matrices (as local values) and then performs the orthonormalization (by applying the function `orthonormalise`).

The functional definition is essentially nothing more than the mathematical definition cast in an entirely formal notation.

2.4.1 POT Example Continued

To illustrate further the definition language, and some of the functions commonly used in definitions, I here discuss the functional definition of the construction of the transformation matrix T used by the POT algorithm (the construction of T is mentioned but not defined in step 2 of the definition; it appears as an application of the function `transform` in the functional specification of POT). Figure 2.10 is the mathematical definition of the transformation matrix; figure 2.11 is the functional definition. Note that in the mathematical definition, the matrix B is a component of a sequence of matrices, hence the subscript k in some expressions.

This example introduces some further syntax:

- The version of SML used for this thesis does not provide arrays as a primitive type, so there is no in-built notation for indexing arrays. In order to have a syntax reasonably close to standard mathematical notation, I use the form `M@[i,j]`, read as "M at (position) ij".
- The notation

  ```
  fn(arguments)=>expression
  ```

  defines a function expression, similar to a λ-expression in the λ-calculus. Function prototypes are useful when a function expression is to be used only once, to avoid introducing a named definition.
- The unary, prefix operator for negation is 'tilda' (~).

This example also introduces some functions that I have defined to permit operations on array-like data structures, including vectors and matrices:

- The function `shape` is applied to an array, returning a list of the dimensions of the array (as an instance of the `shape` type).
- The function `generate` is used to define arrays. The first argument specifies the dimensions of an array; in the `transform` function, the dimensions are specified by applying the function `shape` to the argument matrix `B`, so the dimensions of the result matrix are the same as the dimensions of `B`. This information is not explicit in the mathematical definition.
- The second argument to `generate` is a function which specifies the elements — the value of an element is given by applying the function to the index for that element. In `transform`, this function performs a simple case analysis in the same manner as the mathematical definition, applying the local function `calculate`, which is an abstraction of the expression given in the mathematical definition.

These array functions are explained in detail later.

[98]

The function $transform$ generates from its real symmetric argument, B_k, a non-singular orthogonal transformation matrix T_k, whose elements are given as follows:

$$\begin{aligned} t(i,j) &= 2b(i,j)/[d(i,j) + sign(d(i,j))\sqrt{}(d(i,j)^2 + 4b(i,j)^2)] && i > j \\ t(i,i) &= 1 && \\ t(j,i) &= -t(i,j) && i < j \end{aligned}$$

where

$$d(i,j) = b(j,j) - b(i,i)$$

and where, for convenience, the subscript k has been ignored.

Figure 2.10: Mathematical definition of $transform$ function

```
fun transform(B:real matrix):real matrix
  = let
      fun calculate(i:int,j:int):real
        = let
            val d:real = B@[j,j]-B@[i,i]
          in
            2*B@[i,j]/(d+sign(d)*sqrt(sqr(d)+4*sqr(B@[i,j])))
           end
      in
        generate(shape(B),fn(i:int,j:int)=>
            if (i>j)
            then calculate(i,j)
            else
        if (i=j)
        then 1.0
        else ~calculate(j,i))
      end
```

Figure 2.11: Functional definition of $transform$ function

The functional definition contains some detail additional to that contained in the mathematical definition (for example, the mathematical definition does not specify the size of the transformation matrix: it is inferred from context). The functional definition also looses some information that is present in the mathematical definition (for example, that the constructed matrix is non-singular and orthogonal). However, I believe that the functional specification is essentially a simple rewriting of the computational part of the mathematical definition. The provision of non-computational information (such as asserting that a matrix is orthogonal, or is sparse, or asserting the size of matrices) in definitions is a matter for further research; further discussion is beyond the scope of this thesis.

2.5 Array-Like Data Types in Mathematical Algorithms

In the preceding sections, I have argued that a functional programming language can provide a suitable conceptual and notational framework for specifying numerical mathematical algorithms. The vast majority of numerical algorithms employ some form of array data structure (vectors, matrices and grids), but the array is a data structure traditionally not well supported in functional languages. Thus, in this section, I discuss an Array Abstract Data Type that I have provided for use in definitions. The purpose of this discussion is not to provide an exhaustive list of function definitions, but to provide familiarity with a small number of general purpose functions.

Examples are drawn from linear algebra, but the ideas are readily extended to other fields.

2.5.1 Basic Operations

The list data type is ubiquitous in functional programming, so it is convenient to implementation other structured data types, such as array, using lists. However, it is important for program manipulation that the underlying representation be hidden (for example, parallelization of list operations can be difficult). So I present a small number of *primitive* functions that support the types of operations usually performed upon arrays; other array functions are defined in terms of these primitive functions, in complete ignorance of the underlying representation. All manipulation of arrays can be performed in terms of the primitive functions, rather than in terms of the underlying implementation.

The basic operations that are required are: constructing an array; given an array, obtaining the value associated with a particular location in the array; combining the elements of an array into a single value (called reduction; an example of a reduction is summing the elements of a real array). These are discussed in detail below: for convenience, indexing is discussed first (so I can use it in examples for the other two).

Indexing

An index for an n-dimensional array is specified in the Array ADT as an instance of the type `index`, which is defined as a list of n integers: e.g. `[1,2,3]`. Indexing is performed by the `element` function: e.g. `element(A,[1,2,3])`. For convenience, indexing can also be performed using the `@` infix operator: e.g. `A@[1,2,3]` (read as "A at (location) 1,2,3").

Construction

An array is a mapping from some (regular) index set onto some set of elements (all of the same, arbitrary type). Thus, the construction of an array requires the specification of (i) the index set and (ii) the set of elements.

(i) The most common way in mathematics to specify an index set, in one dimension, is to give an upper bound n: the index set is the set of integers from 1 to n. In some applications, it is useful to specify also a lower bound l and a stride s, the index set being the set of integers between lower and upper bound at multiples of the stride: $\{i : l \leq i \leq n, 0 \leq p : i = l + ps\}$. Index sets for higher dimensions are constructed as the Cartesian product of one dimensional sets.

In the Array ADT, an index set in one dimension can be specified as a triple of integers (specifying lower and upper bounds and stride). Multi-dimensional index sets are specified as lists of triples. For example `[(1,5,2),(2,4,2)]` specifies the set $\{(1,2),(3,2),(5,2),(1,4),(3,4),(5,4)\}$. A list of triples is an instance of the type `shape`.

(ii) The most common way to specify the set of elements onto which indices are mapped is to give a function which maps each single index onto a value. In the Array ADT, the function can be any valid function expression (of the appropriate type); function prototypes are often used.

Construction is realized in the Array ADT as the `generate` function:

```
generate:shape*(index->'a)->'a array
```

The first argument is the shape of the constructed array; the values of the elements are given by applying the second argument (the *generating function*) to each index that occurs in the shape. For example,

```
generate([(1,10,1)],fn(i:index)=>U@i)
```

is the vector consisting of the elements of the vector `U` at positions 1 to 10.

An important aspect of `generate` from an operational perspective is that no order is implied for the construction of the elements: the generating function may be applied simultaneously to all the indices in the specified shape.

For convenience, several other forms of `generate` are permitted:

- If the individual integer components of an index are required in a generating function, they can be made available using *pattern matching*, in which individual components of an index are named. For example

  ```
  generate([(1,10,1),(1,10,1)],fn([i,j]:index)=>A@[j,i])
  ```

  forms a matrix that is the transpose of the upper left 10 by 10 section of matrix `A`. The two components of the `index` argument of the function prototype are bound to the names `i` and `j`.

- A vector can be specified by providing a single integer for the shape (the upper bound) and a generating function which takes a single integer as argument (rather than a list of integers). For example

  ```
  generate(10,fn(i:int)=>2*i)
  ```

  evaluates to a vector with shape $\{i : 1 \leq i \leq 10 : i\}$ and with element i equal to $2 * i$.

- A matrix can be specified by providing two upper bounds and a generating function which takes two integers as argument. For example, if `A` has shape `[(1,m,1),(1,n,1)]` then

  ```
  generate(n,m,fn(i:int,j:int)=>A@[j,i])
  ```

  is the transpose of `A`.

Each of the 'convenient' forms of `generate` is defined in terms of the basic `generate` (the form with type `shape*(index->`α`)->`α `array`). The various forms are overloaded onto the name `generate`; the language system determines which form is to be used based upon type information. The signature of `generate` may thus be represented as

```
generate:shape*(index->'a)->'a array
generate:int*(int->'a)->'a array
generate:int*int*(int*int->'a)->'a array
```

Reduction

The reduction of the elements of an array to a single value is performed by the repeated application of a binary *reducing function*: each application of the binary function combines a pair of values into one value; its repeated application eventually combines all the elements into a single value.

Reductions are performed using the function `reduce`:

```
reduce:shape*(index->'a)*('a*'a->'a)*'a->'a .
```

A reduction is interpreted as follows:

- The second argument (a generating function) is applied to each index in the first argument (a shape) to produce a set of values G.
- To G is added the fourth argument, the *initial value*.
- The elements of G are combined together using the third argument — a binary function called the *reducing function* — to produce a single, cumulative value.

Thus, `reduce(S,g,r,init)` may be interpreted as $r(init, r(g(i_1), r(g(i_2), \ldots)))$, where i_1, i_2, etc. are the indices in the shape S. However, `reduce` does not specify an order for applying r, which should thus be associative and commutative. [6]

For example, summing the elements of a real array can be expressed as

```
sum(A:real array):real
  = reduce(shape(A),fn(i:index)=>A@i,fn(x:real,y:real)=>x+y,0.0)
```

This expression my be interpreted as: for each index i in the index set of A, compute A_i, and add all the values so obtained to the initial value 0.0.

As a second example, the logical and of the elements of a boolean array can be computed as

```
all(A:bool array):bool
  = reduce(shape(A),
    fn(i:index)=>A@i,fn(x:bool,y:bool)=>x andalso y,true)
```

The use of a shape and generating function to specify the values to be reduced may seem overly complex, but it does allow some common operations (such as the inner-product of two vectors) to be expressed succinctly; examples are discussed below.

For convenience, simple reductions (which directly combine the elements of an array) can be expressed using the function `fold`:

```
fold:'a array*('a*'a->'a)*'a->'a
fold(A:'a array,r:('a*'a->'a),init:'a):'a
  = generate(shape(A),fn(i:index)=>A@i,r,init)
```

e.g. summation could be expressed as

```
sum(A:real array):real = fold(A,fn(x:real,y:real)=>x+y,0.0)
```

Of course, it is somewhat academic how the common reductions are defined, since they are provided as library functions.

Array construction appears in many guises in mathematics: for example, as projections of part of a matrix (e.g. a row, a column or the diagonal elements) or as permutations of the elements (e.g. matrix transpose). Functions for such operations are readily defined using `generate`. However, one form of construction — mapping — occurs so frequently that it is worth highlighting here.

Mapping

In a unary mapping, an array is constructed by applying some specified function to each element

[6]The inclusion of an initial value ensures a reduction is well-defined even if there is only one index in the shape over which the reduction is performed.

of an array. Examples of unary mapping are scaling a numeric array (multiplying each element by some fixed value) and forming the logical complement of the elements of a boolean array.

In a binary mapping, an array is constructed from two arrays by applying some specified binary function to pairs of corresponding elements of the arrays. (The arrays have the same set of indices). Example of binary mappings are adding two numeric arrays or forming the logical *and* of two boolean arrays.

Unary and binary mappings are expressed using the map function:

```
map:′a array * (′a -> ′b) -> ′b array
map:′a array * ′b array * (′a * ′b -> ′c) -> ′c array
```

e.g. to negate the elements of a real array

```
negate(A:real array):real array = map(A,fn(x:real)=>~x)
```

and to add two integer arrays

```
plus(A:int array,B:int array):int array
  = map(A,B,fn(x:int,y:int)=>x+y)
```

The map functions are defined in terms of generate. For example, the unary (non-overloaded) version of map is defined as

```
unary_map(A:′a array,f:(′a->′b)):′b array
  = generate(shape(A),fn(i:index)=>f(A@i))
```

One further type of operation occurs frequently: obtaining information about the index set of an array. For example, the complete index set of an array A can be obtained by applying the function shape to the array; the size of the array in a particular dimension d (numbered from 0) can be obtained as size(A,d). Such operations are rarely found in mathematics texts as the index set is usually known from context. However, the functions are useful in definitions to enable functions to be constructed independent of context.

2.5.2 Two Properties of the Basic Functions

It should be clear that the following two identities hold for generate:

$$element(generate(S,\lambda i{\cdot}g),i')\equiv\lambda i{\cdot}g(i')$$

$$shape(element(S,\lambda i{\cdot}g))\equiv S$$

That is, the shape of an array constructed by the generate function is simply that shape passed as argument to generate; and the value of any element of the array is found by applying the generating function to the index for that element.

These two identities form the basis of most of the manipulation of arrays.

2.5.3 Other Common Operations

To further illustrate the use of the Array ADT, some common operations in linear algebra are defined below:

- Inner product of two vectors.

- Left and right multiplication of a vector by a matrix.
- Algebraic multiplication of two matrices.

There are various mathematical definitions of the inner product of two vectors. Perhaps the simplest is that the inner product is the sum of the product of the vectors:

$$(U,V) \equiv sum(U.V).$$

A similar SML definition can be given:

```
fun times(U:real vector, V:real vector):real vector
  = map(U,V,fn(x:real,y:real)=>x*y)
fun innerproduct(U:real vector, V:real vector):real
  = sum(times(U,V)).
```

The Array ADT defines three functions for vector-matrix, matrix-matrix and matrix-vector multiplication; these functions are overloaded onto the `*` operator, so the programmer does not have to distinguish them.(v)

The definitions of vector-matrix and matrix-vector multiplication are

```
fun vmmult(V:real vector,A:real matrix):real vector
  = generate(size(A,1),fn(i:int)=>innerproduct(V,col(A,i)))
fun mvmult(A:real array,V:real vector):real vector
  = generate(size(A,0),fn(i:int)=>innerproduct(row(A,i),V))
```

where the functions `row` and `col` return the specified row or column of a matrix.

The following definition of matrix-matrix multiplication follows from the 'row-into-column' mathematical definition:

```
fun matrix_matrix_product(A:real matrix,B:real matrix):real matrix
  = generate(size(A,0),size(B,1),
    fn(i,j)=>innerproduct(row(A,i),col(B,j))) .
```

An alternative mathematical definition of matrix-matrix multiplication is: if A has dimensions $N_1 \times N_2$ and B has dimensions $N_2 \times N_3$ then $P \equiv A \times B$ has dimensions $N_1 \times N_3$ and elements given by

$$P_{ij} \equiv \sum_{k=1}^{N_2} A_{ik} B_{kj}.$$

An equivalent SML definition is:

```
fun matrix_matrix_product(A:real matrix,B:real matrix):real matrix
  = generate(size(A,0),size(B,1),
      fn(i,j)=>reduce(size(A,1),fn(k)=>A@[i,k]*B@[k,j],fn(x,y)=>x+y,0)).
```

When considered from an operational perspective, this definition may be considered more efficient than the previous definition as it avoids the creation of intermediate vectors (for the rows and columns of A and B, and the elementwise products thereof) — the multiplication of the elements is merged with the summation of those products. However, efficiency is not a primary concern in definitions: a programmer should normally construct definitions in the modular manner of the first definition. The second definition does though illustrate the greater flexibility of `reduce` function over `fold`.[7]

[7]The second, efficient version can be derived from the first, modular definition through program transformation.

2.6 Functional Specifications

The functional definition of an algorithm is the source from which imperative implementations of the algorithm are derived. An implementation is naturally required to be ‘equivalent’ to the definition. The term ‘equivalent’ requires some clarification.

A functional definition is an expression, and I have emphasized throughout this chapter that the meaning of an expression is the value of the expression. Thus, one aspect of equivalence is that an implementation computes the same value as its definition.

However, that is not sufficient. Often there are several algorithms that, given the same input, will compute the same value: for example, Jacobi’s algorithm and the Conjugate Gradient algorithm both compute the unique solution to a system of linear equations; the POT algorithm and Householder’s algorithm both compute the unique eigensystem of a set of equations. A derivation that begins with a definition of POT should not end with an implementation of Householder!

What distinguishes one algorithm from another that computes the same results is *how* the results are computed; thus, a second aspect of equivalence is that an implementation of a definition must also use the same method to compute that value.

That is not to say that a derivation must preserve every detail of the method employed by a definition: in any definition, some aspects are important, while other aspects are present merely to conform to some formal syntax or to allow the definition to be executed. An implementation is required to reflect only the important aspects of the method.

For example, the SML version of the Array ADT uses lists to represent arrays, indices and index sets: such a representation is convenient in SML. However, the details of the representation are entirely unimportant as regards the definition of an algorithm; indeed, such details are deliberately withheld from algorithm developers, who work only in terms of the array ADT functions.

Thus, there is no obligation on an implementation to reflect the list operations present in a definition: an expression such as

```
generate([n,n],fn(i,j)=>if (i=j) then 1 else 0)
```

may be implemented as:

```
   DO 20 i=1,n
   DO 10 j=1,n
   A(i,j) = 0
10 CONTINUE
   A(i,i) = 1
20 CONTINUE
```

Fortran77

or

```
A = 0
A(patunitdiag(n)) = 1
```

Fortran Plus Enhanced

with no list-based operations at all.

Thus, there are aspects of a definition that are central to the nature of the algorithm being defined, aspects that must be reflected in an implementation; and there are aspects of a definition that are almost irrelevant: how, or even if, these latter aspects are reflected in an implementation is a matter of convenience.

An example may help illustrate the difference. The functional definitions in figures 2.9 and 2.11 together define the POT algorithm. These definitions contain three important functions: `POT`, `POTStep` and `Transform`. `POT` defines the overall nature of the algorithm: it is iterative. `POTStep` and `Transform` together define the nature of each iteration. The three functions together capture the essence of the POT algorithm.

The definitions of other functions in the figures, such as `orthonormalise` and the linear algebra functions (primarily matrix multiplication) do not characterize the algorithm: an implementation of POT could use *any* orthonormalization function and *any* form of matrix multiplication, and the implementation would still be an implementation of the POT algorithm.[8]

However, although the functions `POT`, `POTStep` and `Transform` are the important aspects of the definition, an implementation need not contain these functions *as functions*, as separate program units. What is important is that an implementation carry out the process defined by these functions. For example, the SML implementation of `iterate`, which defines the iterative nature of POT, may be recursive but a Fortran implementation of POT may perform the iterations using a simple loop. Further, the definitions of `POTStep` and `Transform` would probably be 'unfolded' so that neither function would exist as a separate program entity.

Similarly, the existence of variables corresponding the the matrices `V`, `B`, `T` and `R` in figure 2.9 is *not* required. Structure, such as local definitions and function definitions, is taken to exist in a definition primarily to assist human readers: it should remain in an implementation only if it improves efficiency.

Thus, while a functional definition must contain complete details of all computations if it is to be executable, many of these details may be ignored or interpreted rather liberally when the definition is implemented. In such a context, I refer to a functional definition as a *functional specification*, to emphasize that the definition is indicative rather than prescriptive, defining what is to be computed rather than the details of how the computation is to be performed.

This emphasis on the indicative rather than prescriptive nature of functional specifications is motivated by a number of considerations:

- Discarding unimportant structure, such as function definitions, allows the programmer the freedom to employ whatever level of functional decomposition he feels is useful in ensuring his specification is a clear, natural statement of the algorithm, while knowing that the efficiency of implementations is not going to be impaired.
- Ignoring the implementation details of the basic array functions allows the development of an array semantics (and hence, manipulation techniques) based upon the axioms of the basic functions, rather than based on the properties of the whatever representation happens to have been used in the SML library that is used for rapid prototyping.
- A derivation is free to implement operations in whatever manner is best suited to the implementation environment (regardless of what form was convenient for expressing the operation in a specification).

2.7 Summary

A certain form of algorithm definition employed by mathematicians, expression-based definitions, provides a high degree of clarity and possesses simple semantics. While not every mathematical algorithm definition is of this form, the form is nevertheless consistent with the general principles of mathematics. However, the form, while expression-based, does use natural language and so is not easily manipulated by machine.

The definition language that I use in this thesis is a functional programming language; it may be viewed as essentially a completely formal version of the mathematical form of definition, being based

[8]The details of the implementations of these functions is still important though: orthonormalization and the matrix products make up the vast bulk of the execution time.

upon expressions and often employing a similar notation. The functional specification of an algorithm is often merely a rewriting of a mathematical definition, to conform to some precise notation.

A functional specification is designed to capture in a clear, simple, natural manner the essential aspects of an algorithm; those aspects that define *that* algorithm as opposed to some other algorithm.

2.8 Chapter Notes

i (page 9)

Having emphasized the difference between an algorithm and an implementation, I should note that the distinction can be blurred, and that not everyone would agree with my viewpoint. Many people would view a computer program, say a Fortran program, as an algorithm rather than just one of possibly many potential realizations of a more abstract entity. In addition, a computer program may be expressed in a language that provides a programming model that is, to a greater or lesser extent, abstracted from computer systems. For example:

- Standards such as the emerging High Performance Fortran [59] are designed to eliminate some of the sequential architecture characteristics inherent in Fortran77.
- The recent Message Passing Interface [52] is a generic message passing standard, designed to function with any message passing architecture.
- A more fundamental level of abstraction is provided by many functional programming languages, whose programming models are often unrelated to any physical computer system.

This last point about functional programming languages is particularly important for the work discussed in this thesis. The importance stems from the facts that many mathematical algorithms are defined, by mathematicians, in terms of expressions, and that functional languages are expression based. This is discussed in more detail later.

ii (page 18)

Aside from the value-based, or *denotational* meaning, a second, *operational* meaning can also be associated with an expression: for example, the denotatonal meaning of $5+3$ is 8 while the operational meaning is "perform the integer addition operation with operands 5 and 3". Operational meanings can be somewhat more complex than denotational meanings, but fortunately much manipulation and analysis can be performed using only the denotational meaning; the operational meaning is important mainly when execution complexity is being considered or when there is the possibility of non-termination or computational error. Almost all of the work reported in this thesis is based upon denotational meaning.

iii (page 19)

The formal meaning of a function definition/application is somewhat more complex, due to the possibility of recursive definitions, but the ultimate meaning of a function application is still simply a value.

iv (page 19)

The value computed by a function is also determined by global function and value definitions. The important difference between such definitions and global variables in imperative programs is that the former are constant; they cannot change by the program itself.

v (page 30)

In mathematical definitions of algorithms, vectors enjoy something of a dual nature: sometimes they are one-dimensional objects, sometimes they are two-dimensional. This is apparent when a mathematician uses the terms "row-vector" and "column-vector": two vectors, with the same elements in the same order, may be distinguished only by their orientation in a two-dimensional plane. Thus, it is legal to multiply a matrix by a column vector on the right, but not by a row vector, and vice-versa for left multiplication. So in mathematical texts, one often sees expressions involving the transpose of a vector, the meaning of which is not clear if a vector is a one-dimensional object; the simplest interpretation is that there is an implicit promotion of a vector to a matrix in which one of the dimensions is 1, so that vector transpose is actually matrix transpose and vector-matrix and matrix-vector multiplication are actually matrix-matrix multiplication. (Some authors claim that "vector" is simply another name for "matrix", but they still distinguish the two, when discussing dimensions and in the the number of subscripts used for indexing, for example.)

Chapter 3

Implementation of Specifications

In the previous chapter, I introduced the style of specifications used in this work for numerical mathematical algorithms. Such specifications can be useful in themselves, for formally defining and investigating algorithms. However, functional specifications are also intended to be the basis from which efficient implementations are produced.

As the specification language is a subset of a programming language, a specification itself could serve as an implementation. However, few practitioners in numerical mathematics would even consider using a functional language to implement algorithms, for two main reasons:

- Many practitioners in numerical mathematics seem obsessed with performance, to the extent that an implementation of an algorithm may be considered worthless if its execution performance is even slightly worse (say, by a few percent) than that of other implementations. By this standard, the execution performance on a given computer of programs produced from functional specifications by traditional compilers is inadequate when compared with the performance that can be achieved on the same computer by hand-crafted (imperative) implementations.(i)
- Few compilers for functional languages are available on most of the advanced computer systems used for numerical computing.

To overcome these problems, implementations in an imperative language are constructed from functional specifications; the intention is that the imperative implementations be as efficient as possible. In this chapter, I discuss general methods that can be used for constructing implementations and the problems that arise during construction; in the following chapter, I discuss in detail the method used for this thesis.

The discussion is as follows:

- The efficiency of functional programs.
- The problems encountered in implementing a function specification in an imperative language.
- Methods used to ameliorate these problems: top-down decomposition, program refinement and transformation.

3.1 Efficiency of Functional Programs

Generally, functional implementations of numerical algorithms are inefficient: implementations of the algorithm could be produced that provide much better execution performance. There are two main reasons for this inefficiency:

The functional style emphasizes clarity over efficiency

A programmer may develop modular, generally applicable routines; may employ functional abstraction of data; may develop higher-order functions that encapsulate common patterns of computation. All these techniques help structure a program in a way that facilitates human understanding of the program and simplifies reasoning about the program, but they can also reduce the efficiency of an implementation.

Even at the level of a single function, a programmer may choose a simple, clear form for the function over a more complex, but also more efficient form. For example, the Fibonacci numbers are defined by

$$fib(n) = \begin{cases} 1, & n = 1, 2 \\ fib(n-1) + fib(n-2), & n > 2 \end{cases} .$$

The SML functions `fib` and `fibit` both compute the Fibonacci numbers.

```
fun fib(n:int)
  = if (n=1 orelse n=2) then 1 else fib(n-1)+fib(n-2)
```

```
fun fibitaux(n:int,i:int,previous:int,current:int)
  = if (i>n)
  then current
  else fibitaux(n,i+1,current,previous+current)

fun fibit(n:int)
  = if (n=1 orelse n=2) then 1 else fibitaux(n,3,1,1)
```

The definition of the `fib` function may be preferred to that of the `fibit` function because the former is simpler and mirrors the mathematical definition. However, `fib` has exponential complexity while `fibit` has linear complexity.

The effect of a simple, clear style on efficiency is particularly important for functional specifications, since the underlying philosophy of functional specification is to enhance clarity with no regard whatsoever for efficiency.

Functional languages introduce overheads

Functional languages involve many concepts which ease using the language, but which are expensive to implement. For example, referential transparency is one of the most useful properties of functional languages, but it requires an implementation to continually create copies of data structures rather than updating existing structures, introducing overheads for copying and for managing storage (e.g. through some form of garbage collection).

Functional languages tend to have a minimalist conceptual basis: for example, all computation being performed through the application of functions. Such minimality facilitates learning a language, developing a formal semantics for the language and manipulating programs. However, it also introduces further overheads: for example, in many functional languages, all iteration is expressed using recursive function calls, but some forms of iteration can be implemented much more efficiently at the machine level as a simple loop updating a fixed set of storage locations, avoiding the overheads of the function calling mechanism.

In addition, functional languages tend to simplify the expression of algorithms by avoiding many of the restrictions usually associated with imperative languages. For example, a function application in a functional language can return a value of any type (a simple scalar value, a structured value such as an array, or even a function), whereas most imperative languages restrict functions to returning a single scalar value.

The language implementation must provide general purpose mechanisms to support this expressiveness: e.g. the function application mechanism must allow the return of a value that may occupy an arbitrary number of storage locations. In contrast, the function application mechanism for an imperative language typically permits only scalar values to be returned.

Some of the above problems can be addressed by a compiler:

- The compiler may perform function unfolding, which helps ameliorate the detrimental performance effects of an abstract, modular style. (However, as explained later in section 5.7, unfolding is not sufficient to completely remove the detrimental effects.)
- Some forms of recursion, such as tail recursion, may be recognized and implemented as simple loops, rather than recursive functions.
- Analysis of the use of data structures may reveal instances where a data structure can be overwritten. For example, consider the expression

  ```
  λx:real vector·λy:real vector·B (times(x,x)) (e)
  ```

  in which the vector `x` is bound to some expression `e` and the vector `y` has elements that are the squares of the elements of `x`. In general, the storage for `x` and `y` must be distinct, since `x` may be used in the body `B` of the expression, and so its value must be preserved until evaluation of `B` is complete.

  However, if `B` does not use `x`, then `y` can use the same storage locations as `x`: the values stored in those locations are squared *in situ*.

  In general though, the analysis required to permit such destructive overwriting of storage is complex.
- Type information may be used to allow specific, efficient forms of function calling protocol to be used. For example, if the return type of a function is `integer`, the result of an application of the function may be returned in a processor register.

The central point though is that pure functional languages do not provide means for *the programmer* to address some issues that are important as regards execution efficiency. In contrast, most imperative languages not only allow the programmer to address such issues but often *require* him so to do:

- The programmer must explicitly organize the use of storage.
- Fortran77 does not allow recursive functions, so the programmer must implement iteration using loops (and possibly some storage stack mechanism).
- Many imperative languages restrict functions to return only scalar values. If a function is to compute a structured value, the programmer must explicitly provide a storage location for the result.

That functional languages do not allow such fine granularity of control over the machine is not a criticism: functional languages are designed with an entirely different philosophy. However, it does seem the considerable effort necessarily devoted by programmers in the manual construction of an imperative program is likely to result in significantly greater efficiency that can be obtained by compiling a functional program.

Indeed, it is an open question as to whether *any* automated system can be as effective as a programmer as regards efficiency of execution; but it is a tenet of this thesis that automated systems can be as effective for at least certain classes of algorithms.

3.2 Imperative Implementations

When efficiency is of primary importance, an imperative language such as Fortran or C is probably a better choice of language than a functional language: the closeness of imperative languages to physical implementation systems, which makes imperative languages ill-suited as algorithm specification languages, does permit the implementation system to be controlled to the fine degree required to obtain the best performance from the system.(ii)

Thus, for specifying algorithms I use a functional language; for implementing algorithms, I use an imperative language. The problem addressed here is, given a functional specification, how to obtain an imperative implementation. Typically, a programmer may proceed with the construction of an imperative implementation in two ways:

- A programmer can construct an imperative implementation based directly upon the functional specification: i.e. the textual contents of the specification are manipulated so as to produce an implementation. In this scenario, the functional specification is the definitive statement of the algorithm: the implementation is a faithful reflection, by construction, of the specification.
- Alternatively the programmer can use the functional specification to obtain an understanding of the algorithm, and then somehow ‘come up with’ an implementation that faithfully reflects that process. In this scenario, the functional specification’s rôle is as a tool for informing the programmer: the functional specification is a concise yet precise statement of the algorithm, but there is no direct link between specification and implementation.(iii)

The merits of these two methods for producing an implementation are discussed below. But first, I discuss *implementation issues* that must be addressed by the programmer, regardless of the method used for implementation.

3.2.1 Implementation Issues

When constructing an imperative implementation of a functional specification, a programmer must address several (probably many) implementation issues, caused by the radically different natures of the definition and implementation languages, and by performance criteria: examples of such issues are discussed below.

Conversion from expression basis to state basis The functional specification is expression based: operations are denoted as pure functions and execution of the algorithm corresponds to evaluation of the expressions. In contrast, the imperative implementation language is state-based: operations are performed by altering the values of variables.

The programmer must establish a correspondence between the values computed by the expressions and the values stored in variables. For example, suppose an algorithm repeatedly applies some function to a set of values[1]:

$$F(a,b,c) = F(\alpha(a,b,c), \beta(a,b,c), \gamma(a,b,c)) \ .$$

The programmer would implement this function as a loop, of which each iteration updates variables a,b and c

```
REPEAT (a,b,c) := (α(a,b,c),β(a,b,c),γ(a,b,c)).
```

If the implementation language does not permit the simultaneous assignment of multiple variables, then the programmer must produce a sequence of single variable assignments. Since the functions α, β and γ may make use of the values of any or all of the variables a, b and c, the assignments to the variables must be performed in such a manner that each of the variables is overwritten only after its original value is no longer required. Depending on the nature of the functions, it may be possible to satisfy this condition by simply ordering the assignments to a, b and c; but in general additional variables must be introduced in which the values of the function applications are stored until all of the functions have been computed, when the values are copied into a, b and c.

Further, the implementation language may not permit the functions α, β and γ to be denoted as expressions; for example, Fortran does not permit conditional *expressions*, and does not permit expressions to evaluate to structured values such as arrays. Then the programmer may have to convert the function applications into statement sequences.

Implementation of Abstract Data Types If an algorithm manipulates abstract data types, the programmer must implement the ADTs using whatever data types are provided by the programming language. For example, the majority of numerical algorithms employ a vector, matrix or grid ADT, which may be implemented in, say, Fortran using the array data type.

Several factors may affect the implementation of ADTs:

- The implementation language may not provide any module mechanism by which the *abstract* nature of the algorithm data types may be maintained; then the concrete representation cannot be isolated from the rest of the implementation and the programmer cannot enforce a suitable, abstract interface to the ADT. The representation of the ADT becomes manifest throughout the implementation: manipulation of the underlying representation of the ADT takes place at each point in the program which uses an ADT operation, rather than being encapsulated within the ADT itself. Such ubiquity of the representation of ADTs obfuscates the implementation and hinders the implementation process. (It also hinders maintenance and extension of the implementation.)

- Even if a module mechanism is provided by the implementation language, efficiency considerations may require the programmer to eschew its use, and to manipulate directly the underlying representation of an ADT (as described above) since the use of a module mechanism (even a basic one such as subroutines) can decrease efficiency, partly by introducing overheads (e.g. for the subroutine invocation mechanism), but more importantly by separating operations that can be more efficiently implemented in a combined form.

  For example, if a matrix ADT provides a routine to add two matrices, a natural way to add three matrices is to invoke the subroutine twice: i.e. $A := B + C + D$ is implemented as shown in figure 3.3(a), requiring two loop executions and an additional matrix (T) to store an intermediate value. For some computational architectures, the pair of additions can be more efficiently implemented as a single loop which combines the additions, as shown in figure 3.3(b).

[1]The function used in the example is non-terminating, but this is just an example!

```
subroutine add(A,B,R,m,n)
  integer m,n
  real A[m,n],B[m,n],C[m,n]
  DO i := 1,m
  DO j := 1,n
    R[i,j] := A[i,j]+B[i,j]
  END
  END
RETURN

integer m,n
real A[m,n],B[m,n]
real C[m,n],D[m,n]
real T[m,n]
T := add(B,C,m,n)
A := add(T,D,m,n)
```

Figure 3.1: (a) Inefficient

```
integer m,n
real A[m,n],B[m,n]
real C[m,n],D[m,n]
DO i := 1,m
DO j := 1,n
  A[i,j] := B[i,j]+C[i,j]+D[i,j]
END
END
```

Figure 3.2: (b) Efficient

Figure 3.3: Two implementations of triple matrix addition

(This form is more efficient as the overheads for managing the loops have been reduced, there is one fewer assignment to an array, and one fewer array variable is required.) The combination of operations in this manner can be very important for the efficiency of numerical algorithms, the majority of which make extensive use of array operations.

Thus, although the use of a module mechanism is usually desirable to facilitate construction of the implementation, efficiency may require the direct manipulation of the concrete representation. The interaction of multiple operations of even a single ADT can cause an implementation to become quite intricate; if an algorithm uses multiple ADTs, the implementation can become awash with minutiae.

- A simple, elegant representation of an ADT may not be the most efficient, therefore a programmer may choose a more complex, but more efficient, representation, causing a corresponding increase in complexity of the implementations of the ADT operations. For example, a simple representation of a matrix is a two-dimensional array, but a programmer may decide to 'flatten' the array into one-dimension, as this can increase efficiency. Flattening can complicate expressions though: for example, the $(i, j)^{th}$ element of a matrix A of width m is denoted in the two-dimensional representation as `A(i,j)`, but as `A(i+(j-1)*m)` in the one-dimensional representation.

Implementation of recursion Many mathematical algorithms involve recursive computations. Where possible, a programmer should, for reasons of efficiency, implement recursive computations as simple loops manipulating a fixed set of variables. For some types of recursion (e.g. tail recursion) the conversion to simple iteration is straightforward; for other types of recursion (e.g. exponential recursion, as occurs in the Fibonacci function, page 36), complex analysis is required; indeed, there may not be be any systematic way to convert the recursion into iteration.

Storage management If the size or number of data structures manipulated by an algorithm cannot be fixed before execution of the algorithm, the programmer must implement some form of storage management. A common Fortran programming technique is to allocate a large array of fixed size, from which vectors and matrices can be dynamically allocated. As before, this implementation technique is conceptually simple, but it can give rise to cumbersome, intricate computations; the implementation of such computations is error-prone.

For example, suppose an algorithm is to multiply matrix A of size $m_a \times n_a$ by matrix B of size $m_b \times n_b$ (with $n_a = m_b$). The computation of the product may be implemented as

```
  integer MAXM,MAXN
  parameter(MAXM=???,MAXN=???)
  real W(MAXM,MAXN)
  integer ma,na,mb,nb,mc,nc
  integer wa,wb,wc
  ⋮

  mc = ma
  nc = nb
  wa = 1
  wb = wa+ma*na
  wc = wb+mb*nb
  DO 2 i=1,mc
  DO 2 j=1,nc
  W(wc+i+(j-1)*mc) = 0
  DO 1 k=1,na
  W(wc+i) = W(wc+i)+W(wa+i+(k-1)*ma)*W(wb*k+(j-1)*mb)
1 CONTINUE
2 CONTINUE
```

where A is stored in the first $m_a.n_a$ elements of the array `W`, B is stored in the next $m_b.n_b$ elements. The product is a matrix of size $m_a \times n_b$ and is stored in the next $m_a.n_b$ locations in `W`.

Vectorization and parallelization Many vector and parallel computer systems are programmed using Fortran77. It is the task of the compiler to interpret a Fortran77 program in such a way as to make efficient use of the computer's hardware, to vectorize or parallelize the program. This task is rather complex, so often a compiler will fail to vectorize or parallelize sections of a program that nevertheless could be executed efficiently by the hardware. In such cases, a programmer may have to reorganize sections of the program.

Often the burden of such reorganization is not that it is difficult, but that there is no alternative to experimenting with different forms to discover which provides the best execution performance. For example, the two loops

```
  DO 2 i=1,n
  V(i) = W(i)
  DO 1 j=1,n
  V(i) = V(i)+A(i,j)*U(j)
1 CONTINUE
2 CONTINUE
```

and

```
  DO 2 i=1,n
  V(i) = 0.0
  DO 1 j=1,n
  V(i) = V(i)+A(i,j)*U(j)
1 CONTINUE
  V(i) = V(i)+W(i)
2 CONTINUE
```

both implement the combination of matrix-vector product with vector addition (they compute the vector $A * U + W$). They differ only in that the addition of one component of W is performed in the first implementation before the inner loop, and in the second, after the inner loop. Yet this small difference results in considerable difference in execution performance.

Process management The programmer may have to organize processes and their communication or synchronization.

For example, in a distributed memory system, one processor may require a row of a matrix stored on another processor; the programmer would typically have to arrange the following:

Initialization Before any communication can be performed, some form of initialization must typically be performed. This may involve, for example, allocating buffers for the communication routines, assigning 'communication channels' and setting routing options.

Sending processor The sending processor must copy the row into a vector (because communication routines normally require data to be stored in consecutive storage locations but, in Fortran, the elements across a row are not stored consecutively). It must then call the 'send' routine, specifying the vector, the size (in bytes) of the data to be sent and the destination of the data, and possibly several other parameters such as which 'communication channel' to use, what 'mode' of communication (such as buffered/non-buffered, synchronous/asynchronous).

Receiving processor The receiving processor must call the 'receive' routine, specifying a vector in which the data is to be stored.

Depending upon the characteristics of the computer system, it may be important to overlap communication with computation, so rather than simply invoking the 'send' routine, the sending processor must initiate a send and proceed with computation, checking at regular intervals if the send has completed (assuming it has more data to be sent). The receiving processor must initiate a receive and proceed with computation, periodically checking if the receive is complete, and perhaps waiting for completion if it reaches a point in the computation that requires the data being received.

Summary

When constructing an imperative implementation of an algorithm from a high-level, abstract, machine-independent specification, the programmer must address many implementation issues. For each issue, he must choose some implementation technique. Often, the implementation techniques are not conceptually complex, but their combination can cause an implementation to become difficult to untangle.

3.3 Implementation based Indirectly upon Specification

A programmer has been given a functional specification of an algorithm and is to produce an imperative implementation of the algorithm (that is consistent with the specification). He reads the specification to obtain an understanding of the algorithm and sets about constructing an implementation based upon this understanding (rather than based directly upon the specification), perhaps referring to the specification for details.

The programmer may construct an implementation gradually, by first constructing an outline of the implementation and then refining the outline into a complete implementation; or the programmer may attempt an immediate construction, by constructing the final implementation complete in all details, with no preliminary, outline implementations. These methods of implementation are discussed below.

3.3.1 Immediate Construction

For simple algorithms, a programmer may immediately construct an implementation based upon the algorithm's definition. For more complex algorithms though, there are several problems with immediate construction:

- All main implementation issues, such as those discussed above, are addressed simultaneously — a rather daunting prospect for non-trivial algorithms as, although most issues are conceptually simple, their combination often results in intricate computations. It is often relatively simple to construct an implementation that is correct *in form*, but ensuring the *details* are correct is much more difficult. Thus direct, manual implementation is error prone.

- The methods used to address all the implementation issues are manifest simultaneously in the implementation. It can be difficult to decide from examination of the implementation itself, what implementation issue caused a particular section to have the form that it does, or to identify those sections that were influenced by a particular implementation issue.

  For example, if a particular array index expression is preventing the vectorization of a loop, it may not be easy to identify why the expression has the form that it does, and if an alternate form could be produced that would allow vectorization. Or if an alternative method for addressing some implementation issue is proposed (say a different partitioning scheme for a distributed memory architecture), it may not be easy either to identify the changes required in the implementation, or to assess the impact of adopting the alternative (as regards execution performance or the structure of the implementation). Similarly, if the algorithm is changed significantly, adapting the implementation to correspond may be difficult.

- As explained in the previous section, for efficiency, a programmer may eschew the use of modular programming to support a degree of abstraction in the implementation, in favour of direct manipulation of concrete representations. One consequence of this is that methods used to address implementation issues are manifest throughout an implementation. Thus, adopting an alternative method to address an issue may require extensive changes throughout the implementation. Performing all such changes manually is tedious and error-prone.

- Manual construction cannot be trusted to produce a correct implementation *ab initio*, but *a posteriori* verification is difficult since an informal review is hindered by the mass of detail in an implementation and a formal proof is impractical for non-trivial programs due again to the mass of detail and also due to the cumbersome semantics of state-based languages.

3.3.2 Top-Down Decomposition and Stepwise Refinement

Rather than construct an imperative implementation directly, a programmer may gradually construct the implementation using *stepwise refinement*: the programmer initially constructs an outline of the imperative implementation in which major operations are indicated (e.g. by name or by type signatures) but not detailed, and in which most implementation issues are ignored; he then *refines* the implementation by detailing each of the outlined operations individually and by incorporating techniques to address the implementation issues. A complex operation may require several layers of refinement: the major steps of the operation are outlined, then the major steps in each of these steps are outlined, and so on, until the steps are considered simple enough to implement directly.

For example, a programmer may initially assume that the implementation language provides a module mechanism for defining ADTs, and construct his initial implementation using ADT operations:

```
real matrix[n,n] A,B,C
A := B+C
```

If the implementation language does not provide a module mechanism, he may then refine his implementation by replacing instances of the ADT operations with equivalent operations on the representation:

```
real array(n,n) A,B,C
DO i := 1,n
DO j := 1,n
  A[i,j] := B[i,j]+C[i,j]
END
END
```

Similarly, a programmer may initially assume that all data may be referenced directly, then refine his implementation for a distributed memory system by introducing communication where required.

The following observations may be made about stepwise refinement:

- The intellectual task of implementing an algorithm is simplified since a programmer need consider only one operation or implementation issue at a time. Implementation issues can not, however, be considered independently: each refinement alters the implementation and such alteration may need to be allowed for in successive refinements.
- It may be possible to formally prove that each refinement preserves the meaning of the implementation (though this is rarely attempted for complex algorithms implemented in widely used languages). It should be easier (though probably even more tedious) to validate each refinement than to validate the final implementation.
- An implementation produced through top-down decomposition may be inefficient, due to the separation of operations that can be more efficiently implemented when combined (see the discussion on page 39).
- If refinements are to be performed entirely manually, then refining a non-trivial algorithm is a tedious and error-prone task.
- It should be possible for the programmer to maintain a copy of each version of an implementation as refinements are performed, so that the entire 'refinement history' is available. Then, if a different method for addressing some implementation issue is required, the programmer can select an appropriate version of the implementation and begin a new refinement.

  In practice, the maintenance of such a refinement history can be cumbersome and a programmer is more likely to refine a program by altering a single document. Then, it is impossible to revert to where a particular implementation issue was addressed, so if a new approach is required, the entire refinement process may need to be started afresh.

3.3.3 Program Transformation and Derivation

Certain forms of refinements will be used over and over, so it is often worthwhile to generalize a refinement by parameterizing it over program constructs (such as statements or variable names) to form a *refinement rule* or *transformation*. For example, the transformation

```
X := if (p) then T else F
→
if (p)
then X := T
else X := F
```
where `X` is a variable, `p`, `T` and `F` are expressions

converts the assignment of a conditional expression into a conditional statement.

The advantages of developing transformations are:

- A transformation need be validated only once, rather than each time it is applied.
- A transformation reduces the refinement process to the application of rules (requiring little or no consideration from the programmer), and so reduces the possibility of error in application.
- Some transformations have the potential for the programmer to be eliminated from the application process altogether. (Some transformations may require programmer guidance as to *when and where* they should be applied, but the *mechanics* of application may still be automated.) Thus, much of the tedium and potential for error can be removed from refinement.

- Particularly when transformations are automated, a programmer's efforts are devoted to developing transformations that may be used for many refinements, rather than to performing a single refinement.
- As formal constructs, transformations themselves become amenable to formal analysis.

Transformations have some disadvantages:

- The implementation method is rather circumspect: every change made to a program is based upon some formal rule that has perhaps been formally verified. It if often tempting to make a change that is 'obviously' correct without generalizing the change into a transformation.
- An enormous number of applications of transformations are usually required for non-trivial algorithms, making refinement extremely tedious if performed manually. The number of applications can perhaps be reduced by developing transformations that perform large scale changes to a program. However, such transformations tend to be difficult to verify and also tend to be very specific (resulting in an exponential growth in the number of transformations with which a programmer is expected to be familiar).

When refinement is performed through transformations, it is often called *derivation*: the refined version of a program is derived from some source version.

3.4 Implementation based Directly upon Specification

The use of program refinement and transformation, as discussed in the previous section, can help improve the chances of an implementation being correct by introducing at least some formality into the implementation process. However, the original source which is refined into an implementation is produced informally: a programmer reads the functional specification of an algorithm and then produces some initial outline of an imperative implementation. There is no guarantee that the outline implementation is consistent with the functional specification, and so no guarantee that the complete implementation is consistent with the functional specification (only that it is consistent with the outline implementation).

To ensure that an implementation is consistent with a functional specification, the implementation process can be based *directly* upon the specification: the functional specification itself is used as the source from which an implementation is produced by refinement or transformation. This is the basis of the method used to produce implementations for this thesis: formal transformations are automatically applied to a functional specification to derive imperative implementations. Details of transformations and the derivation process are discussed in following chapters. Here, I summarize the merits of this method of producing implementations:

- Because the implementation process is based upon a functional specification, there is a direct link between specification and implementation.
- A programmer develops an implementation *method*, encoded as transformations, rather than a single implementation.
- Each implementation issue can be addressed by a separate set of transformations, thus simplifying and providing some structure to the implementation process.
- Transformations are formal, and so potentially can be formally shown to preserve the meaning of a program. If each transformation preserves the meaning of a program, then any number of applications of the transformations must also preserve the meaning.

In practice, most of the transformations used for this thesis are simple enough to be evidently correct.

- Application is entirely automatic: a programmer selects a sequence of transformations and initiates the derivation process — he has no further rôle in the process. Little effort is thus required of the programmer to apply a derivation, and the possibility of an error occurring in the application of a transformation is reduced to that of there being an error in the transformation system itself.
- The derivation process itself is formal, and so there is the possibility that it too can be formally investigated (to prove completeness or termination, for example).
- Because transformations are applied by machine, it is mostly irrelevant how many applications are required. There is thus not the same tendency to amass a huge number of ever more complex transformations as when transformations are applied manually.
- Again, because derivations are automatic, a simple solution exists to the problem of adapting an implementation should a specification change: re-apply the derivation, to the modified specification; or should a new method be required to address some implementation issue: modify the derivation and re-apply.

That the application of transformations is entirely automatic may also be seen to be a disadvantage, as it eliminates the potential to use human judgement directly in the implementation process; it is thus an open question as to whether automatic transformation is entirely general.

3.5 Summary

Functional specification is a method for defining algorithms in a high-level, abstract manner that facilitates understanding, reasoning and manipulation. But to obtain high efficiency from most of the advanced computer systems available today, it is necessary to construct imperative implementations in low-level, machine specific imperative languages.

When constructing an implementation, a programmer must address many implementation issues caused by the discrepancy between specification language and implementation language. He may choose to tackle all issues simultaneously, but such an approach is error prone and wasteful of programmer time and effort, as each implementation must be started afresh.

Alternatively, the programmer can develop program transformations that address implementation issues, and structure the transformations into program derivations. This approach has the benefit of adding formality to the construction process, helping to improve the chances of the implementation being correct and perhaps permitting formal analysis of the construction process. This approach does though have the disadvantage of being very circumspect, so that if applied manually, it can be very tedious and labourious.

However, if derivation is performed automatically, then these disadvantages are irrelevant. Thus, in this thesis, the method I use to construct implementations from functional specifications is to apply fully automatic program derivations directly to the specifications.

3.6 Chapter Notes

i (page 35)

It may be, and indeed has been, argued that an assessment of an implementation based solely upon execution performance is inadequate, as the assessment fails to take into account how much programmer time and effort is required to produce and maintain an implementation: one implementation may execute in half the time of another, but may also have taken ten times as long to develop. In addition, the issue of reliability or correctness of an implementation should also be considered: after all, it does not matter how quickly an implementation executes if the results it produces are incorrect.

However, such arguments seem to hold little weight with those who make extensive use of the types of algorithms discussed in this thesis.

ii (page 38)

Of course, using a machine level language would provide an even finer degree of control the implementation system, and there are occasions when a hand-crafted machine level implementation is considerably more efficient than a compiler Fortran implementation. However, Fortran provides a much more convenient programming model than machine language, while *in general* not introducing considerable overheads.

iii (page 38)

Indeed, it seems that often the implementation of an algorithm is undertaken with *no* precise definition of the algorithm, only a general idea of the process carried out the the algorithm; the programmer completes the details of the process as the implementation develops. Such a method of implementation is not necessarily a bad method, especially if the person who specified the algorithm is the same person who is implementing the algorithm: its merit depends on just what aspects are omitted from the initial understanding; some aspects are indeed relatively unimportant and any reasonable implementation of those aspects will suffice.

However, it must then be asked whether the construction of an implementation is part of computer *science*, or whether it is a craft.

Chapter 4

Derivations and Transformations

For the work reported in this thesis, implementations are derived by applying transformations using the TAMPR transformation system. Transformations are the basic tool of this system: a single transformation typically causes some simple, local change to a program. More complex changes are effected by grouping transformations into transformation sequences, and grouping transformation sequences into derivations.

In this chapter, I discuss general aspects of the TAMPR system and the notation I will use in the rest of this thesis to denote transformations. I also discuss the structuring of transformations into derivations. The purpose of this chapter is to provide some familiarity with the nature of transformations and derivations and with the capabilities of the transformation system; this chapter is not intended as a reference manual for TAMPR though.

The discussion is as follows:

- A very brief overview introduction to the TAMPR system.
- TAMPR transformations.
- Transformation sequences.
- Derivations.

4.1 The TAMPR Transformation System

The TAMPR transformation system [17, 18, 19, 20, 22, 21, 23] was developed by Boyle at Argonne National Laboratory. One of the earliest applications of the system was for the automatic construction of families of numerical routines, members of the families being designed for single- or double-precision floating point operations, and real or complex arithmetic. It has also been used to construct Fortran implementations from Lisp specifications for sequential, vector and multiprocessor systems.

4.2 TAMPR Transformations

A TAMPR transformation is a formal rewrite rule, consisting of a *pattern* and a *replacement*. For example, the transformation

```
.sd.
  x+0 ==> x
.sc.
```

has *pattern* `x+0` and *replacement* `x`. (The symbols `.sd.` and `.sc.` designate the start and end of the transformation: they, and other symbols enclosed in periods, are markers for the transformation system rather than parts of the pattern or replacement. `.sd.` stands for "structural definition" and `.sc.` for "structural change".) When a transformation is applied to a program by the TAMPR system, sections of the program that match the pattern are substituted by the replacement. Application of transformations is entirely automatic.

TAMPR transformations are defined in a formal, wide-spectrum grammar: i.e. a grammar designed to permit the description of constructs that occur in a wide range of languages (Fortran, Pascal, C, Lisp, SML, etc.). A wide-spectrum grammar is useful when transforming between two radically different languages (such as from pure functional to imperative) as it permits programs to (temporarily) exist during transformation in a form that conforms to neither the source nor target language, but that has aspects of both.

All of the work reported in this thesis uses formal TAMPR transformations, but for clarity transformations are presented in an informal style, as the idea underlying the transformation is more important than the precise syntactic details required for an automated system. One danger in abandoning formal notation is the possibility of using some form that is beyond the abilities of the transformation system; a form that, for example, requires human reasoning to apply. To provide some understanding of the capabilities of the TAMPR system, I discuss here the forms of transformations commonly used, presenting both the informal and formal TAMPR forms.

4.2.1 Basic Transformations

The most basic form of transformation consists of a pattern and a replacement expressed using literal strings and both terminal and non-terminal symbols. For example, the transformation

```
.sd.
  <entity>"1"+<entity>"1" ==> 2*<entity>"1"
.sc.
```

has pattern `<entity>"1"+<entity>"1"`, in which `<entity>` is a non-terminal symbol in the language being transformed roughly corresponding to any expression which does not contain operators and in which `+` is a terminal symbol. The label "1" on the `<entity>`s indicates that both `<entity>`s must match the same expression for the pattern as a whole to match.

So the pattern matches expressions such as

```
1+1
```

(with `<entity>"1"` matching `1`) and

```
f(g(x))+f(g(x))
```

(with `<entity>"1"` matching `f(g(x))`). The pattern does *not* match the expression `1+2` since the two expressions `1` and `2` are different.

The replacement in the above transformation is `2*<entity>"1"`, in which `<entity>"1"` represents whatever expression was matched by `<entity>"1"` in the pattern. Thus, the above expressions would be transformed as

```
1+1 ==> 2*1
f(g(x))+f(g(x)) ==> 2*f(g(x))
```

As a second example, consider the transformation

```
.sd.
  <entity>"1"+<entity>"2" ==> plus(<entity>"1",<entity>"2")
.sc.
```

Since the labels on the `<entity>`s in the pattern are different, there is no requirement for the expressions matched to be the same, though they may be. Thus, `x+y` would be transformed into `plus(x,y)`, and `x+x` into `plus(x,x)`.

For most of this thesis, the general idea involved in a transformation is more important that the precise syntactic form, so transformations will normally be presented in an informal style, with letters representing non-terminal symbols. For example, the two transformations given above may be written:

```
.sd.
  e+e ==> 2*e  where e is an expression
.sc.
.sd.
  e1+e2 ==> plus(e1,e2)  where e1,e2 are expressions.
.sc.
```

Further, qualifications such as 'where e is an expression' will often be omitted when implied by context.

4.2.2 Additional Forms of Transformations

The TAMPR transformation system provides several more advanced types of transformations, the most useful of which are discussed briefly below.

Qualified Patterns

A pattern may be qualified by a second pattern, indicated as a pattern in braces: a qualified pattern matches only if both patterns match. For example, consider the transformation

```
.sd.
  <expr>"1"{<entity>+<entity>}+<entity>"1"
  ==>
  plus(<expr>"1",<entity>"1")
.sc.
```

where the `<expr>` non-terminal corresponds to an arbitrary expression. Here, `<expr>"1"` will match an expression only if that expression is the addition of two `<entity>`s.[1] Thus, the expression `1+2+3` would be transformed into `plus(1+2,3)`.

Indefinites

It is frequently useful to test if a pattern occurs anywhere inside another pattern; for example, to test if an expression contains a particular identifier. An indefinite pattern can be used to detect such cases. An indefinite pattern is indicated using question marks ("?"): for example, the pattern

```
lambda <ident>"1" @ <entity>"1" { ? <ident>"1" ? } end.
```

where the form `lambda ... @ expression end` is the notation used for λ-expressions, will match a λ-expression in which the body expression contains the bound identifier.

The indefinite pattern is `{ ? <ident>"1" ? }`; it qualifies the pattern `<entity>"1"`. The two combined may be read as "entity 1 of the form: an entity containing ident 1".

Thus, the full pattern would match

```
lambda x @ g(f(x,y)) end
```

but would not match

```
lambda x @ g(2) end.
```

If a transformation has a pattern containing an indefinite pattern, the replacement may also contain an indefinite pattern of the same structure. A question mark in the replacement may be thought of as representing whatever was matched by the corresponding question mark in the pattern. For example, the transformation

```
.sd.
  lambda <ident>"1" @ <entity>"1" { ? <ident>"1" ? } end (<ident>"2")
  ==>
  lambda <ident>"1" @ <entity>"1" { ? <ident>"2" ? } end (<ident>"2")
.sc.
```

in which the bound identifier of a λ-expression is bound to a second identifier, would replace one instance of the bound identifier with that second identifier. Thus, the expression

```
lambda x @ g(f(x,y)) end (y)
```

would be transformed into

```
lambda x @ g(f(y,y)) end (y).
```

In informal notation, indefinites, and qualified patterns in general, will normally be indicated using English comments. For example, the pattern

```
lambda <ident>"1" @ <entity>"1" { ? <ident>"1" ? } end
```

may be written informally as

$\lambda \mathtt{x} \cdot \mathtt{e}$ where e contains x.

[1]Although `<expr>"1"` and `<entity>"1"` have the same label, `"1"`, there is no requirement for them to match the same expression, since they designate different syntactic classes.

Exclusions

It can be useful to apply a transformation only when a pattern *fails* to match; for example, when a λ-expression has a body expression which does *not* contain any instances of the bound identifier. An exclusion pattern can be used to specify that a section of a program must fail to match a specified pattern. Exclusions are indicated using the `.differing from.` marker. For example, the pattern

```
<expr>"1" .differing from. {<entity>"1"+<entity>"2"}
```

would match any expression that was not the addition of two `<entity>`s. For example, it would match `2*3`, but would not match `2+3`.

Matching a λ-expression with a body that does not contain the bound identifier may be performed by combining an exclusion with an indefinite pattern. So the transformation

```
.sd.
  lambda <ident>"1"
    @ <expr>"1" .differing from. { <expr> { ? <ident>"1" ? } }
    end (<expr>)
  ==>
  <expr>"1"
.sc.
```

would reduce such a λ-expression into just the body expression. The pattern of this transformation may be interpreted as follows:

- The bound identifier of the λ-expression is matched by `<ident>"1"`.
- `<expr>"1"` matches the body expression, which must *not* be of the form: an expression containing `<ident>"1"`.
- The bound value is matched by the final `<expr>`. This pattern does not have a label since it is not used in the replacement and is not required elsewhere in the pattern. (i.e. it does not need to be correlated with any other pattern in the transformation.)

Exclusions will normally be indicated in informal transformations using comments. For example, the above transformation may be written

```
.sd.
  λx·B (e) ==> B  where B does not contain x.
.sc.
```

Sub-transformations

Many transformations are such that they should only be applied in a certain context. In such cases, the context is provided by defining a main transformation (as explained above) and applying a *sub-transformation* to the replacement of the main transformation. For example, the β-reduction of a λ-expression (in the absence of name-clashes) can be performed by replacing with the bound value all instances of the bound identifier in the body of the expression. A transformation to perform this is:

```
.sd.
  lambda <ident>"1" @ <entity>"1" end (<entity>"2")
  ==>
  <entity>"1" { .sd. <ident>"1" ==> <entity>"2" .sc. }
.sc.
```

When this transformation is applied, a λ-expression is replaced with its body expression, modified by applying the sub-transformation `<ident>"1" ==> <entity>"2"`.

Informally, this transformation may be written:

```
.sd.
  λx·e1 (e2) ==> e1 { .sd. x ==> e2 .sc. }
.sc.
```

though in the case of β-reduction, standard λ-calculus notation is used:

```
.sd.
  λx·e1 (e2) ==> e1 [e2/x]
.sc.
```

Provisional Transformations

Sometimes the simplest way to detect if a transformation should be applied to a program section, is to provisionally apply the transformation and analyse the result, by matching it against some pattern (such as the original program section).

For example, consider again the example of a λ-expression in which the body does not contain the bound identifier. Such an expression can be detected by replacing all occurrences of the bound identifier in the body expression and then comparing with the original body expression; if the two are the same, then the body cannot have contained the bound identifier.

This process can be performed by TAMPR using a provisional transformation, or *where clause*:

```
.sd.
  lambda <ident>"1" @ <expr>"1" end (<expr>"2")
  .where.
  {
    <expr>"1" { .sd. <ident>"1" ==> <expr>"2" .sc. }
    .is matched by.
    <expr>"1"
  }
  ==>
  <expr>"1"
.sc.
```

The *where clause* is introduced by the `.where.` marker and consists of everything in braces following the marker. The clause applies a transformation to some program section (in this example, `<expr>"1"`) and then checks if the result can be matched by a pattern specified after the `.is matched by.` marker (in this example, the pattern is `<expr>"1"`, but in general the pattern can be any valid TAMPR pattern). If that match succeeds, the main transformation is applied (replacing the entire λ-expression with just `<expr>"1"`); if the match fails, the main transformation is *not* applied, and the transformation as a whole is considered not to have been applied. The reasoning underlying this example is: if all occurrences of the bound identifier are replaced by the bound value, and there is no change, then the λ-binding is redundant.

Summary

The forms of transformations discussed above constitute almost the entire repertoire of the transformation language. All the transformations used in the work reported in this thesis were constructed using only (combinations of) these forms.

4.3 Transformation Sequences

Often, several transformations are designed to be applied together, either because they are closely related conceptually or because they form a mutually recursive set (i.e. the output of one transformation in the set can be matched by the pattern of other transformations in the set).

In particular, many of the constructs in the grammar are defined recursively and transformations for manipulating recursive constructs naturally form 'base case'/'recursive case' pairs. For example, in the wide-spectrum grammar used for this thesis, the non-terminal `<expr list>` is a comma-separated list of one or more `<expr>`s, defined as

```
<expr list> ::= <expr list>, <expr>
             |   <expr>
```

Manipulation of `<expr list>`s thus often requires two transformations: one transformation for processing an `<expr list>` that is a single `<expr>`; and a second transformation for processing longer `<expr list>`s.

In such cases, the transformations can be grouped together as a *transformation sequence*, indicated by enclosing transformations in braces. The application of a transformation sequence is performed by exhaustively applying each transformation in the sequence until none of the transformations can be applied.[2]

For example, the following transformation sequence performs β-reduction on multiple-argument λ-expressions (assuming no name-clashes and assuming no λ-expression is partially applied[3]):

```
{
.sd.
  lambda <ident>"1" @ <expr>"1" end (<expr>"2")
  ==>
  <expr>"1" { .sd. <ident>"1" ==> <expr>"2" .sc. }
.sc.

.sd.
  lambda <expr list>"1", <ident>"1"
  @ <expr>"1"
  end (<expr list>"2", <expr>"2")
  ==>
  lambda <expr list>"1"
  @ lambda <ident>"1" @ <expr>"1" end (<expr>"2")
  end (<expr list>"2")
.sc.
}
```

The first transformation applies to a 'base case', where the λ-expression has a single bound identifier. The second transformation applies to 'recursive cases', converting a λ-expression with multiple bound identifiers into a nest of λ-expressions, each with a single bound identifier, by stripping off the trailing `<ident>` and `<expr>` into a new λ-expression.

These two transformations are closely related and also form a mutually recursive set: an application of the second clearly creates an expression that can be transformed by the first transformation and possibly the second; and if one of the bound values is an open applied λ-expression, the first transformation may create an expression that can be transformed by either the first or the second. For example:

[2] The transformations form a sequence rather than a set because their order is important, as discussed in the following section.

[3] A function is partially applied when the number of actual arguments is less than the number of formal arguments.

```
λx,f·f(2,x) (a,λy,z·y*z)
==>  by the second transformation
λx·λf·f(2,x) (λy,z·y*z) (a)
==>  by the first transformation
λx·λy,z·y*z (2,x) (a)
==>  by the second transformation
λx·λy·λz·y*z (x) (2) (a)
==>  by the first transformation
λx·λy·y*x (2) (a)
==>  by the first transformation
λx·2*x (a)
==>  by the first transformation
2*a
```

4.3.1 Traversal Order

There are some occasions when it is important to consider the application order of the transformations which constitute a transformation sequence.

A program is parsed into a tree according to a wide-spectrum grammar. The TAMPR system iterates over this tree, applying a transformation sequence. Applying a transformation sequence implies applying each transformation in the sequence, in the order that they occur: the first transformation whose pattern matches is the one that is successfully applied. What happens once a transformation has been successfully applied depends on the *traversal order*.

The traversal order is one of three basic modes designated as `*`, `1` and `down`.

`*` In `*` mode, TAMPR starts at the bottom of the parse tree and attempts to apply the transformation sequence. If application fails, it moves up the tree, and continues moving up until application succeeds, when traversal begins again at the bottom of the tree. Application terminates only when TAMPR makes a complete traversal of the tree without applying any transformation in the sequence: thus, on termination, it is guaranteed that no part of the program is matched by any of the patterns in the transformation sequence.

`1` In `1` mode, TAMPR behaves as in `*` mode, except that when a transformation is applied, TAMPR attempts to re-apply the transformation sequence at the same level in the parse tree, rather than restarting at the bottom of the tree.

`down` The `down` mode is similar to `1` mode, except TAMPR begins at the top of the tree and moves down when application of the transformation sequence fails.

The `1` and `down` modes can be further qualified at the individual transformation level: a transformation can be marked to indicate that when it has been applied successfully, TAMPR should move to the next level of the tree, rather than re-applying at the same level.

Most transformation sequences use the `*` mode, to ensure that all occurrences of a pattern in a program are transformed. Transformation sequences can always be written so that `*` mode can be used, but sometimes termination of the application process is more easily obtained by using mode `1` and, rarely, the modified mode. Mode `down` is used when it is important to match outer occurrences of a pattern before inner occurrences.

4.4 Summary of Transformations

The basic points about transformations may be summarized as follows:

- A transformation is a rewrite rule, consisting of a pattern and a replacement.
- The pattern and replacement describe program sections using syntactic classes drawn from a wide-spectrum grammar.
- Transformations to be applied by TAMPR are grouped into transformation sequences.
- TAMPR applies each transformation in the sequence in order.
- Application is usually exhaustive.

When developing transformations, it is important to remember that the TAMPR system is essentially a tree manipulator with powerful tree matching abilities; it is not a theorem prover, so any ‘reasoning’ that is to be performed about a program must be represented explicitly in the program (using some notational convention). Transformation sequences are often designed first to make explicit some information that is implicit in a program, or to recast information into a form more readily manipulated; then to make use of the information; and maybe to remove the information if it is no longer required.

4.5 Derivations

A transformation sequence is normally designed to perform some specific, relatively simple task, such as β-reduction in the absence of name-clashes. Larger tasks are accomplished by applying a sequence of transformation sequences, called a *derivation*.

For example, a more useful form of β-reduction than that previously discussed would allow for name-clashes. Also, for some purposes, it would be useful if β-reduction were more selective: for example, expressions involving applications of program defined functions should normally not be reduced as reduction often results in an unacceptable increase in the size and computational complexity of a program. Thus, practical β-reduction might be performed as follows:

Mark applications of program defined functions
: After this stage, expressions that should be reduced are distinguished from expressions that should not be be reduced.

Perform α-conversion
: After this stage, there are no name-clashes.

β-reduce
: Reduce all applied λ-expressions in which the bound value does not contain the mark introduced by the first step

Remove marks
: After this stage, all applied λ-expressions remaining have bound values containing (potentially) computationally expensive expressions.

Each of these steps may correspond to one or more transformation sequences. A program developer constructs the derivation by listing the transformation sequences in the order in which they are to be applied; the transformation system applies the derivation by applying each transformation sequence once, in turn; that is, each transformation sequence is applied once, though the transformations within each sequence are applied exhaustively.

Thus, when constructing a derivation to perform some task, it is necessary to split the task into a sequence of steps, each with a well-defined objective that can be implemented as one or more transformation sequences. Many of the steps will be designed to simplify some main step by enforcing some convention or by introducing some local notation that makes explicit information required for the main step.

4.5.1 Sub-derivations and Intermediate Forms

If a derivation is to achieve some radical change in a program, then it may be convenient to consider a derivation as a sequence of *sub-derivations*, each of which is a sequence of transformation sequences. The overall change is then decomposed into a sequence of simpler changes: the program progresses from the initial form through a sequence of *intermediate forms* to the final form.

As an example of a fairly complex derivation, I will here consider the conversion of a simple program in a functional form (the λ-calculus) into an equivalent Fortran program. The discussion is based upon [20], which gives a detailed justification for the approach.

For simplicity, I assume that the only data types are integer and boolean, and that booleans occur only in the predicates of conditionals. As an example program, I use factorial:

```
fun fac = λn·if (eq(n,0)) then 1 else times(n,fac(difference(n,1)))
fun init = λN·λf·write(f) (fac(N)) (read_integer).
```

The transition from referentially transparent, recursive, expression-based λ-calculus to state-based, non-recursive Fortran is quite a complex problem. To simplify the problem, the transition is performed in four main stages by sub-derivations, each of which is implemented as a sequence of transformation sequences.

Each sub-derivation creates an intermediate form for a program:

- **Recursive 'Fortran'**
  The transition is made from an expression basis to a state basis: i.e. variables are introduced. After this stage of the derivation, the program is imperative in nature. However, functions may be applied recursively, contrary to the Fortran77 standard.

- **Recursive parameterless Fortran**
  Recursive functions are converted into recursive parameterless subroutines by introducing a stack for function arguments, local variables and return values.

- **Non-recursive Fortran**
  Recursive subroutines are converted into non-recursive routines by introducing `GOTO`s to direct execution. The subroutine structure now serves no purpose, so the subroutines are concatenated to form a single program unit.

- **Standard Fortran77**
  Functional notation is converted into Fortran notation and various aspects are 'cleaned up'.

The stages are described in detail below.

Recursive Fortran

Consider a λ-calculus function such as

```
fun fac = λn·if (eq(n,0)) then 1 else times(n,fac(difference(n,1)))
```

In a Fortran function, the function identifier is treated as a variable which stores the return value of the function. Thus, a pseudo-Fortran equivalent to the λ-calculus function is:

```
function fac(n)
fac = λn·if (eq(n,0)) then 1 else times(n,fac(difference(n,1)))
end
```

Fortran does not have a construct for conditional expressions so the assignment to `fac` is implemented as a conditional statement:

```
function fac(n)
if (eq(n,0))
then fac = 1
else fac = times(n,fac(difference(n,1)))
endif
end
```

Expressions such as conditional expressions and applied λ-expressions are referred to here as *non-primitive* expressions. In general, a non-primitive expression is implemented as a statement sequence in Fortran, with variables introduced to store the value of the expression. Implementing non-primitive expressions is the main issue in this stage of the derivation.

The derivation proceeds as follows:

- Non-primitive expressions are identified by introducing identity λ-expressions; that is, λ-expressions of the form λx·x (e).
- All λ-expressions (not just identity λ-expressions) are propagated out of primitive expressions; for example:

  ```
  .sd.
    times(λx·x (e),3) ==> λx·times(x,3) (e).
  .sc.
  ```

  Any identity λ-expressions remaining after propagation are reduced.

  The purpose of this propagation is to ensure that no non-primitive expression occurs as an argument to a primitive function, so that later transformations need only examine the 'outer' level of expressions in assignments to convert non-primitive expressions.
- The Fortran convention of assigning the return value of a function to the function identifier is introduced.
- If the right-side of an assignment is a non-primitive expression, it is implemented appropriately: a conditional expression is implemented as a conditional statement; a λ-expression is implemented as a pair of assignments (the first to assign the bound value to the bound identifier; the second to assign the body of the λ-expression).

The form of the `fac` function after this stage of the derivation is shown in figure 4.1. The `block ... end` construct allows variables to be introduced at any point in a program.

Recursive Parameterless Fortran

The main implementation issue in this stage of the derivation is converting recursive functions into recursive parameterless subroutines by introducing a global stack for the formal arguments to functions, local variables and return values.

The derivation proceeds as follows:

```
==>
integer function fac(n)
if (eq(n,0)) then
  fac =1
else
  block
    integer t2
    t2 = fac(difference(n,1))
    fac = times(n,t2)
  end
endif
end
```

Figure 4.1: Recursive Fortran form of the factorial function

- Local blocks are removed by moving all declarations to the top of function definitions. This simplifies the replacement of local variables by stack components.
- The formal arguments of a function are removed and the function converted into a subroutine. Instances of the arguments in the function's body are replaced with references to the stack.
- The local variables of a function are replaced with stack references.
- Assignments to the function identifier (which designates the function's return value) are replaced with assignments to the appropriate component of the stack.
- Applications of program defined functions are replaced with subroutine calls, with arguments placed on the stack before the call and the result removed from the stack after.

```
==>
subroutine fac
if (stack(p-2)=0) then
  stack(p) = 1
else
  stack(p+1+1) = difference(stack(p-2),1)
  p = p+1+1+2
  call fac
  p = p-1-1-2
  stack(p+1) = stack(p+1+1+2)
  stack(p) = times(stack(p-2),stack(p+1))
endif
end
```

Figure 4.2: Derivation to recursive parameterless Fortran of the factorial function

Figure 4.2 shows the `fac` function at the end of this stage of the derivation.

Non-recursive Fortran

The main implementation issue remaining is the removal of recursive subroutines. This removal is achieved by replacing each subroutine call with a statement that places a return address on the stack and a `GOTO` which transfers execution to the start of the body of the subroutine; on completion of the subroutine, the return address is popped off the stack and used in a computed `GOTO` to return execution to the point after the subroutine call. (A computed `GOTO` has the form `GOTO v` where `v` is a variable which stores a statement label; execution is transferred to the statement marked by the label. A statement label `L` is assigned to a variable `v` using the command `assign L to v`.)

The derivation is as follows:

```
==>
    subroutine init
    read(stack(p))
    p = p+1+1
    assign 1 to stack(p-1)
    goto 2
1   continue
    p=p-1-1
    write(stack(p+1+1))
    goto 100

2   continue
    if (stack(p-2)=0) then
    stack(p) = 1
    else
    stack(p+1+1) = stack(p-2)-1
    p = p+1+1+2
    assign 3 to stack(p)
    goto 2
3   continue
    p = p-1-1-2
    stack(p+1) = stack(p+1+1+2)
    stack(p) = times(stack(p-2),(stack(p+1))
    endif
    goto stack(p-1)
100 continue
    end
```

Figure 4.3: Derivation to non-recursive Fortran of factorial function

- Mark the start of each subroutine with a statement label.
- Replace each subroutine call with a `GOTO` to the appropriate label, marking the point after the call with a label and placing that label on the stack.
- At the end of each subroutine, insert a computed `GOTO` that returns execution to the point in the program marked by the statement label on the stack (placed there by the previous step).
- Since the scope of a statement label is restricted to the program unit (subroutine) in which it is defined, the program units must be concatenated into a single unit for the `GOTO`s to be valid.

Figure 4.3 shows the program after this stage of the derivation.

Standard Fortran77

The final stage of the derivation performs some ‘cleaning up’; for example:

- declaring the stack and stack pointer and converting instances of the `IF-THEN-ELSE-ENDIF` control construct into `GOTO`s (so removing the jumps into and out of conditional statements created in the previous stage; some people find such jumps disturbing);
- converting some notation to ensure compatibility with the Fortran77 standard — such as converting prefix functions into infix operators and converting applications of the `element` function into Fortran’s bracket notation.

This stage of the derivation is straightforward, so I do not discuss it in detail. The final program is shown in figure 4.4.

```
==>
    program init
    parameter(stacksz=10000)
    integer stack(stacksz)
    integer p
    p=0
    read(stack(p+1))
    p = p+1+1
    assign 1 to stack(p-1)
    goto 2
1   continue
    p=p-1-1
    write(stack(p+1+1))
    goto 100

2   continue
    if (stack(p-2)=0) then
    stack(p) = 1
    goto 4
    endif
    stack(p+1+1) = stack(p-2)-1
    p = p+1+1+2
    assign 3 to stack(p)
    goto 2
3   continue
    p = p-1-1-2
    stack(p+1) = stack(p+1+1+2)
    stack(p) = stack(p-2)*stack(p+1)
4   continue
    goto stack(p-1)
100 continue
    end
```

Figure 4.4: Final Fortran77 implementation of the factorial function

4.6 Summary

A complex derivation from some initial form to some final form can be structured into a sequence of sub-derivations; each sub-derivation creates an intermediate form, with the sequence of forms leading gradually from initial to final form. It is the identification of a suitable sequence of intermediate forms that may be viewed as the most important aspect of derivations. Each sub-derivation may be further structured into low-level sub-derivations, each of which creates a finer level of intermediate form.

Ultimately, a sub-derivation is structured as a sequence of transformation sequences. Each transformation sequence typically performs some well-defined task required for the transition from one program form to the next.

A transformation sequence consists of one or more transformations which perform the actual rewriting of the program. Each transformation is typically quite simple. The power of derivations lies in the transformation system performing many applications of simple transformations, and in the structuring of transformations into transformation sequences, sub-derivations and derivations.

Chapter 5

AMT DAP 510 Derivation

In previous chapters I have discussed the principles of functional specification and program derivation. In this chapter, I demonstrate the application of these principles to the derivation of implementations for the AMT DAP 510 array processor. I use matrix-vector multiplication as an example.

The discussion is as follows:

- The functional specification in SML of matrix-vector multiplication.
- A description of the target environment: the AMT DAP 510 array processor, programmed using Fortran Plus Enhanced.
- A preliminary derivation upon which the SML to Fortran Plus Enanced derivation is based.
- Extension of this preliminary derivation.

5.1 Introduction

In previous chapters I have discussed the general principles of functional specification and program derivation. Functional specifications may be of value in themselves, as a means for formally expressing and investigating algorithms, but I believe their value would be greatly diminished if it were not possible to derive efficient implementations from such specifications. Whether or not this is possible in general is still an open question.

A demonstration of feasibility was performed by Boyle and Harmer [22], who derived an efficient implementation for a CRAY vector system of a Cellular Automata algorithm from a Lisp specification. In this chapter, I discuss a derivation for a rather different architecture — an AMT DAP 510 array processor — and using a different specification language.

The DAP was chosen as the target machine for several reasons

- An efficient DAP implementation of an algorithm must use operations that are very specific to the architecture; given that a specification is independent of architecture considerations, the production of an efficient implementation is not trivial.
- The data-parallel subsets of the dialects of Fortran used to program the DAP provide programming models that are well-defined and well-matched to the architecture. (In contrast, the programming model provided by Fortran77 is both ill-defined and ill-suited to many architectures for which it is used, such as the CRAY vector architecture.)
- Array processors are commonly used by numerical mathematicians. In addition, programming models, particularly the so-called Single Program Multiple Datastream (SPMD) model, similar to the whole-array programming model used for the DAP are being employed for other architectures (through languages such as Fortran90 and High Performance Fortran; I anticipate that the derivation developed for the DAP will be suitable for extension to those languages).
- The POT algorithm (defined earlier) had been implemented (manually) on the DAP; the hand-coded implementation provided a measure for the efficiency of any derived implementations.

In summary, the DAP is a system that is of practical importance to practitioners of numerical mathematics and that provides an interesting and challenging target for derivation.

The functional specification of the POT and Conjugate Gradients algorithms were discussed earlier; the derivation of implementations of those algorithms for the DAP is discussed in the following chapter. However, those algorithms are rather too complex to serve as introductory examples for the DAP derivation, so instead I use matrix-vector multiplication:

- Matrix-vector multiplication is ubiquitous in numerical mathematics: its efficient implementation is central to the efficient implementation of many linear algebra algorithms.
- The derivation of matrix-vector multiplication illustrates many of the principles required for larger examples, while still being concise enough to be easily followed.

Thus, in this chapter I discuss in detail each stage of the derivation that produces implementations for the DAP from SML specifications. To illustrate the derivation, I trace the derivation of a DAP implementation of matrix-vector multiplication from an architecture independent SML specification.

5.2 Functional Specification of Matrix-Vector Multiplication

The multiplication of a matrix of real numbers and a vector of real numbers is a commonly used operation, so it is provided in a library of routines that can be used by a programmer in ignorance of the details of their implementation. For illustrative purposes, the definition of matrix-vector multiplication is shown in figure 5.1, along with several functions required by the definition.

```
⋮
structure realarray = struct
  open array;

  fun times(A:real array,B:real array):real array
    = map(A,B,real.times);
  fun sum(A:real array):real
    = fold(A,real.plus,0.0);
  fun innerproduct(A:real array,B:real array):real
    = sum(times(A,B));
  fun mvmult(A:real matrix,V:real vector):real vector
    = let
      val m:int = dim(A,0)
    in
      generate(m,fn(i:int)=>innerproduct(row(A,i),V))
    end
  fun multiply(A:real array,B:real array):real array
    = let
      val ra:int = shape.rank(A);
      val rb:int = shape.rank(B)
    in
      if (ra=1 andalso rb=2) then vmmult(A,B)
      else if (ra=2 andalso rb=1) then mvmult(A,B)
      else mmmult(A,B)
    end;
⋮
end;

⋮
structure op = struct
  fun eq
    = overload(integer.eq,real.eq,bool.eq);
  fun times
    = overload(integer.times,real.times,realarray.multiply);
end;

open op;

⋮

(*An example of using matrix-vector mult.*)
fun init(n:int):real vector
  = let
    val A:real matrix = read_matrix([n,n]);
    val V:real vector = read_vector([n])
  in
    A*V
  end;
```

Figure 5.1: SML specification of matrix-vector multiplication

- The functional specification defines a value, that value computed by the 'driver' function `init`. The value computed is the product of matrix `A` and vector `V`, where the values of `A` and `V` are unknown until program execution (as indicated by the `read_vector` and `read_matrix` functions).

- The product of A and V is denoted as A*V. The * operator is the infix equivalent to the prefix function op.times (that is, the function times defined in the structure op).
- In the op structure, the function times is defined as being overloaded onto the functions[1]

  integer.times which performs integer multiplication;

  real.times which performs multiplication of real numbers;

  realarray.multiply which performs matrix-vector or matrix-matrix multiplication according to the dimensionalities of its arguments.

  Thus, the function name op.times represents several functions: the overloading is resolved (i.e. which of the overloaded functions is to be used is determined) using type information.
- The important component of this overloading for this example is realarray.multiply, defined in the realarray structure. This structure is based upon the array structure (as indicated by the open array statement at the beginning of the structure): that is, all the types and functions defined in the array structure also form part of the realarray structure. The latter does though define additional functions suited only to arrays of real numbers.
- The multiply function determines whether to perform matrix-vector, matrix-matrix or vector-matrix multiplication based on the dimensions of its arguments.[2] For matrix-vector multiplication, the function mvmult is used; this function defines the product as a vector of length equal to the number of rows in the matrix (dim(A,0)) with each element given by the inner-product of a row of the matrix with the vector argument.
- The remaining functions are a straightforward definition of inner-product as the sum of the elementwise product of two vectors.

It is important to remember that a programmer would normally be completely unaware of the details of these structures: he would simply use the * operator to perform various types of multiplication.

The specification of MVM is an abstract specification: it is independent of any particular computational architecture and is intended to be simple to understand.

5.3 Target Environment: Fortran Plus Enhanced on an AMT DAP 510

The AMT DAP 510 is a Single Instruction Multiple Datastream (SIMD) system, consists of a 32 by 32 grid of processing elements controlled by a single master processor (see figure 5.2).

The master processor — essentially a conventional 32 bit processor with some additional components for controlling the operations of the processing elements — performs most of the scalar calculations. The processing elements, which are single bit processors, perform the parallel processing operations. The master processor issues instructions to the processing elements, which all obey the instruction simultaneously. The master processor may also issue data to the processing elements.

Each processing element has its own local memory to which it has direct access; no processing element has direct access to the memory of any other processing element. In general, in a given

[1]These are the functions relevant to this example; the overloading is actually somewhat more extensive, including, for example, the multiplication of an array by a scalar.

[2]Overloading cannot be used here as the types vector and matrix are indistinguishable: they are both abbreviations for array.

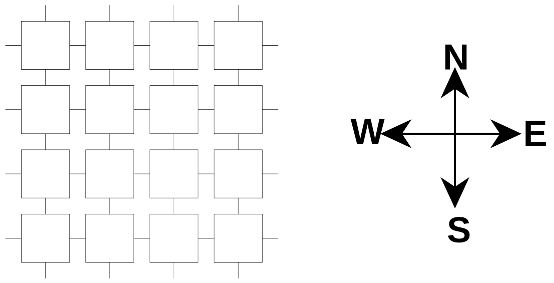


Figure 5.2: DAP processor grid

operation, all processing element access the same component of their respective memories. Thus, the memory of all the processing elements may be thought of as consisting of a sequence of planes, the k^{th} plane being the collection of the k^{th} component of each processor's memory; and the processor grid can be thought of as performing operations on these memory planes. (See figure 5.3.)

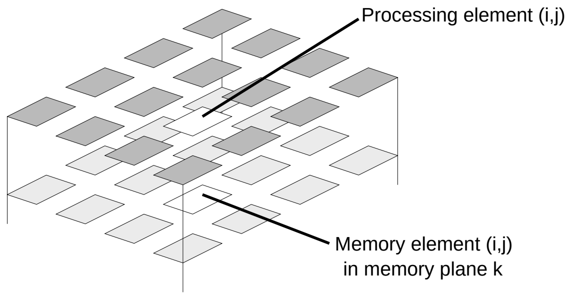
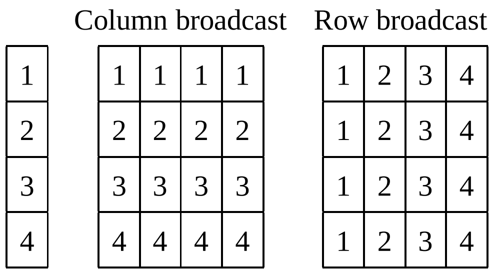


Figure 5.3: DAP memory planes

When a processing element requires a value stored in the memory of another element, it must obtain the value through a communication mechanism: each processing element is connected to its four nearest neighbours in the grid, an element on an edge being connected to the corresponding element on the opposite edge (directions on the grid are designated as north, south,east and west); all the processing elements can simultaneously obtain a value from one neighbour, though the direction of the neighbour is the same across the entire grid.

In addition to the nearest neighbour connections, the DAP hardware supports three broadcast mechanisms which can be used to duplicate values across the grid. A single scalar value can be broadcast to each processing element, or a set of 32 scalar values (called a vector) can be broadcast to each row or to each column of the grid. (See figure 5.4.)



1
2
3
4

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

Figure 5.4: DAP broadcasting to rows and columns

Although the processing elements are single bit processors, the hardware supports certain operations for manipulating data when each datum is represented using more than one bit. Such data are stored in memory in either vertical or horizontal mode: in vertical mode, successive bits of a datum are stored in successive memory planes – each plane thus contains one bit for each of 32 by 32 components of the data; in horizontal mode, (up to 32) successive bits of a datum are stored across a row of a memory plane – each plane thus contains bits for up to 32 components of data.

The processor grid can perform parallel operations utilizing both these storage modes. For example, in horizontal mode, the processor grid or a memory plane can be interpreted as containing 32 integer values (where each integer is represented using 32 bits). The processor grid can perform in parallel 32 additions between corresponding integers in the processor grid and a memory plane. Similarly, in vertical mode, the processor grid or a memory plane can be interpreted as representing 32 by 32 boolean values (where each boolean is represented using 1 bit). For example, the hardware can perform 32 by 32 conjunctions in parallel.

Associated with each processing element is an activity register which controls whether or not the element participates in certain operations. The activity mask (that is, the grid of 32 by 32 activity registers) can be set under program control and can thus be used to implement conditional operations.

The DAP hardware also supports reduction operations (such as summation and conjunction) over the entire processor grid, and along only the rows or columns (to produce a vector of values).

5.3.1 The DAP Software

The DAP can be programmed in several languages:

DAP assembler The machine level language. Since the processing elements are single bit processors, using solely DAP assembler would require manipulation at extremely low levels.

Fortran Plus A superset of Fortran77 with support for data-parallel operations on one and two dimensional arrays *of size equal to the size of the grid of processing elements*: that is, a vector of size 32 or a matrix of size 32 by 32.

Fortran Plus Enhanced Similar to Fortran Plus, but allows vectors and matrices to be any size (restricted only by available memory). The compiler automatically *sections* large arrays: i.e. it divides an array that is larger (in either dimension) than the processor grid into regions the same size as the processor grid.

For this thesis, I chose to use Fortran Plus Enhanced as it provides the most convenient programming model, permitting vectors and matrices of any size are processed in a data-parallel manner. An investigation reported in [26] showed that while Fortran Plus Enhanced could equal Fortran Plus in performance for common operations, Fortran Plus can give superior performance for less common operations. It would be interesting to extend the derivation presented here to derive Fortran Plus implementations, by explicitly sectioning vectors and matrices, but that is a topic for later work.

5.3.2 Fortran Plus Enhanced

Fortran Plus Enhanced (FPE) is based on Fortran77, but contains extensions that allow the processor grid to be used. These extensions are based on two data types: Vector and Matrix. A Vector is a one-dimensional array of Scalars, a Scalar being a floating point number, an integer or a boolean; a Matrix is a two-dimensional array of Scalars.

For manipulating Vectors, FPE provides the abstraction of a processor array that is as large as the largest Vector; each component of a Vector is manipulated by a different processor, but corresponding elements of different Vectors share a processor. A similar abstraction is supported for Matrices. The FPE compiler automatically partitions Vectors (which are stored in horizontal mode, see above) or Matrices (vertical mode) that are larger than the physical processor grid into sections that are the same size as the grid.

Fortran Plus Enhanced allows Vectors and Matrices to be treated in much the same way as scalars in Fortran 77:

- a variable can be of type Vector or Matrix;
- expressions can evaluate to a Vector or Matrix;
- a Vector/Matrix variable can be assigned a Vector/Matrix value in a single operation;
- the arguments to a function can be of type Vector or Matrix;
- a function can return a Vector or Matrix value.

In addition, FPE provides a set of functions that manipulate or construct vectors and matrices in a data-parallel manner:

Elementwise functions

Fortran Plus Enhanced extends the common arithmetic and boolean functions to Vectors and Matrices: a Scalar function is applied to each element of a Vector or Matrix, or to corresponding elements of a pair of Vectors or Matrices. In the latter case, the Vectors or Matrices must be the same size. In addition, for binary functions, FPE allows a Scalar value to be used in place of one of the Vector or Matrix arguments, with the meaning that the scalar is expanded to the same size as the remaining argument.

For example:

- the addition of two Matrices `A` and `B`: `A+B`
- adding a Scalar value to each element of a Vector `V`: `5+V`
- taking the square root of the elements of a Vector `V`: `sqrt(V)`
- performing the logical *and* of two Vectors `U` and `V`: `U.and.V`
- forming the boolean Matrix with elements true where the corresponding elements of `A` and `B` are equal: `A.eq.B`

There is no method in FPE for extending an arbitrary (programmer defined) Scalar function into a parallel operation for Vectors and Matrices (i.e. there is no equivalent of the `generate` or `map` functions discussed in chapter 2). If an arbitrary Scalar operation is to be applied to each element of a Vector or Matrix, a loop must be used to apply the operation one element at a time.

Reductions

FPE provides a set of common reductions: summation, conjunction, disjunction, maximization and minimization. These functions can be applied to an entire Vector or Matrix (giving a Scalar), or to the rows or columns of a Matrix (giving a Vector). For example:

- the conjunction (logical *and*) of the elements of the Vector `V`: `all(V)`
- the sum of the rows of the Matrix `A`: `sumc(A)`

- the maximum value in Vector `V`: `maxv(V)`.

As with elementwise functions, there is no method for performing arbitrary reductions on Vectors or Matrices (without using a loop).

Permutations

There are several functions available in FPE for permuting[3] the elements of a Vector or Matrix such as the Matrix transpose function, the Vector reversal function (which reverses the order of the elements), and the Vector and Matrix "shift" functions in which all the components in a Vector or Matrix are moved in the same direction (north, south, east or west for a Matrix, left or right for a Vector).

Expansions

As mentioned above, FPE allows a Scalar value to be used in place of a Vector or Matrix in binary elementwise functions. FPE also allows a Scalar value to be explicitly expanded into a Vector or Matrix. In addition, a Vector can be expanded either row-wise or column-wise into a Matrix.

Patterns

It is frequently useful to have a boolean Vector or Matrix with true values only in certain locations. For example, along the leading diagonal, or in a particular range of columns. FPE provides several functions for defining such patterns.

Indexing techniques

In addition to the functions discussed above, Fortran Plus Enhanced makes use of a variety of indexing notations to provide certain useful expressions. For example:

- `A( ,i)` where the row index is intentionally blank, is a Vector with elements the same as those of column `i` of `A`
- `A(i, )` is row `i` of `A`
- `U(V)` is a Scalar, where `V` is a boolean Vector with only one element true. The Scalar is the element of `U` in the same position as the true element of `V`.
- `A( ,M)` is a Vector, where `M` is a boolean Matrix with only one element true in each row.

Source matrix					Boolean index matrix					Result vector
1	2	3	4		T					1
5	6	7	8				T			7
9	10	11	12			T				10
13	14	15	16					T		16

Figure 5.5: Indexing a Matrix using a boolean Matrix

Element i of this Vector is taken from row i of Matrix `A`. The column position of this

[3]Strictly speaking, the functions are not permutations, as there is not a 1-1 mapping between the elements of the argument and the elements of the result.

element is the position of the true value in row i of `M`. (See figure 5.5.) A particular example of this type of indexing is extracting the diagonal of a Matrix, for which the indexing Matrix contains `true` values along only the leading diagonal.

- `A( ,I)` is a Vector, where `I` is an index Vector, i.e. it is an integer Vector in which all the integers are the in the range 1 to m where m is the size of a column of A.

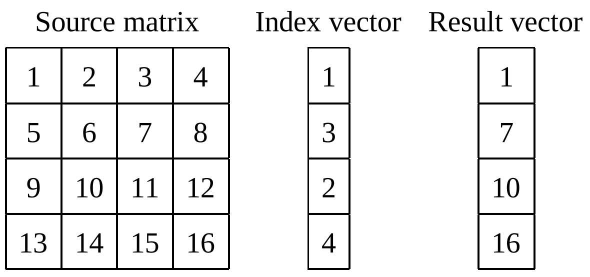


Figure 5.6: Indexing a Matrix using an index Vector

Component `i` of the result is element `I(i)` in row `i` of `A`. (See figure 5.6.)

The last two indexing techniques can be combined with a Scalar index for some truly arcane notation. For example, if `M` is a boolean matrix, then `A(2,M)` is that element of `A` with row position 2 and column position the same as the only true element of row 2 of `M`.

In addition to the functions listed above, FPE can modify all of an array or parts of an array in a single assignment. For example, if `A` and `B` are matrices of the same size, then

```
A = B*B
```

assigns to each element of `A` the square of the corresponding element of `B`; and

```
A(abs(A.gt.1.0)) = 1.0
```

assigns the value 1 to each element of `A` which has magnitude greater than 1, using *masked assignment*. The mask is the matrix expression `abs(A.gt.1.0)`: the assignment affects only those elements of `A` for which the corresponding element of the mask has value `true`.

5.3.3 Summary

In general, the primary concern when implementing an algorithm on the DAP is to make use of elementwise operations wherever possible; the elementwise operations make best use of the grid of processing elements. For example:

```
A = B+C
```

is vastly more efficient than

```
   DO 10 i=1,m
   DO 10 j=1,n
   A(i,j) = B(i,j)+C(i,j)
10 CONTINUE
```

because the former is, conceptually, executed in a single step by the processor grid, while the latter is executed in $m \times n$ steps, with only one element of `A` being assigned in each step.

Other operations such as expansions, permutations and indexing techniques are used to manipulate data in such a way as to allow elementwise operations to be performed; for example:

- Vectors are expanded into matrices so they can be employed in elementwise operations with other matrices. For example:

  ```
  A = B+matr(V,m)
  ```

  is much more efficient than

  ```
     DO 10 i=1,m
     A(i, ) = B(i, )+V
  10 CONTINUE
  ```

  where `matr(V,m)` constructs a matrix with m rows, each identical to the vector `V`.

- Parts of a matrix, such as the diagonal, can be extracted as a vector, so that they can be employed directly in an elementwise operation with another vector, or be expanded into a matrix and then employed in an elementwise operation with another matrix.

  For example, both of the following program fragments normalize a matrix by dividing each row by the diagonal element on that row, but

  ```
  A = A/matc(A(patunitdiag(n), ),n)
  ```

  is much more efficient than

  ```
     DO 10 i=1,n
     A(i, ) = A(i, )/A(i,i)
  10 CONTINUE
  ```

  .

- Masked assignment is employed for conditional operations so that elementwise operations can be applied to an entire vector or matrix, even though not all the elements are affected in the same manner. The alternative would be to iterate over the vector or matrix sequentially, determining one element at a time how that element should be affected by the conditional operation — vastly less efficient.

  For example

  ```
  A = 0
  A(patunitdiag(n), ) = 1
  ```

  is vastly more efficient than either

  ```
     DO 10 i=1,n
     DO 10 j=1,n
     IF (i=j) THEN
     A(i,j) = 1
     ELSE
     A(i,j) = 0
     ENDIF
  10 CONTINUE
  ```

  or

  ```
     DO 20 i=1,n
     DO 10 j=1,n
     A(i,j) = 0
  10 CONTINUE
     A(i,i) = 1
  20 CONTINUE
  ```

  where `patunitdiag(n)` is the FPE diagonal pattern function (leading diagonal elements .true.; all other elements false) .

- Patterns are employed to allow extractions as above or to facilitate masked assignment.

An obvious exception to this ‘elementwise’ rule is reductions, which are inherently cumulative rather than elementwise. However, even in reductions, elementwise operations are important in forming the vector or matrix to be reduced; the alternative would be to iterate sequentially over the vector or matrix combining one component at a time into the cumulative value.

For example

```
s = sum(U*V)
```

is much more efficient than

```
   s = 0
   DO 10 i=1,n
   s = s+U(i)*V(i)
10 CONTINUE
```

.

5.4 Basic Structure of Derivation

In this section I present an overview of the derivation to convert an SML specification into a Fortran Plus Enhanced implementation for the DAP; following sections discuss each of the stages of the derivation in greater detail.

The SML-to-DAP derivation is based upon a Lisp-to-Fortran77 derivation developed by Boyle and Harmer [22]:

$$\boxed{\text{Lisp} \longrightarrow \text{Fortran77} \quad \equiv \quad \text{Lisp} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \longrightarrow \text{Fortran77.}}$$

The important aspects for this chapter of this derivation are:

- The specification is in Lisp, rather than SML.
- The specification is initially transformed into the λ-calculus (extended to include named function definitions and type information). The λ-calculus form is a functional form with a syntax that, for reasons discussed later, is simpler to manipulate using transformations.
- Function definitions are unfolded to reduce the overheads caused by function applications. The Unfolded form is essentially an optimization of the λ-Calculus form: unfolding does not alter the basic nature of the program. The importance of this is that the final stage of the derivation, converting to Fortran77, is the same whether or not unfolding is performed.
- The implementation language is Fortran77, not FPE. The basic aspects of the conversion from λ-calculus to F77 were discussed in the preceding chapter.

Rather than construct an entirely new derivation, I chose to extend the above Lisp-to-Fortran77 derivation, so I could reuse the sub-derivations that perform unfolding and conversion to Fortran. Reuse of the latter is not as surprising as it may at first appear, since FPE is a superset of F77.

The first problem to be addressed in extending the derivation is that the specification language, SML, is not catered for by the derivation; two obvious solutions are to convert an SML specification into a Lisp specification, or to convert an SML specification into the λ-calculus form. There seems little to be gained by converting into Lisp, and since Lisp is type-less, information would be lost in the transition; so, the first stage of the SML-to-DAP derivation converts SML into the λ-calculus form. Once in the λ-calculus form, the specification can be optimized in the same manner as a Lisp specification by unfolding definitions.

$$\boxed{\text{SML} \longrightarrow \text{F77} \quad \equiv \quad \text{SML} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \longrightarrow \text{F77.}}$$

Thus, an SML-to-F77 derivation can be constructed. Such a derivation would, strictly speaking, serve as an SML-to-FPE derivation, since FPE is a superset of F77. (i.e. the implementation produced by this derivation would be acceptable to the FPE compiler.)

However, such an implementation would be inefficient, for the following reason. All array operations in a specification are based primarily upon the two functions `generate` and `reduce`, but the only way to implement an *arbitrary* generation or reduction in FPE is to use a sequential loop. For example, the F77 derivation would implement

```
generate([n,n],
  λi,j·element(A,[i,j])+element(B,[i,j]))
```

as

```
   DO 10 i=1,n
   DO 10 j=1,n
   C(i,j) = A(i,j)+B(i,j)
10 CONTINUE
```

which would execute in n^2 sequential steps. In contrast, an efficient implementation that makes effective use of the processor grid, is

```
C = A+B
```

The Fortran77 sub-derivation is designed to perform the conversion from a simple functional form (λ-calculus) into imperative form; it is not designed to address the issue of converting array operations expressed using `generate` and `reduce` into operations that make effective use of the DAP hardware. This latter issue is addressed by introducing a new sub-derivation that converts, where possible, generations and reductions into combinations of FPE's basic data-parallel operations. This new sub-derivation introduces the *Array Form* into the derivation:

SML ⟶ DAP ≡ SML ⟶ λ-Calculus ⟶ Unfolding ⟶ Array Form ⟶ FPE.

The Array Form contains applications of the basic data-parallel operations supported by FPE. However, these operations are expressed as pure functions, rather than in the mix of functions, operators, special index notation and statements employed by FPE. For example, forming a matrix in which each element is the greater of the corresponding elements of two matrices `B` and `C` would be expressed in FPE and in the Array Form as

DAP FPE
```
M = B.gt.C
A(M) = B
A(.not.M) = C
```

and

Array Form
```
join(greaterp(B,C),B,C)
```

where the function `join` denotes a data-parallel conditional expression: it evaluates to an array of the same shape as its arguments, with each element taken from `B` or `C` according to whether the corresponding element of the first argument is true or false. Using a pure, functional form has all the advantages discussed earlier for functional forms: primarily simpler semantics and syntax.

Finally, the Array Form is converted into imperative form, in the DAP specific syntax of Fortran Plus Enhanced. The sub-derivation to convert to FPE is similar to the sub-derivation to convert λ-calculus into Fortran77 because FPE is a superset of F77. The major differences in the two sub-derivations is that the FPE sub-derivation contains extra transformations for the idiosyncrasies of FPE vector and matrix syntax. Indeed, the FPE sub-derivation could be used to convert a specification in standard λ-calculus form (with no explicitly data-parallel functions) into standard Fortran77.

5.4.1 Two Refinements to the Derivation

The basic SML-to-DAP derivation is:

SML ⟶ DAP ≡ SML ⟶ λ-Calculus ⟶ Unfolding ⟶ Array Form ⟶ FPE.

However, the derivation used in practice contains two further important sub-derivations for performing *Static Evaluation* and *Common Sub-expression Elimination*:

SML ⟶ DAP
≡ SML ⟶ λ-Calculus ⟶ Unfolding ⟶ <u>Static Evaluation</u>
⟶ Array Form ⟶ <u>Common Sub-expression Elimination</u> ⟶ FPE.

Static Evaluation

Function unfolding removes much of the structure of a definition that is introduced by a programmer to facilitate human understanding, but which can decrease the efficiency of an implementation.

However, unfolding does not remove *intermediate data structures* which are caused by constructing the arguments to functions and return values from functions.

For example, simple unfolding of matrix-vector multiplication results in the following expression

```
fun times(A:real array,B:real array):real array
  = map(A,B,real.times);
fun sum(A:real array):real
  = fold(A,real.plus,0.0);
fun innerproduct(A:real array,B:real array):real
  = sum(times(A,B));
fun mvmult(A:real matrix,V:real vector):real vector
  = let
    val m:int = dim(A,0)
  in
    generate(m,fn(i:int)=>innerproduct(row(A,i),V))
    end
```

$\longrightarrow$

```
array.generate ([m],λi·
  λA′·
    [array.reduce(shape(A),
      λi′·element (A′,[i′]),
      real.plus,0.0)
    ]
  (λA″·
    [array.generate(shape(A″),
      λi″·real.times(
        element(A″,[i″]),
        element (V,[i″]))
      )
    ]
    (array.generate (shape.dim(A,0),
      λi‴·element (A,[i,i‴]))
    )
  )
)
```

An operational interpretation of the unfolded form is:

- Construct a vector with m elements, in which each element is computed as follows:
  - Form a binding for A′: the bound value is computed as follows:
    - Form a binding for A″ in which the bound value is a vector constructed by copying all the elements from a row of A.
    - Construct a vector of the same shape as A″ in which each element is the product of an element of A″ and an element of the vector V.
  - Perform a reduction over the shape the same as the shape of A, in which each value contributing to the reduction is an element of the vector A′.

The important aspects of the operational interpretation are that, for *every* element of the product vector, a vector is constructed by copying a row of the matrix, and a second vector is constructed by multiplying the elements of the first vector by those of V. These two vectors are not necessary for the computation: they are present solely because of the modular definition of matrix-vector multiplication.

It should be clear that the first vector can be eliminated by accessing the elements of the matrix *in situ*, rather than copying them into a vector; and the second vector can be eliminated by merging the forming of the elementwise product with the summation. Eliminating the data structures results in the expression:

```
array.reduce(shape(A),
  λi′·real.times(
    element(A,[i,i′]).
    element(V,[i′]))
  )
```

Such simplifications are performed by the Static Evaluation sub-derivation, so called because the simplifications are based on evaluating expressions, using the properties of the λ-calculus and various data types.

The drastically reduced form of MVM created by Static Evaluation is certainly simpler in structure than the initial, unfolded version. But the simplicity is more fundamental than structural simplicity, though that may not be apparent from the above example. In the library functions provided to allow an SML specification to be executed, array indices are implemented using list operations; such an implementation was convenient for SML. However, I have no intention of implementing a list data type for the DAP: array indexing in the FPE implementation will be performed using standard Fortran77 indexing, or the extensions allowed by FPE. Performing static evaluation has the benefit of removing almost all list operations performed on indices from a specification.

Aside from the effect list-based array indexing would have on the execution performance of an implementation, judging whether the reduced form of MVM is more efficient than the original form is difficult, as what is efficient depends upon the nature of the implementation system. For example, if the matrix `A` is distributed over several processors, it may indeed be more efficient for the processor that is computing a particular component of the product vector to copy a row of the matrix into a vector which is stored locally (i.e. to prefetch all the required elements of `A` in a single communication operation, rather than fetch each element individually).

However, matters that are peculiar to the implementation system are best left until late in the derivation: if for a particular implementation system it is better to copy a row of the matrix, then the copy operation should be introduced in a stage of the derivation devoted to that implementation system.

Common Sub-expression Elimination

Consider an expression such as `f(e,e)`, where `e` is some computationally expensive expression. A straightforward implementation of this expression (such as created by the final stages of derivations) would result in `e` being evaluated twice. A more efficient implementation would evaluate `e` once, assign the value to a variable, and refer to the variable twice. In functional terms, this more efficient implementation corresponds to the expression λ`v`·`[f(v,v)](e)`.

The λ-abstraction of multiple occurrences of an expression is known as *common sub-expression elimination*, or CSE. Many compilers for imperative languages perform CSE, but imperative languages present many difficulties (mainly arising from side effects) that complicate CSE; it is to be expected that CSE can be performed much more easily in a pure, referentially transparent context.

5.4.2 Summary

The SML-to-DAP derivation has the form

$$\boxed{\begin{array}{l}\text{SML} \longrightarrow \text{DAP}\\ \equiv \text{SML} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \longrightarrow \text{Static Evaluation}\\ \quad\longrightarrow \text{Array Form} \longrightarrow \text{Common Sub-expression Elimination} \longrightarrow \text{FPE.}\end{array}}$$

The stages can be summarized as follows:

- **SML** The source form — the specification — is constructed in an abstract, modular fashion using standard notations to facilitate human understanding.

- **λ-Calculus** The specification is cast into a standard, functional form whose simple syntax facilitates transformation.

- **Unfolding** The structure of the specification is simplified to improve efficiency.

- **Static Evaluation** Further structure is removed, to produce a succinct form.

- **Array Form** Array expressions are recast into a form that make makes effective use of the DAP hardware.

- **Common Sub-expression Elimination** The specification is restructured to eliminate repeated computations.

- **FPE** The program is converted into imperative form, expressed in DAP specific syntax.

Of the stages of the derivation:

- The conversion from SML to λ-calculus is mostly my own work, though several of the transformations were provided by Harmer.
- The Unfolding sub-derivation is the same as a sub-derivation developed by Boyle and Harmer for Lisp specifications, but which I extended to accommodate type information.
- The Static Evaluation sub-derivation is my own work.
- The Array Form sub-derivation was developed by me, though the idea of separating shape information was suggested by Harmer.
- The Common Sub-expression Elimination derivation is my own work.
- The sub-derivation to convert Array Form into FPE was based on the λ-calculus to F77 sub-derivation developed by Harmer and Boyle. I extended the derivation to accommodate type information and the idiosyncrasies of FPE and to extend reuse of storage.

Probably the most interesting of the sub-derivations is the conversion to Array form: that sub-derivation produces a fundamental change in the nature of a specification, but does so in an elegant, straight-forward manner. It is a prime example of how I believe derivations *should* work.

5.5 From SML to λ-Calculus

In this and the following sections, I discuss each stage of the derivation in detail. This section discusses the conversion from SML to λ-calculus.

SML is convenient as a specification language as it provides mechanisms for a programmer to structure a specification (primarily function and structure definitions) and allows standard notation (e.g. + indicating matrix addition) or fairly intuitive notation (`fn(x)=>e` indicating a function), facilitating human understanding of a specification.

However, human understanding is not of primary importance when performing a derivation automatically: ease of manipulation is. In this respect, SML (in common with many languages) is somewhat flawed; for example:

- When manipulating expressions in operator form, attention to details of precedence amongst the various operators is required.
- The same basic concept appears in multiple guises: for example, local expressions can be viewed as being essentially the same as function prototypes which have been applied to arguments, and function definitions are essentially named function prototypes. For example, the local expression

```
let
  val x = X
in  B
end
```

is equivalent to

```
(fn(x)=>B)(X)
```

.

- Definitions can be introduced in multiple locations: at the top level of the specification, inside structures and inside local expressions. Transformations that manipulate definitions must first undertake the task of locating the definitions.
- The use of `open` statements to make the definitions contained in structures available as top level definitions means that the same function can be invoked under two different names (function `f` in structure `S` can be invoked as both `f` and `S.f`).
- The same name can be introduced by more than one definition: a use of such a name refers to the most recent, preceding definition. Manipulations involving such a name must determine to which definition it refers.

These comments are not criticisms of SML: SML is useful in this work precisely because of its convenience for human programmers, not for its suitability (or lack thereof) for transformation.

To simplify transformation, an SML specification is converted into a simple functional form: essentially the λ-calculus, extended to allow definitions to be named and to include type information. The λ-calculus form is a list of function definitions in which expressions are denoted in function prefix form (e.g. the expression `x+y` becomes `op.plus(x,y)`). The λ-calculus form is much more readily manipulated than SML:

- Expressions are in function prefix form, with arguments to functions explicitly delimited, so no precedence rules are required.
- Function prototypes, function definitions and local expressions are all expressed as λ-expressions, and so can all be manipulated uniformly. For example:

```
fn(x)=>F       ⟶   λx·F

fun f(x) = F   ⟶   fun f = λx·F

let
 val x = X
in E           ⟶   λx·E (X)
end
```

- Definitions all occur at the top level where they are easily located.
- Definitions are unordered: where necessary, definitions have been renamed to ensure their names are unique, so there is no need for complex rules on the scope of names.
- All references to a definition use the full name of the definition (in the form `Structure.name`).
- Where necessary, definitions have been renamed so that no definition introduces a name that has already been introduced by another definition.

The conversion from SML to λ-calculus is mostly syntactic, so details of the method are postponed until appendix A. Figure 5.7 shows the λ-calculus form of MVM.

5.5.1 Summary

After conversion from SML to λ-calculus, a specification consists of a (conceptually) unordered set of function and type definitions in which expressions can take only the forms: constant, identifier,

```
fun realarray.times:(real array*real array->real array)
  = λA,B·realarray.map2(A,B,real.times);
fun realarray.sum:(real array->real)
  = λA·realarray.fold(A,real.plus,0.0);
fun realarray.innerproduct:(real array*real array->real array)
  = λA,B·realarray.sum(realarray.times(A,B));
fun realarray.mvmult:(real matrix*real vector->real vector)
  = λA,V·
    λm:int·
      realarray.generate(m,
        λi:int·realarray.innerproduct(matrix.row(A,i),V))
    (shape.dim(A,0));
fun realarray.multiply:(real array*real array->real array)
  = λA,B·
    λra:int·λrb:int·
      if (bool.and(op.eq(ra,1),op.eq(rb,2)))
        then realarray.vmmult(A,B)
      else if (bool.and(op.eq(ra,2),op.eq(rb,1)))
        then realarray.mvmult(A,B)
      else
        realarray.mmmult(A,B)
    (shape.rank(B))
    (shape.rank(A));
fun op.eq
  = overload(integer.eq,real.eq,bool.eq);
fun op.times
  = overload(integer.times,real.times,realarray.multiply);
fun init:(int->real vector)
  = λn:int·
    λA:real matrix·λV:real vector·
      op.times(A,V)
    (realarray.read_matrix([n,n]))
    (realarray.read_vector([n]));
```

Figure 5.7: λ-calculus form of MVM

prefix function application, λ-bindings and conditional expressions.

A specification is basically unchanged in nature (for example, there is a one-to-one mapping from function definitions in the SML form of the specification to function definitions in the λ-calculus form; there is also a one-to-one mapping between expressions) but is is simpler in form and syntax, so facilitating later stages of the derivation.

5.6 Unfolding

In functional specification, programmers are encouraged to employ functional abstraction of data (e.g. using functions to access components of data rather than accessing the components directly) and functional decomposition of operations to facilitate human understanding of a specification. One consequence of this is that a specification often contains many functions that are simply compositions of other functions, that aid the programmer in developing the specification by improving its modularity, but that perform no significant computation in themselves.

If a derivation preserves all functions as functions/subroutines in an implementation, that implementation can suffer high overheads associated with the function invocation mechanism (for, say, placing arguments on a stack and retrieving results from the stack, or saving the values of processor registers). To reduce such overheads, the number of function invocations can be reduced by unfolding definitions: each use of a definition is replaced with the body of the definition, with any formal and actual arguments appropriately matched.

Aside from reducing overheads due to function calls, unfolding is useful for other reasons:

Unfolding permits an axiomatic approach to program manipulation

When constructing Abstract Data Types, it is often possible to encapsulate the underlying implementation of the data type into a small set of *basic* functions; the rest of the functions — the *derived* functions — for the data type can be constructed in terms of those basic functions, in ignorance of the underlying representation. For example, the basic functions of the array ADT are `generate`, `element`, `shape` and `reduce`; functions such as `map`, `fold`, `row`, `sum`, etc. are defined, ultimately, in terms of those basic functions.

When such a manner of construction is used for a data type, the semantics of the data type can be defined by specifying just the semantics of the basic functions: the semantics of all the remaining functions follow from the semantics of the basic functions, which thus act as axioms for the data type.

When function unfolding is applied to a data type, only the derived functions may be unfolded, leaving only applications of the basic functions. Then, when considering transformations for a data type, it is necessary to consider only the basic functions, since the applications of all the derived functions have been removed. For example, when converting array expressions into data-parallel form, it is necessary to consider only `generate` and `reduce`: `map` need not be considered, because `map` no longer exists.

Thus, unfolding definitions permits program manipulation to be performed based upon the axioms of data types.

Unfolding increases the granularity of computation

The function definitions remaining after unfolding generally represent major parts of the computation required by an algorithm. Each of these functions is likely to be a suitable candidate for a process in a process-parallel execution model.

Even in a sequential execution model, unfolding may help improve the effectiveness of any processor instruction or data caches by increasing the number of operations performed before the cache must be rebuilt (due to transfer of execution to another function).(i)

Unfolding places definitions in context

One of the advantages (as regards human construction and understanding of specifications) of functional decomposition is the high degree of isolation enforced between a function definition and applications of that function: a definition can be considered independently of the context in which it is to be applied, and a function can be used without regard to the function's definition.

However, often the context in which a function is applied can be used to perform optimizations or simplifications. For example, the `multiply` function was designed to perform different computations (matrix-vector, matrix-matrix or vector-matrix multiplication) based upon some tests on the dimensionalities of its arguments. Unfolding the definition of `multiply` at the point of application may allow the outcome of the tests to be determined by analysis, so eliminating the need to perform them during program execution; the definition can then be simplified to just that portion required for the context (matrix-vector multiplication for the MVM example).

In addition, the presence of a program-defined function in a generating function can obstruct the conversion of the generation into data-parallel form: How would the application of the program-defined function be performed in a data-parallel manner? No mechanism exists in Fortran Plus Enhanced for, say, applying a program-defined function to all the elements of a matrix simultaneously.

For example, many of the array functions used in this thesis are polymorphic, permitting arrays to have any type of element. Unfolding places each such function in context, where the type of the array elements may be known, thus obviating the need to support polymorphism in an implementation, and also perhaps permitting optimizations based on the element type.

Thus, unfolding effectively allows a definition to be optimized for or specialized to each of its uses independently, and can dramatically increase the effectiveness of Static Evaluation.

The sub-derivation for unfolding was based on a sub-derivation developed by Boyle and Harmer for Lisp specification, which I extended to accommodate type information.

5.6.1 The MVM Example

```
fun init
  = λn·
    λA·λB·
      λra·λrb·
        if (bool.and(integer.eq(ra,1),integer.eq(rb,2)))
          then ...
        else if (bool.and(integer.eq(ra,2),integer.eq(rb,2)))
          then ...
        else
          λm·
            array.generate([m].λ[i]·
              λA′·
                array.reduce(shape(A),
                  λ[i′]·element(A′,[i′]),
                  real.plus,0.0)
              (λA′′·
                array.generate(shape(A′′),
                  λ[i′′]·real.times(
                    element(A′′,[i′′]),
                    element(V,[i′′]))
                  )
                (array.generate(shape.dim(A,0),
                  λ[i′′′]·element(A,[i,i′′′]))
                )
              )
            )
            (shape.dim(A,0))
        (shape.rank(B))
        (shape.rank(A));
      (realarray.read_matrix([n,n]))
      (realarray.read_vector([n]));
```

Figure 5.8: Unfolded form of MVM

Unfolding definitions causes MVM to collapse into a single function, as shown in figure 5.8, where:

- The ellipses represent calculations that are irrelevant for this example; they will be eliminated by Static Evaluation.
- A' is bound to the element-wise product of a row of A and V.
- A'' is bound to a row of A.

5.6.2 Summary

Unfolding recasts a specification into a form with increased locality: each definition is placed into the contexts of its applications, allowing each application to be optimized independently. Any function definitions remaining after unfolding are likely to represent major computations.

5.7 Static Evaluation

In this section, I discuss the simplification of a specification by Static Evaluation.

The modular structure of a functional specification requires the construction of many data structures that are arguments to functions or return values from functions. Simple unfolding of definitions reduces the number of function applications, but it does not remove the data structures; removal of the data structures is performed by Static Evaluation, which uses the properties of the λ-calculus and of various data types to evaluate expressions.

For example, the unfolded form of MVM is shown in figure 5.8. The unfolded form contains the following expression:

```
λA″·
  array.generate(shape(A″),
    λ[i″]·real.times(
      element(A″,[i″]),
      element(V,[i″]))
    )
  (array.generate(shape.dim(A,0),
    λi‴·element(A,[i,i‴]))
  )
```

Removal of A″ is performed as follows:

- β-reduce the outer λ-expression.

```
array.generate(shape(A″),
  λ[i″]·real.times(
    element(
      array.generate(shape.dim(A,0),
        λ[i‴]·element(A,[i,i‴])).
      [i″]),
    element(V,[i″]))
  )
```

- The expression now contains an application of `element` to `generate`: that is, the expression constructs an array and then retrieves a single element. But the semantics of `generate` specify that the value of an element of an array constructed using `generate` is the value of the application of the generating function to the index for that element:

$$element(generate(S, \lambda i \cdot g), i') \equiv \lambda i \cdot g(i') \ .$$

  Thus, the application of `element` to `generate` can be evaluated.

```
array.generate(shape(A″),
  λi″·real.times(
    λ[i‴]·element(A,[i,i‴]) (i″),
    element(V,[i″]))
  )
```

- The λ-expression so formed can be reduced.

```
array.generate(shape(A′′),
  λ[i′′]·real.times(
    element(A,[i,i′′]),
    element(V,[i′′]))
  )
```

Thus, a sequence of simple evaluations, based on the properties of the λ-calculus and of the array functions, removes the intermediate data structures.

5.7.1 Method

The exhaustive application of transformations performed by the transformation system allows Static Evaluation to be realized very simply, as a sequence of transformations, automatically applied, with each transformation performing one simple, local evaluation based upon the properties of the λ-calculus or of various data types (primarily arrays). There is no need to encode some mechanism in the transformations to locate expressions that can be simplified; no need to consider the interaction of evaluations, how one evaluation might lead to another. It is sufficient simply to list all the evaluations that produce simplifications.

Informally, each transformation is readily seen to recast an expression in a simpler form, so the application process is believed to terminate; it should not be difficult to prove termination formally by providing a formal measure for expressions and showing that each transformation reduces that measure; further discussion is beyond the scope of this thesis.

In the rest of this section, I list some of the transformations that perform the individual evaluations required by Static Evaluation. The rest of the transformations are similar in nature to the listed transformations.

Arrays

The defining properties of `generate` are:

$$element(generate(S, \lambda i{\cdot}g), i')\equiv\lambda i{\cdot}g(i')$$

$$shape(generate(S, \lambda i{\cdot}g), i')\equiv S \ \ .$$

These properties can be directly utilized as transformations:

```
.sd.
  element(generate(S,λi·g),[j])
  ==>
  λi·g (j)
.sc.
.sd.
  shape(generate(S,λi·g))
  ==>
  S
.sc.
```

It is worth noting that these transformations, as with all the transformations in this section, are made possible by the simple semantics of expressions: it is correct to replace one expression with another simpler expression when they have the same value because the only meaning of an expression is its value.

Reals

Most of the common operations on real numbers are taken to be primitive operations (i.e.

having no definition in the specification). Each of these operations has algebraic identities, often associated with the numbers one and zero. For example:

$$x + 0.0 \equiv x \quad 0.0 + x \equiv x$$

$$x * 0.0 \equiv 0.0 \quad 0.0 * x \equiv 0.0$$

$$x * 1.0 \equiv x \quad 1.0 * x \equiv x$$

$$x / x \equiv 1.0 \ .$$

Again, transformations follow directly from these identities.

```
.sd. plus(x,0)      ==> x   .sc.  .sd.  plus(0,x)    ==> x .sc.
.sd. times(x,0)     ==> 0   .sc.  .sd.  times(0,x)   ==> 0 .sc.
.sd. times(x,1.0)   ==> x   .sc.  .sd.  times(1.0,x) ==> x .sc.
.sd. quotient(x,x) ==> 1.0 .sc.
```

Many (though not all) of the transformations for reals have counterparts for integers. It should be clear that transformations can also be constructed in a similar manner for booleans.

Conditional Expressions

Consider a conditional expression

```
if (p) then T else F
```

where `T` is the 'true limb' and `F` the 'false limb'. Only when `p` is true, is `T` evaluated; consequently, when `T` is being evaluated, the expression `p` is known to have value `true`. In other words, any occurrence of the expression `p` in `T` can be replaced with `true`. Similarly, the expression `p` can be replaced with the value `false` in `F`.

The transformation for this *contextual evaluation* of conditional expressions is[4]:

```
.sd.
  if (p) then T else F
  ==>
  if (p) then T [true/p] else F [false/p]
.sc.
```

A programmer is unlikely to construct expressions which would be affected by contextual evaluation, but experience has shown that function unfolding often creates expressions of the form

```
if (p)
then if (p) then A else B
else if (p) then C else D
```

presumably because functional decomposition often requires a conditional expression to be repeated in several functions. Such expressions are simplified by contextual evaluation

```
==> if (p)
then if (true) then A else B
else if (false) then C else D
```

and then simplified with the obvious transformations

[4]This transformation must be qualified if it is to be used in practice, otherwise its application would never terminate. One way to qualify it would be to use indefinites in the pattern to check for the presence of the predicate in either limb. Alternatively, a provisional transformation can be used: the TAMPR system allows a transformation to be provisionally applied, and for the provisional application to be considered a true application only if the transformation causes a change.

```
.sd. if (true) then T else F  ==> T .sc.
.sd. if (false) then T else F ==> F .sc.
```

to give

```
==> if (p) then A else D
```

The transformations listed above, and others similar in nature, when combined with a β-reduction transformation in a transformation sequence and exhaustively applied by the transformation system, perform Static Evaluation.

5.7.2 The MVM Example

The Statically Evaluated form is illustrated by MVM in figure 5.9. Comparison with figure 5.8 demonstrates the effect Static Evaluation can have in drastically simplifying programs.

```
fun init
  = λn·
    λA·λV·
      array.generate([n],λ[i]·
        array.reduce([n],
          λ[j]·real.times(
            element(A,[i,j]),
            element(V,[j])),
          real.plus,0.0))
      (realarray.read_matrix([n,n]))
      (realarray.read_vector([n]));
```

Figure 5.9: Statically Evaluated form of MVM

5.7.3 Summary

The specification now is still expressed in λ-calculus form, but it has been greatly simplified (and possibly optimized), first by Unfolding, then by Static Evaluation. Almost all the ‘structure’ that was present in the initial specification (to facilitate human understanding) has been stripped away, leaving a succinct form amenable to manipulation. From this point on, the derivation reintroduces structure, but structure that benefits the final implementation, either by simply making it compatible with the implementation language, or by improving the operational efficiency.

5.8 Array Form

In this section, I discuss the conversion of array operations expressed as generations and reductions into data-parallel operations suited to the DAP.

In earlier chapters I demonstrated various methods for constructing arrays, using general functions such as `generate` and `map`, or using more specific functions such as `transpose`. Ultimately though, all such functions are defined in terms of `generate` and `reduce`. The Unfolding stage of the derivation ensures that functions such as `map` have been eliminated, so I need only consider applications of the basic functions.

In a specification, `generate` is not defined as a function: rather, `generate` is defined by the identities

$$element(generate(S, \lambda i{\cdot}g), i') \equiv \lambda i{\cdot}g(i')$$

$$shape(generate(S, \lambda i{\cdot}g)) \equiv S \ \ .$$

One important aspect of this definition is that no order is specified for the creation of the elements: thus, `generate` can be readily interpreted for data-parallel execution, and so is potentially well-suited to an array processor.

However, the Fortran Plus Enhanced (FPE) language supports only a fixed set of data parallel operations (such as the common arithmetic and boolean functions)[5]: the only way to implement an *arbitrary* generation is as a sequential loop. Thus, if an efficient implementation is required, generations must be converted into combinations of those data parallel operations that are supported.

For example:

```
generate(S,λi·
  plus(element(U,i),
    element(V,i))
  )
```

becomes

```
U+V
```

and

```
generate([n,n],
  λ[i,j]·or(
    eq(element(A,[i,j]),
      element(B,[j,i])),
    greaterp(element(C,[i,j]),
      element(D,[i,j]))
    )
  )
```

becomes

```
(A.eq.tran(B)).or.(C.gt.D)
```

.

Some of the data parallel operations supported by the DAP are designated by pure functions in FPE; other operations are designated as operators or using special indexing notation in statements. Rather than attempt to convert from a simple functional form to a mixture of functions, operators and statements, I first convert to an Array Form which is a pure, functional abstraction of the supported data-parallel operations, and then convert the Array Form into DAP specific form.

The initial conversion to Array Form thus separates the problem of converting array expressions denoted using `generate` into the supported data-parallel operations, from the problem of converting a functional form into an imperative form. The postponement of conversion into imperative form allows any further manipulations on the Array Form to take advantage of the simpler semantics of pure, functional forms. It also postpones the introduction of the sometimes esoteric Fortran Plus Enhanced syntax which, though presenting no particular conceptual problems, would unnecessarily complicate transformations.

The conversion to the supported data-parallel operations is relatively straightforward when all the operations are elementwise (i.e. no inter-processor communication is required). Conversion is more difficult when index i requires elements stored at some other location (i.e. inter-processor communication is required).

The purpose of this section is to demonstrate how algebraic laws can be used to perform the conversion from `generate` to the Array Form functions. I shall first illustrate the conversion to Array Form, and then discuss in detail the Array Form functions and the conversion to Array Form.

[5]This restriction is to be expected given the nature of the hardware.

5.8.1 Conversion to Array Form: Illustration

The most important functions in the Array Form are the elementwise versions of the basic arithmetic and logical functions. The elementwise versions are designated using the function `array.map`: for example `array.map(A,B,real.plus)` is the addition of two real arrays.

The primary principle in converting to Array Form is to convert to applications of the elementwise functions where possible. For example, given the expression

```
generate(S,λi·real.plus(x,y))
```

the *application of* `generate` *is propagated through the outer function* `real.plus` to introduce the Array Form elementwise version:

```
array.map(generate(S,λi·x),generate(S,λi·y),real.plus)
```

Each of the arguments to the elementwise version are now themselves generations, on which the processes of conversion is repeated. At some point, propagation is not possible: i.e. the outer function does not have an elementwise version; then, the generation is checked to see if it corresponds to certain forms that correspond to other, non-elementwise Array Form functions such as the pattern functions or expansion functions. Conversion to these functions may result in further generations being introduced, which in turn are processed. (For example, when a row- or column-wise expansion of a vector is introduced, the vector that is to be expanded must be defined. It is defined as a generation.)

Other principles can be invoked in the conversion to optimize expressions: for example, a vector generation of vector reductions can be optimized by converting it into a row- or column-wise reduction of a matrix, so that all the reductions are performed simultaneously.

To further illustrate conversion, I will trace the conversion of MVM. The statically evaluated form is:

```
array.generate([n],
  λ[i]·
    array.reduce([n],
      λ[j]·real.times(
        element(A,[i,j]),
        element(V,[j])),
      real.plus,0.0)
  )
```

This expression is a generation over i in which the generating function is a reduction over j. The generating function can be evaluated for all the i simultaneously by converting the `generate`/`reduce` composition into a partial reduction: the summation of a matrix along the rows.

```
==> fold.rows(
  array.generate([n,n],
    λ[i,j]·real.times(
      element(A,[i,j]),
      element(V,[j]))
    )
  real.plus,0.0)
```

where the `fold.rows` function indicates a reduction along the rows of its matrix argument.

The outer function in the generating function is now `real.times`; this function has an elementwise version, so the generation is propagated inward, the application of `generate` becoming an application of the data-parallel mapping function, `array.map`:

```
==> fold.rows(
  array.map(
    array.generate([n,n],λ[i,j]·element(A,[i,j])),
    array.generate([n,n],λ[i,j]·element(V,[j])),
    real.times),
  real.plus,0.0)
```

The first of the remaining generations is an identity generation: it is equivalent to A:

```
array.generate([n,n])(λ[i,j]·element(A,[i,j])) ==> A
```

In the second of the remaining generations, the body of the generating function is independent of one of the generating indices, `i`, and so the generation is an *expansion*: i.e. a matrix is formed by duplicating a vector along the rows or columns. The vector to be duplicated is formed simply by deleting the index `i`:

```
array.generate([n,n],λ[i,j]·element(V,[j]))
==>
expand.rows(n,generate([n],λ[j]·element(V,[j])))
```

The remaining generation is another identity generation, equivalent to V:

```
generate([n],λ[j]·element(V,[j])) ==> V
```

Thus, the entire MVM expression becomes:

```
==> fold.rows(
  array.map(A,expand(n,V),real.times),
  real.plus,0.0)
```

The evaluation of this expression is illustrated in figure 5.10.

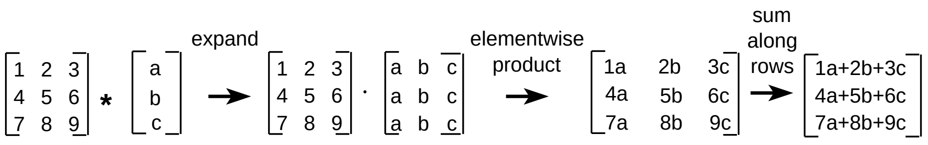


Figure 5.10: Data-parallel evaluation of MVM

- The vector V is expanded into an array the same size as A.
- The two arrays are multiplied element-by-element.
- The rows of the resulting array are summed.

5.8.2 Definition of the Array Form and Details of Conversion

The conversion to Array Form is based upon an algebra for converting array expressions involving single-element expressions into whole-array expressions. The preceding section illustrated that the most important aspect of the algebra is the equivalence between a generation of a scalar function which has an elementwise version and an application of that elementwise version. Other important aspects are equivalences involving generations of conditional expressions and λ-bindings. Finally, there are numerous special cases for functions such as expansions and the pattern functions.

In this section, I provide definitions for the major Array Form functions and various examples of less important functions. I also discuss the equivalences between the Array Form functions and expressions that may occur in the Statically Evaluated form.

Main Categories of Expressions

The most important categories of expression are:

Elementwise functions

Most of the basic arithmetic and boolean functions have data-parallel versions in FPE. These functions can all be expressed using the `map` function, but to emphasize the data parallel aspect, I add the prefix `array` for the Array Form. For example

- Elementwise addition of two real arrays is expressed as `array.map2(A,B,real.plus)` and of two integer arrays `array.map2(U,V,integer.plus)`.
- Elementwise *not* of a boolean array: `array.map(A,bool.not)`.

The mapping functions are defined as

```
array.map(A:α array,f:α->β):β array
  ≡ generate(shape(A),λi·f(element(A,i)))
array.map2(A:α array,B:β array,f:α*β->γ):γ array
  ≡ generate(shape(A),λi·f(element(A,i),element(B,i)))
```

where α, β and γ are restricted to integer, real and boolean, where `f` is one of the functions which can be applied in parallel, and where `A` and `B` have the same shape.

These definitions can be used to introduce elementwise operations into an expression that uses `generate`:

```
.sd.
  generate(S,λi·f(a))
  ==>
  array.map(generate(S,λi·a),f)
   where f is one of the functions which can be applied in parallel
.sc.
```

That is, a generation of a function `f` which can be applied in parallel is converted into an explicitly parallel mapping. The array over which the mapping is performed is computed by applying the generation to the argument of `f`.

A similar transformation applies for binary mappings.

```
.sd.
  generate(S,λi·f(a,b))
  ==>
  array.map2(generate(S,λi·a),generate(S,λi·b),f)
.sc.
```

Elementwise reductions

An elementwise reduction is a reduction in which the values being reduced are all elements of an array: e.g.

```
reduce(S,λi·element(A,i),reals.plus,0.0)
```

is an elementwise reduction but

```
reduce(S,
  λi·real.times(element(U,i),element(V,i))
  reals.plus,0.0)
```

is *not* elementwise since the generating function is not simply an application of `element`. All the reductions provided by FPE and by the Array Form are elementwise.

The data parallel versions of the elementwise reductions are expressed using the function `array.fold`. Partial reductions are indicated by adding a `rows` or `cols` suffix. For example:

- Summation: `array.fold(A,real.plus,0.0)`
- Forming the logical *and* along the rows of a matrix:
  `array.fold.rows(A,bool.and,true)`.

The function `array.fold` is defined as

```
array.fold(A:α array,r:α*α->α,init:α):α
  ≡ reduce(shape(A),λi·element(A,i),r,init)
```

This definition is the basis for the transformation for introducing explicitly parallel reductions into an expression that uses `reduce`:

```
.sd.
  reduce(S,λi·g,r,init)
  ==>
  array.fold(generate(S,λi·g),r,init)
.sc.
```

The generation so introduced is then further transformed.

The partial reductions are defined using a combination of `generate` and `reduce`:

```
array.fold.rows(A:α array,r:α*α->α,init:α):α array
  ≡ generate(size(A,0),λ[i]·
    reduce(size(A,1),λ[j]·element(A,[i,j]),r,init))
```

That is, a row-wise reduction of a matrix is a vector in which each element is the reduction of the elements in the corresponding row of the matrix. A similar definition applies for column-wise reduction.

This definition can be used to optimize the evaluation of a vector of one-dimensional reductions, by evaluating all the reductions simultaneously.

```
.sd.
  generate([m],λ[i]·reduce([n],λ[j]·g,r,init))
  ==>
  array.fold.rows(generate([m,n],λ[i,j]·g),r,init)
   where r and init are independent of i
.sc.
```

This transformation has precedence over the the general transformation for reductions: that is, a combination of a vector generation and a vector reduction must be transformed before the general transformation of reductions.

Conditional expressions

Data parallel conditional expressions are denoted using the `join` function: if arrays `P`, `T` and `F` all have shape S, then `join(P,T,F)` has shape S with elements taken from `T` or `F` according to whether the corresponding element of `P` is $true$ or $false$. For example, the expression

```
join(array.greaterp(A,B),A,B)
```

forms an array in which each element is the greater of the corresponding elements of A and B.

The `join` function is defined as

```
join(M:bool array,A:α array,B:α array):α array
  ≡ generate(shape(M),λi·
    if (element(M,i)) then element(A,i) else element(B,i))
```

This definition forms the basis of the general transformation for generations of conditional expressions:

```
.sd.
  generate(S,λi·if (p) then T else F)
  ==>
  join(generate(S,λi·p),generate(S,λi·T),generate(S,λi·F))
.sc.
```

Several optimizations are possible: for example, if the predicate `p` is independent of the generating index `i`, then the conditional can be moved outside the generation:

```
.sd.
  generate(S,λi·if (p) then T else F)
  ==>
  if (p) then generate(S,λi·T) else generate(S,λi·F)
   where p is independent of i
.sc.
```

This is an optimization as only one of the two remaining generations will be evaluated.

Other optimizations are possible if the predicate is true for only a restricted region of a shape, such as a single element a row or column. Then only a scalar or a vector need be computed for the true part of the `join`, rather than an entire matrix.

λ-expressions

The Array Form does not introduce any new aspects to λ-bindings: the λ-calculus already permits arrays to be bound. But here I will discuss how array bindings are formed when converting to Array Form.

A generation of a λ-expression which binds a scalar value can be converted into data-parallel form by binding an array of values: references to the bound identifier in the body of the expression are converting into references to the appropriate element of the bound array.

```
.sd.
  generate(S,λi·λx·B (e))
  ==>
  λX·generate(S,λi·B [element(X,i)/x]) (generate(S,λi·e))
.sc.
```

An optimization is possible for λ-expressions: if the bound value is independent of the generating index, the λ-binding is moved outside the generation:

```
.sd.
  generate(S,λi·λx·B (e))
  ==>
  λx·generate(S,λi·B) (e)
   where e is independent of i
.sc.
```

Other Categories of Expressions

Other functions that occur in the Array Form are:

Shifts

The Array Form defines functions to correspond to the vector and matrix shifting functions of FPE: e.g. `vector.shiftright` and `matrix.shiftnorth`. The main purpose in using function names such as the above rather than the names employed by FPE is that some transformations manipulate functions based solely on the `vector` or `matrix` prefix. Thus, the names are more convenient for some transformations.

The conversion of generations into shifts is performed by transformation such as

```
.sd.
  generate([n],λ[i]·if (eq(i,1)) then 0.0 else element(A,[i-1])
  ==>
  vector.shiftright(A)
.sc.
```

Expansions

FPE allows the expansion of a scalar into a vector or matrix, and the row- or column-wise expansion of a vector into a matrix. The equivalent Array Form functions are `array.expand`, `array.expand.rows` and `array.expand.cols`. As with the shift functions, these names are more convenient than those used by FPE.

Expansions arise from generations in which the generating function is independent of the generating index

```
.sd.
  generate(S,λi·e)  ==> expand(S,e)
   where e is independent of i
.sc.
```

or some component of the generating index

```
.sd.
  generate([m,n],λ[i,j]·e)
  ==>
  expand.rows([m],generate([n],λ[j]·e))
   where e is independent of i
.sc.
.sd.
  generate([m,n],λ[i,j]·e)
  ==>
  expand.cols([n],generate([m],λ[i]·e))
   where e is independent of j
.sc.
```

Extractions

FPE allows parts of a matrix (such as a row or column, or the diagonal elements) to be extracted into a vector using special indexing techniques. The Array Form defines functions for the common cases `row`, `col` and `diagonal`.

Extractions arise from vector generations with generating functions that accesses elements of a matrix according to certain patterns (along a row, or along the diagonal). For example

```
.sd.
  generate([n],λ[j]·element(A,[i,J]))
  ==>
  generate([n],λ[j]·element(row(A,i),[J]))
   where i is independent of j, J is any function of j
   and where A has shape [m,n].
.sc.
.sd.
  generate([n],λ[i]·element(A,[i,i]))
  ==>
  diagonal(generate([n,n],λ[i,j]·element(A,[i,j])))
.sc.
```

(The generation introduce by the `diagonal` transformation is to allow for cases where `A` is not of shape `[n,n]`.)

Patterns

The Array Form defines functions to correspond to FPE's pattern functions: e.g. `pattern.diagonal` and `pattern.lowertri`. Again, these names are more convenient than those used by FPE.

Patterns arise from matrix generations in which the generating functions are simple boolean relations between the generating indices. For example, the relation $i \leq j$ corresponds to the lower triangle pattern. Examples of transformations to convert such generations are:

```
.sd.
  generate([n,n],λ[i,j]·bool.eq(i,j))
  ==>
  pattern.diagonal(n)
.sc
.sd.
  generate([n,n],λ[i,j]·bool.lesseqp(i,j))
  ==>
  pattern.lowertri(n)
.sc
```

Note that the boolean predicates are functions which have elementwise forms, so applications can be converted by the elementwise transformation. The transformations listed here have precedence over the elementwise transformation.

Summary

The conversion to Array Form is based primarily upon propagation rules that drive generations into expressions, creating applications of data-parallel functions *en route*, and upon rules which eliminate generations entirely, replacing them with data-parallel functions.

These rules form a simple algebra relating generations of scalar functions to data-parallel functions.

5.8.3 MVM Example

The Array Form of MVM is shown in figure 5.11.

5.8.4 Summary

After conversion to Array Form, a specification is in a form with an operational interpretation well-suited to the DAP, although the form is still pure and functional.

```
fun init
  = λn·
    λA·λB·
      fold.rows(
        map2(A,
          expand.cols(V,n),
          real.times),
        real.plus,0.0)
      (realarray.read_matrix([n,n]))
      (realarray.read_vector([n]));
```

Figure 5.11: Array form of MVM

5.9 Common Sub-expression Elimination

In this section I discuss a sub-derivation that performs common sub-expression elimination (CSE), which attempts to ensure that expensive computations are not repeated, by λ-abstracting multiple occurrences of expensive computations.

When constructing a functional specification, a programmer is encouraged to develop a clear, modular specification. A modular structure can often require more than one function to compute the same value. For example, in the full specification of the POT algorithm (discussed in the following chapter), the functions `POTstep` and `IsOk` both compute the triple matrix product $R^T * A * R$ (bound to the identifier `B`). If an implementation were to repeat this computation, it would be extremely inefficient.

It would be possible to rewrite the specification so that the product is evaluated only once, and its value passed as an argument to both functions. But a specification is supposed to be immune to concerns of efficiency: the specification of POT was written in manner that seemed most straightforward, and it should not be changed just to try to improve efficiency.

Further, even when a specification is structured to avoid multiple evaluations of the same expression (say by binding the value to an identifier), there is no guarantee that the structure will be preserved by the Unfolding and Static Evaluation stages of the derivation. Those stages assume that all structure in a specification is present to facilitate construction or understanding, not for efficiency; those stages strip away most of the structure of a specification to produce a simple form, because that facilitates manipulation of programs.

Thus, the task of structuring an implementation to avoid multiple evaluations of an expression is left to the derivation producing the implementation: the Common Sub-expression sub-derivation introduces structure (in the form of λ-bindings) in an attempt to ensure that *expensive* computations are performed only once. The introduction of this structure is performed late in the derivation, just before conversion to imperative form, by which stage the specification explicitly contains all the expressions that will be evaluated in the implementation.

Thus, given an expression such as

```
F(e,g(h(e)))
```

where the evaluation of `e` is computationally expensive, the Common Sub-expression sub-derivation λ-abstracts `e`:

```
λx·F(x,g(h(x))) (e)
```

CSE is rather more complex than simply spotting multiple occurrences of some expression and abstracting it: the main problems that arise are:

- **Identifying computationally expensive functions;**
- **Obeying scope rules;**
- **Denotational equality** as opposed to syntactic equality;
- **Sub-expressions of generating functions;**
- **Conditional expressions** and computational errors.

Identifying Computationally Expensive Expressions

The first issue to be addressed in CSE is: how to identify computationally expensive expressions. The approach taken in this thesis is to identify forms of expressions that are *not* to be considered expensive, and assume all other expressions are expensive. The inexpensive expressions are:

- Literal constants of any type.
- Identifiers.
- Shapes: it is expected that all computations on shapes have been eliminated (by static evaluation, say) and so any shapes remaining must be simple lists of dimensions.
- Applications of the tuple component function, `pi`.
- Applications of the `element` function.
- Applications of certain functions (such as integer addition and boolean conjunction) are identified as being inexpensive, if each of their arguments is itself inexpensive.

Obeying Scope Rules

An expression cannot be abstracted at just any point in a program: scope rules must be obeyed. For example, an expression that contains the bound identifier of a λ-expression cannot be moved outside the body of that λ-expression. So, for example, the following abstractions are *wrong*:

```
λx·g(x)+g(x) (e)
==>
λv·λx·v+v (e) (g(x))

f(λx·g(x)+1 (e1),λx·g(x)+2 (e2))
==>
λv·f(λx·v+1 (e1),λx·v+2 (e2)) (g(x))
```

The first is wrong because it moves the sub-expression `g(x)` outside the binding scope for `x`. The second is wrong because it abstracted two sub-expressions which have different values, though they are syntactically the same.

Such incorrect abstractions are prevented by a combination of two techniques:

- Abstraction always occurs at the 'lowest' possible position. This prevents errors of the first type, since abstraction would already have taken place *inside* the λ-body (and so would not be attempted outside the body).

- All bound identifiers are ensured to be unique. This prevents errors of the second type by syntactically distinguishing the two applications of `g`.

Denotational Equality

To function, CSE must be able to decide if two expressions are 'equivalent'. In the most general sense, 'equivalent' means the expressions compute the same value, but it is impossible to decide whether two arbitrary expressions compute the same value. Syntactic equality is simple to ascertain, but is overly restrictive for CSE; for example, it would prevent any two λ-expressions which differ only in their bound-identifiers from being considered equivalent.

So when comparing two expressions, CSE makes copies of the expressions and applies two modifications:

- all applied λ-expressions are β-reduced;
- if the two expressions are generations or reductions, the unapplied λ-expression that is the generating function of one expression is α-converted to use the same generating index as the other expression.

If the resulting expressions are syntactically equal, the original two expressions are considered equal.

Sub-Expressions of Generating Functions

Suppose an algorithm is to compute the following two values:

```
generate([n],λ[i]·g(f(i)))  and reduce([n],λ[j]·h(f(j)),r,init)
where f is a computationally expensive function.
```

The evaluation of each of these two expressions requires the evaluation of `f(i)` for `i` in the range 1 to n. Thus, the values of `f(i)` are computed and stored in an array, and the generation and reduction access the elements of this array:

```
generate([n],λ[i]·g(V@[i]))  and reduce([n],λ[j]·h(V@[j]),r,init))
where V is generate([n],λ[k]·f(k)).
```

Conditional Expressions

Abstracting an expression can cause a problem by moving an expression out of a limb of a conditional expression. For example, the abstraction

```
if (p)
then f(e,e1)
else if (q)
  then f(e,e2)
  else e3
  end
end
```

==>

```
λv·
  if (p)
  then f(v,e1)
  else if (q)
    then f(v,e2)
    else e3
    end
  end
(e)
```

is *wrong* since it causes the expression `e` to be evaluated regardless of the values of `p` and `q`, whereas in the original expression, `e` is *not* evaluated if both `p` and `q` are false — disaster if `e` can cause a computational error.

Note, however, that the following transformation is correct:

```
if (p)                      λv·
then f(e,e1)                  if (p)
else if (q)                   then f(v,e1)
  then f(e,e2)       ==>      else if (q)
  else f(e,e3)                  then f(v,e2)
  end                           else f(v,e3)
end                             end
                              end
                            (e)
```

since `e` occurs in all three limbs of the conditionals and so is guaranteed to be evaluated regardless of the values of the predicates.

If-levels The method used to ensure that abstraction does not cause an expression to incorrectly move outside a conditional expression is to assign each expression an *if-level* which indicates how deeply that expression is nested inside conditional expressions; transformations must not abstract an expression in such a way that its if-level would change.

So, for example, the above expression before abstraction would be marked

```
if (p:0)
then f(e:1,e1:1):1
else if (q:1)
  then f(e:2,e2:2):2
  else f(e:2,e3:2):2
  end:1
end:0.
```

where each expression's if-level is one greater than that of the immediately enclosing conditional expression. The if-level forms part of an expression: two expressions are considered equal only if they have the same if-level.

Then, if an expression `E` contains multiple occurrences of an expression `e`, a λ-abstraction of `e` can be formed around `E` only if `E` and `e` have the same if-level.

Thus, given the expression

```
if (p:0)
then f(e:1,e1:1):1
else if (q:1)
  then f(e:2,e2:2):2
  else e3:2
  end:1
end:0
```

the abstraction of `e` is not performed because `e` has if-levels 1 and 2 while the (outer) conditional expression as a whole has if-level 0.

Lowering If-Levels The rule given above for if-levels is overly restrictive. For example, the rule as stated above would assign if-levels as shown below:

```
if (p:0)
then f(e:1,a:1):1
else g(e:1,b:1):1
end:0
```

With such if-levels, the sub-expression `e` cannot be abstracted from the conditional because `e` is at a higher if-level than the conditional. However, because `e` occurs in both limbs of the conditional (and not inside a further conditional), it is guaranteed to be evaluated if the conditional is evaluated, so the if-level of `e` can be lowered to that of the conditional without the risk of introducing computational errors, allowing abstraction

```
λx·if (p:0)
  then f(x:1,a:1):1
  else g(x:1,b:1):1
  end:0
(e:0):0
```

Thus, the following rule holds: if a conditional at level `i` contains a sub-expression `e` at level `i+1` in both of its limbs, then the level of both `e`s can be lowered to `i`.

5.9.1 Method

The implementation of CSE is detailed in appendix B.

5.9.2 MVM Example

There are no repeated expressions in the MVM example, so the Common Sub-expression sub-derivation has no effect on the example. It is however important for the extended examples (chapter 6).

5.10 Summary

The philosophy of constructing specifications for clarity rather than efficiency requires a derivation to undertake the task of structuring a specification to avoid repeating expensive computations. The Common Sub-expression stage of the derivation λ-abstracts multiple occurrences of manifestly equivalent expressions whose computation is non-trivial.

One limitation of the sub-derivation is that there may be occasions when human reasoning is required to decide that two expressions which are not manifestly equivalent do nevertheless compute the same value; the sub-derivation would fail to abstract such expressions.

A further limitation of the sub-derivation is that abstraction is restricted to within a single function definition: the same expression occurring in different functions would not be abstracted. This restriction is not important for the examples considered in this thesis (and, I believe, for the majority of numerical algorithms) because function unfolding collapses each specification into a single function.

5.11 Fortran Plus Enhanced

The derivation for implementing the Array Form in Fortran Plus Enhanced is similar to that described in section 4.5.1 for converting the λ-calculus into Fortran77. The main extensions to that derivation required for FPE are transformations to implement the basic array functions `generate` and `reduce`[6] and to implement the data parallel conditional `join`. Other extensions are mostly for matters of syntax: for example, to accommodate the declaration of vectors and matrices and the novel indexing techniques employed by FPE.

[6]Not all applications of these functions will always be eliminated by the Array Form derivation.

```
integer n
parameter(n)
real init(*n)
real A(*n,*N)
real V(*n)
call readmatrix(A,n,n)
call readvector(V,n)
init = sumc(A*matr(V,n))
```

Figure 5.12: Fortran Plus Enhanced version of MVM

The Fortran Plus Enhanced version of the MVM example is shown in figure 5.12.

5.12 Summary

This chapter has discussed the stages in a derivation to convert an SML specification into a Fortran Plus Enhanced implementation for the DAP array processor:

$$\begin{aligned}&\text{SML} \longrightarrow \text{DAP}\\ &\equiv \text{SML} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \longrightarrow \text{Static Evaluation}\\ &\quad \longrightarrow \text{Array Form} \longrightarrow \text{Common Sub-expression Elimination} \longrightarrow \text{FPE.}\end{aligned}$$

- A specification is constructed in a form that facilitates human understanding;
- it is then recast into a form that simplifies automated manipulation (by transformations);
- the specification is then reduced to a simple, minimal form by unfolding definitions and evaluating expressions;
- it is then converted into an explicitly data-parallel form with operation semantics suited to execution by an array processor;
- the specification is restructured to eliminate repeated computations;
- finally, it is translated into an imperative form suited to the DAP compiler.

Each stage of the derivation was illustrated with matrix-vector multiplication; an important and non-trivial operation, but still comparatively simple. It remains to be shown how the derivation performs with a more complex algorithm.

5.13 Chapter Notes

i (page 79)
Curiously, though, developers of compilers for imperative languages often argue against unfolding precisely because it *increases* the number of cache rebuilds (caused by the bodies of subroutines becoming too large to be able to fit entirely into the cache). However, as illustrated earlier in the discussion on Static Evaluation, techniques can be employed in a functional context to reduce the size of function bodies after unfolding; such techniques are unlikely to be available for subroutines in an imperative context.

Chapter 6

Extended Examples

In this chapter, I present some extended examples for the DAP derivation:

- The full derivation of the Conjugate Gradient algorithm from SML specification, through each intermediate form, to the Fortran Plus Enhanced implementation for the DAP.
- The SML specification and FPE implementation of the POT algorithm; the performance of the FPE implementation is compared with that of a hand-crafted implementation.

The program sections included in this chapter are taken from derivation traces. I have made some attempt to improve the readability of some of the more awkward sections, but I generally prefer not to change traces as it is easy to introduce errors; the forms of CG after the Static Evaluation stage have been left in curried form (the form used by most stages of the derivation).

6.1 Introduction

A small algorithm, such as the matrix-vector multiplication algorithm in the previous chapter, is useful for illustrating the process of derivation. However, an important aspect of transformations is that they should, when possible, be generally applicable. There is thus the need to illustrate the effectiveness of the transformations when applied to more complex problems, in which operations such as matrix-vector multiplication, vector addition, vector scaling, etc. become combined together.

In this chapter, I present the derivation of implementations of two fairly complex algorithms: a bi-conjugate gradient algorithm (similar to the conjugate gradient algorithm discussed in section 2.2.3) and the POT algorithm (section 2.2.3). These derivations are intended to provide a more persuasive illustration than matrix-vector multiplication of the effectiveness of the derivation.

N.B.: some of the program sections included in this chapter are quite long, and it has usually been necessary to reduce the type size for presentation. There is no other significance in the size changes!

6.2 Conjugate Gradient

In this section, I discuss the Conjugate Gradient algorithm and its specification; following sections discuss the derivation of an implementation of CG for the AMT DAP array processor.

The Conjugate Gradient algorithm uses an iterative process to compute (an approximation to) the vector x of order n satisfying the equation $Ax = b$ where A is a positive, definite, symmetric matrix of order $n \times n$ and b is a vector of order n.

[80] p152

```
To solve $Ax = b$, where $A$ is a positive definite symmetric $n \times n$ matrix:
    Set an initial approximation vector $x_0$,
    calculate the initial residual $r_0 = b - Ax_0$,
    set the initial search direction $p_0 = r_0$;
    then, for $i = 0, 1, \ldots$,
      (a) calculate the coefficient $\alpha_i = p_i^T r_i / p_i^T A p_i$,
      (b) set the new estimate $x_{i+1} = x_i + \alpha_i p_i$,
      (c) evaluate the new residual $r_{i+1} = r_i - \alpha_i A p_i$,
      (d) calculate the coefficient $\beta_i = -r_{i+1} A p_i / p_i^T A p_i$,
      (e) determine the new direction $p_{i+1} = r_{i+1} + \beta_i p_i$,
    continue until either $r_i$ or $p_i$ is zero.
```

Figure 6.1: Mathematical definition of Conjugate Gradient

The mathematical definition of Conjugate Gradient was presented earlier (section 2.2.3), and is repeated here for reference (figure 6.1). The name "Conjugate Gradient" often refers to a class of algorithms which employ the basic method defined in this figure, rather than to a specific algorithm. The particular version used in this chapter is known as a *bi-conjugate gradient* algorithm, though I shall continue to refer to it simply as conjugate gradient or CG. The functional specification is shown in figure 6.2. This specification is a modification of a specification supplied by some colleagues: as far as I am aware, no ‘mathematical definition’ of this precise form was constructed; in effect, the mathematicians developing the algorithm used a functional specification as their definition.

```
type cgstate = real vector*real vector*real vector*real vector;
fun cgiters(a:real matrix,b:real vector):cgstate
 = let
  (* Terminating condition.*)
  fun isok((x,r,p,q):cgstate):bool
   = innerproduct(r,r)<epsilon;

  (* One iteration.*)
  fun cgiter((x,r,p,q):cgstate):cgstate
   = let
    val rr:real = innerproduct(r,r);
    val alpha:real = rr/innerproduct(q,q);
    val x′:real vector = x+p*alpha;
    val atq:real vector = transpose(a)*q;
    val r′:real vector = r-atq*alpha;
    val beta:real = innerproduct(r′,r′)/rr;
    val p′:real vector = r′+p*beta;
    val q′:real vector = a*r′+q*beta
   in
    cgstate(x′,r′,p′,q′)
   end
 in
  iterate(cgiter,cgstate(x0,r0,p0,q0),isok)
 end
```

Figure 6.2: SML specification of *bi-conjugate* gradient

- The algorithm is based upon manipulation of a set of vectors x,r,p and q (x being the approximation to the solution); the type `cgstate` is defined to represent the set of vectors, as a 4-tuple of real vectors. Instances of the `cgstate` type are constructed using the function `cgstate`.
- The function `cgiters` takes A and b as arguments and returns a `cgstate` containing the approximate solution.
- The algorithm is iterative in nature, so `cgiters` is an application of the `iterate` library function.
  - The first argument to `iterate` is a function defining the computation that is to be repeated: `cgiter`.
  - The second argument is a value (an instance of `cgstate`) with which to begin the repetition.
  - The third argument is a function which determines when the repetition is to cease (i.e. when the approximation is sufficiently accurate): `isok`.
- The function defining the repeated computation, `cgiter`, takes one argument of type `cgstate` and returns a value of the same type. In the specification, pattern matching is used to bind the names `x`,`r`,`p` and `q` to the four components of the `cgstate` argument.
- The body of `cgiter` computes the next set of vectors as local values `x′`, `r′`, `p′` and `q′` and returns these values as an instance of `cgstate`.
- For brevity, the computation of the initial values `x0`, `r0`, `p0` and `q0` is not shown.
- The bulk of the computational costs are incurred by the two matrix-vector products, in the computation of `atq` and `q′`.

Although the functional specification is not based directly upon a mathematical definition, I believe that the specification is still clear and natural: anyone with a basic knowledge of linear algebra should be able to follow the specification.

6.2.1 λ-Calculus Form of CG

I now discuss the derivation of a DAP implementation (expressed in Fortran Plus Enhanced) of Conjugate Gradient from an SML specification.

An SML specification is designed to be simple for people to read and understand, but the syntax is often cumbersome to manipulate automatically. So, an SML specification is converted into the λ-calculus form, which has a very simple syntax, facilitating transformation.

$$\boxed{\begin{array}{l}\text{SML} \longrightarrow \text{DAP} \\ \equiv \text{SML} \longrightarrow \lambda\text{-Calculus} \ldots.\end{array}}$$

```
fun cgiters(a:real matrix,b:real vector)
   :cgstate
 = let
  (* Terminating condition.*)
  fun isok((x,r,p,q):cgstate):bool
   = innerproduct(r,r)<epsilon;

  (* One iteration.*)
  fun cgiter((x,r,p,q):cgstate):cgstate
   = let
    val rr:real = innerproduct(r, r);
    val alpha:real = rr/innerproduct(q, q);
    val x':real vector = x+p*alpha;
    val atq:real vector = transpose(a)*q;
    val r':real vector = r-atq*alpha;
    val beta:real = innerproduct(r',r')/rr;
    val p':real vector = r'+p*beta;
    val q':real vector = a*r'+q*beta
   in
    cgstate(x',r',p',q')
   end
 in
  iterate(cgiter,cgstate(x0,r0,p0,q0),isok)
 end
```

$\longrightarrow$

```
fun isok:cgstate->bool
 = λx001·
  λx:real vector·
   λr:real vector·
    λp:real vector·
     λq:real vector·
      innerproduct(r,r)
     (pi(4,x001))
    (pi(3,x001))
   (pi(2,x001))
  (pi(1,x001));
fun cgiter:real matrix->(cgstate->cgstate)
 =  λa:real matrix·
 λx001·
  λx:real vector·
   λr:real vector·
    λp:real vector·
     λq:real vector·
      λrr:real·
       λalpha:real·
        λx':real vector·
         λatq:real vector·
          λr':real vector·
           λbeta:real·
            λp':real vector·
             λq':real vector·
              cgstate(x',r',p',q')
             (op.plus(
               op.times(a,r'),
               op.times(q,beta))
             )
            (op.plus(r',op.times(p,beta)))
           (op.quotient(rr,innerproduct(r',r')))
          (op.difference(r,op.times(atq,alpha)))
         (op.times(matrix.transpose(a),q))
        (op.plus(x,op.times(p,alpha)))
       (op.quotient(rr,innerproduct(q,q)))
      (innerproduct(r,r)
     (pi(4,x001))
    (pi(3,x001))
   (pi(2,x001))
  (pi(1,x001));
fun cgiters:real matrix*real vector->real vector
 = λa·λb·iterate(cgiter(a),
   cgstate(x0,r0,p0,q0),isok)
```

Figure 6.3: Conversion to λ-calculus form of CG

The main changes that occur to CG when it is converted to λ-calculus form (figure 6.3) are:

- The definitions of the functions `cgiters` and `isok` become top-level definitions.
- Local values become λ-expressions (in particular, note the λ-bindings of x′, r′, p′ and q′ in `cgiter`).

- Prefix functions are used instead of operators. (The function `pi` selects a designated component of a tuple.)
- The SML specification used pattern matching to bind `x`, `r`, `p` and `q` to components of the `cgstate` argument of the iteration function (`cgiter`); this pattern matching has been removed by introducing a simple name (`x001`, of type `cgstate`) for the argument and introducing `x`, `r` `p` and `q` in λ-expressions which bind those names to the appropriate components of `x001`.

6.2.2 Unfolded Form of CG

A functional specification should be designed in an abstract manner, using functional decomposition to facilitate human understanding. However, functional decomposition is often detrimental to execution performance and to manipulation of the specification, so function applications are eliminated, where possible, by unfolding function definitions.

$$\boxed{\begin{array}{l}\text{SML} \longrightarrow \text{DAP}\\ \equiv \text{SML} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \ldots\end{array}}$$

When CG is unfolded, the entire specification collapses into a single function definition, with the bulk of the computation being performed by the application of `iterate` (which was contained in the function `cgiters` in the SML specification).

The unfolded form of CG is too large to present in its entirety here, but to provide an illustration of the nature of the unfolded form, I include in figure 6.4 the unfolded form of the expression for `x`$'$.

```
op.plus(x,op.times(p,alpha))
```

$\longrightarrow$

```
λg312:real ma·
  λg311:real ma·
    multiarray.generate
      (multiarray.shape (g312:real ma))
      (λg170·
        reals.plus
          (multiarray.element
            (g312:real ma)
            (g170):real
          )
          (multiarray.element
            (g311:real ma)
            (g170):real
          ):real
      ):real ma
  (λg374:real ma·
    λg373:real·
      multiarray.generate
        (multiarray.shape (g374:real ma))
        (λg167·
          reals.times
            (multiarray.element
              (g374:real ma)
              (g167):real
            )
            (g373:real):real
        ):real ma
    (g439:real):real ma
  (pi (3:int)
    (g686:(real ma*real ma*real ma*real ma*int)):real ma
  ):real ma
  ):real ma
(pi (1:int)
  (g686:(real ma*real ma*real ma*real ma*int)):real ma
):real ma
```

Figure 6.4: The λ-calculus and unfolded forms of `x`$'$

- `g312` is bound to the first component, `x`, of the `cgstate` argument (`g686`) to the iteration function.

- The value bound to `g311` is `p` scaled by α. (The value denoted by `alpha` in the λ-calculus form is bound to the identifier `g439`, which in turn is bound to `g373`; the value bound to `alpha` is not shown.)

All of the bindings shown in the figure can be eliminated, and are eliminated, by Static Evaluation.

6.2.3 Statically Evaluated Form of CG

SML ⟶ DAP
≡ SML ⟶ λ-Calculus ⟶ Unfolding ⟶ Static Evaluation ....

```
fun isok:cgstate->bool
 = λx001·
  λx:real vector·
   λr:real vector·
    λp:real vector·
     λq:real vector·
      innerproduct(r,r)
     (pi(4,x001))
    (pi(3,x001))
   (pi(2,x001))
  (pi(1,x001));
fun cgiter:real matrix->(cgstate->cgstate)
 = λa:real matrix·
 λx001·
  λx:real vector·
   λr:real vector·
    λp:real vector·
     λq:real vector·
      λrr:real·
       λalpha:real·
        λx′:real vector·
         λatq:real vector·
          λr′:real vector·
           λbeta:real·
            λp′:real vector·
             λq′:real vector·
              cgstate(x′,r′,p′,q′)
             (op.plus(
                op.times(a,r′),
                op.times(q,beta))
             )
            (op.plus(r′,op.times(p,beta)))
           (op.quotient(rr,innerproduct(r′,r′)))
          (op.difference(r,op.times(atq,alpha)))
         (op.times(matrix.transpose(a),q))
        (op.plus(x,op.time(p,alpha)))
       (op.quotient(rr,innerproduct(q,q)))
      (innerproduct(r,r)
     (pi(4,x001))
    (pi(3,x001))
   (pi(2,x001))
  (pi(1,x001));
fun cgiters:real matrix*real vector->real vector
 = λa·λb·iterate(cgiter(a),
   cgstate(x0,r0,p0,q0),isok)
```

⟶

```
iterate
 (λg686:real array*real array*...·
  λg436:real array·
   tuple (X′,g436,P′,Q′)
  (R′),
 tuple(
  generate([n],
   λ[g1114]·0.0)
  g457,
  g457,
  generate([n],
   λ[g1123]·
    reduce([n],
     λ[g1122]·real.times(
      element(g458,[g1123,g1122]),
      element(g457,[g1122 ])),
     real.plus,0.0)
   )
  ),
 λg430:real ma*real ma*...·
  real.lessp(
   real.sqrt(reduce([n],
    λ[g1133]·
     real.sqr(element(pi(2,g430),[g1133])),
    real.plus,0.0)
    )
   ),
   1.0E-14
  )
```

Figure 6.5: The λ-calculus and Statically Evaluated forms of CG

Functional decomposition can introduce overheads for the management of the intermediate data structures that are created as arguments for and return values from functions; function unfolding reduces the number of function applications, but not the number of intermediate data structures. Static Evaluation attempts to remove these data structures, leaving the specification in a very simple, concise form.

The Statically Evaluated form of CG, though much shorter than the Unfolded form, is still rather large: it is presented here in sections. Figure 6.5 shows the application of `iterate`, which performs most of the computation. I have introduced labels *X'*, *P'*, *Q'* and *R'* into the expression to indicate large expressions that are detailed below: the labels are for presentation purposes only — they do not correspond to, say, names introduced by λ-expressions. (Note that the figure also shows the λ-calculus form of these expressions, rather than the unfolded form from which the evaluated form is derived; as I previously stated, the unfolded form is too large to be presented here.)

- The bindings for `x`, `r`, `p` and `q` (components of `g436`, the argument of the iteration function) have been reduced.
- The functions `cgiters`, `cgiter` and `isok` have been unfolded: the diagram on the right shows the application of `iterate` after unfolding (and evaluation).
- The instance of `cgstate` returned by the iteration function is now constructed explicitly as a tuple. (i.e. the definition of the function `cgstate` has been unfolded.)
- The bindings for $\mathtt{x}'$, $\mathtt{p}'$ and $\mathtt{q}'$ have been reduced (so the expressions to which the names were bound, rather than the names themselves, now appear directly as components of the tuple expression).
- The binding for $\mathtt{r}'$ has not been reduced, as it is used in multiple places in the tuple expression.

The expressions denoted by the labels *X'*, etc. are detailed below.

X' The expression labeled *X'* (figure 6.6) corresponds to the expression bound to the name $\mathtt{x}'$ in the SML specification: the SML expression is shown beside the Evaluated form below for comparison.

```
val rr:real = innerproduct(r, r);
val alpha:real = rr/innerproduct(q, q);
val x':real vector = x+p*alpha;
```

$\longrightarrow$

```
generate([n],
   λ[g1073]·
    real.plus(
     element(pi(1,g686),[g1073]),
     real.times(
      element(pi(3,g686),[g1073]),
      real.quotient(
       reduce([n],
        λ[g1093]·
         real.times(
          element(pi(2,g686),[g1093])
          element(pi(2,g686),[g1093])),
        real.plus,0.0),
       reduce([n],
        λ[g1105]·
         real.times(
          element(pi(4,g686),[g1105]),
          element(pi(4,g686),[g1105])),
        real.plus,0.0)
       )
      )
     ):real
   ):real array
```

Figure 6.6: Statically Evaluated form of *X'*

The functions for the addition of two vectors, for scaling a vector and for computing the inner product of two vectors are defined in terms of `generate` and `reduce`. Unfolding and Static Evaluation replace applications of the vector functions with equivalent applications of `generate` and `reduce`.

The generation is over a shape $[n]$ as this is the shape of each of the four vectors manipulated by the algorithm. The generating function combines: the computation of the inner product involving `r`; the computation of the inner product involving `q`; the scaling of `p`; and the addition of `x` with the scaled `p`.

- The addition of the two vectors in the SML form gives rise to the real addition at the outer-level of the generating function in the Evaluated form.
- The scaling of p gives rise to the real multiplication which is an argument to the addition.
- The inner products give rise to the two reductions.

R' The expression labeled *R'* (figure 6.7) corresponds to the expression bound to the name r′ in the SML specification.

```
val rr:real = innerproduct(r, r);
val alpha:real = rr/innerproduct(q, q);
val atq:real vector = transpose(a)*q;
val r':real vector = r-atq*alpha;
```

$\longrightarrow$

```
generate([n],
  λ[g1113]·
   real.difference(
    element(pi(2,g686),[g1113]),
    real.times(
     reduce([n],
      λ[g1111]·
       real.times(
        element(g458,[g1111,g1113]),
        element(pi(4,g686),[g1111]))
      real.plus,0.0),
     real.quotient(
      reduce([n],
       λ[g1106]·
        real.times(
         element(pi(2,g686),[g1106]),
         element(pi(2,g686),[g1106])),
       real.plus,0.0),
      reduce([n],
       λ[g1107]·
        real.times(
         element(pi(4,g686),[g1107]),
         element(pi(4,g686),[g1107]))
       real.plus,0.0)
      )
     )
    ):real
  ):real array
```

Figure 6.7: Statically Evaluated form of *R'*

The SML form specifies the subtraction of atq*alpha from r. This vector subtraction becomes a vector generation once function definitions have been unfolded. The shape of the generation is $[n]$ as this is the shape of all the vectors manipulated by the algorithm.

The generating function combines the scaling of atq, the subtraction from r, the computation of atq (involving the multiplication of q by a transposed) and the computation of alpha (involving the computation of two inner products, which give rise to two of the reductions in the evaluated form).

P' The expression labeled *P'* (figure 6.8) corresponds to the expression bound to the name p′ in the SML specification.

As with *R'*, unfolding matrix and vector functions gives rise to generations and reductions over the shape $[n]$ common to all the vectors manipulated by the algorithm. The two reductions derive from the two inner products required for the computation of beta.

Q' The expression labelled *Q'* (figure 6.9) corresponds to the expression bound to the name q′ in the SML specification.

As above, unfolding matrix and vector functions gives rise to generations and reductions over the shape $[n]$. The generating function combines the computation of the product of a with r′, the scaling of q by beta and the addition of the same to that product and the computation of beta.

```
val rr:real = innerproduct(r, r);
val beta:real = innerproduct(r',r')/rr;
val p':real vector = r'+p*beta;
```

$\longrightarrow$

```
generate([n],
  λ[g1000]·
   real.plus(
    element(g436,[g1000]),
      real.times(
       element(pi(3,g686),[g1000]),
       real.quotient(
        reduce([n],
         λ[g1003]·
          real.times(
           element(g436,[g1003]),
           element(g436,[g1003]))
         real.plus,0.0),
        reduce([n],
         λ[g1094]·
          real.times(
           element(pi(2,g686),[g1094]),
           element(pi(2,g686),[g1094]))
         real.plus,0.0)
        )
       )
      ):real
    ):real array
```

Figure 6.8: Statically Evaluated form of *P'*

```
val rr:real = innerproduct(r, r);
val beta:real = innerproduct(r',r')/rr;
val q':real vector = a*r'+q*beta
```

$\longrightarrow$

```
generate([n],
 λ[g998]·
 real.plus(
  reduce([n],
  λ[g990]·
   real.times(
    element(g458,[g998,g990]),
    element(g436,[g990])),
   real.plus,0.0),
  real.times(
  element(pi(4,g686),[g998]),
  real.quotient(
   reduce([n],
   λ[g1004]·
    real.times(
    element(g436,[g1004]),
    element(g436,[g1004])),
   real.plus,0.0),
  reduce([n],
   λ[g1095]·
    real.times(
     element(pi(2,g686),[g1095]),
     element(pi(2,g686),[g1095])),
    real.plus,0.0)
   )
  )
  ):real
 ):real array
```

Figure 6.9: Statically Evaluated form of *Q'*

6.2.4 Array Form of CG

The Statically Evaluated form expresses array operations in an implicitly data-parallel manner, using `generate` and `reduce` (which arise from the definitions of the matrix and vector library functions). The Array Form expresses array operations in an *explicitly* data-parallel manner, using a set of simple functions which have data-parallel operational semantics.

SML $\longrightarrow$ DAP
$\equiv$ SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ Array Form ....

```
iterate
 (λg686real array*real array*...·
  λg436:real array·
   tuple (X',g436,P',Q')
  (R'),
 tuple(
  generate([n],
   λ[g1114]·0.0)
  g457,
  g457,
  generate([n],
   λ[g1123]·
    reduce([n],
     λ[g1122]·real.times(
      element(g458,[g1123,g1122]),
      element(g457,[g1122 ])),
     real.plus,0.0)
   )
  ),
 λg430:real ma*real ma*...·
  real.lessp(
   real.sqrt(reduce([n],
    λ[g1133]·
     real.sqr(element(pi(2,g430),[g1133])),
    real.plus,0.0)
    )
   ),
   1.0E-14
  )
```

$\longrightarrow$

```
iterate
 (λg686·
  λg436·
   tuple (X') (g436) (P') (Q')
  (R')
 )
 (tuple
  (0.0)
  (g457)
  (g457)
  (matrix.reduce.rows (real.plus) (0.0)
   (real.times.array
    (g458)
    (matrix.expand.rows (n)
     (g457)
    )
   )
  )
 )
 (λg430·
  real.lessp
   (real.sqrt
    (reduce.array (real.plus) (0.0)
     (real.times.array
      (pi(2)(g430))
      (pi(2)(g430))
     )
    )
   )
   (1.0E-14)
 )
```

Figure 6.10: Array Form of iteration

The Array Form sub-derivation successfully converts all of the array operations in CG into explicit data-parallel form. For example the expression

```
generate([n],
 λ[g1123]·
  reduce([n],
   λ[g1122]·real.times(
    element(g458,[g1123,g1122]),
    element(g457,[g1122 ])),
   real.plus,0.0)
 )
```

has become

```
matrix.reduce.rows (real.plus) (0.0)
   (real.times.array
    (g458)
    (matrix.expand.rows (n)
     (g457)
    )
   )
```

In the former expression, any data-parallelism is only implicit: a direct Fortran implementation of this expression would sequentially compute each of the n elements of a vector; the computation of each element would involve n multiplications and $(n-1)$ additions, all performed sequentially.

In the Array Form expression, the data-parallelism is explicit: the n elements are computed simultaneously by summing along the rows of a matrix; the matrix itself is formed as an elementwise product of two matrices, one of which is formed by duplicating a vector in a row-wise manner.

The only major sequential operation remaining in the Array form of CG is the repeated application of the iteration function. The Array version of the application of `iterate` is shown in figure 6.10; as before, large expressions have been replaced with labels in the figure; the expressions are shown below.

Figure 6.11 shows the expression that was labelled *X'* in the Evaluated form, and its Array Form equivalent.

```
generate([n],
   λ[g1073]·
    real.plus(
     element(pi(1,g686),[g1073]),
     real.times(
      element(pi(3,g686),[g1073]),
      real.quotient(
       reduce([n],
        λ[g1093]·
         real.times(
          element(pi(2,g686),[g1093])
          element(pi(2,g686),[g1093])),
        real.plus,0.0),
       reduce([n],
        λ[g1105]·
         real.times(
          element(pi(4,g686),[g1105]),
          element(pi(4,g686),[g1105])),
        real.plus,0.0)
       )
      )
     ):real
   )
```

→

```
(real.plus.array
    (pi(1)(g686))
    (real.times.array
     (pi(3)(g686))
     (real.quotient
      (reduce.array (real.plus) (0.0)
       (real.times.array
        (pi(2)(g686))
        (pi(2)(g686))
       )
      )
      (reduce.array (real.plus) (0.0)
       (real.times.array
        (pi(4)(g686))
        (pi(4)(g686))
       )
      )
     )
    )
   )
```

Figure 6.11: Evaluated and Array forms of *X'*

- The generation of real addition has become an application of elementwise addition.
- The first argument to the addition has become the whole vector that is the first component of `g686`.
- The argument to the addition was a real multiplication; it has become an application of elementwise multiplication.
- The first argument to this multiplication has become the whole vector that is the third component of `g686`.
- The second argument to this multiplication was a real quotient; it remains a real quotient (i.e. a scalar value) since the entire quotient is independent of the generating index. (A scalar is equivalent to an expansion.)
- The values being divided were reductions; they are now elementwise reductions.
- The generating functions of the reductions were real products; the vectors being reduced in the elementwise reductions are elementwise products, with arguments that are whole vectors (components of `g686`).

The expressions labeled *R'*, *P'* and *Q'* are shown in the following figures.

```
generate([n],
  λ[g1113]·
   real.difference(
    element(pi(2,g686),[g1113]),
    real.times(
     reduce([n],
      λ[g1111]·
       real.times(
        element(g458,[g1111,g1113]),
        element(pi(4,g686),[g1111]))
      real.plus,0.0),
     real.quotient(
      reduce([n],
       λ[g1106]·
        real.times(
         element(pi(2,g686),[g1106]),
         element(pi(2,g686),[g1106])),
       real.plus,0.0),
      reduce([n],
       λ[g1107]·
        real.times(
         element(pi(4,g686),[g1107]),
         element(pi(4,g686),[g1107]))
       real.plus,0.0)
      )
     )
    ):real
  )
```

$\longrightarrow$

```
(real.difference.array
   (pi(2)(g686))
   (real.times.array
    (matrix.reducecols (real.plus) (0.0)
     (real.times.array
      (g458)
      (matrix.expand.cols (n)
       (pi(4)(g686))
      )
     )
    )
    (real.quotient
     (reduce.array (real.plus) (0.0)
      (real.times.array
       (pi(2)(g686))
       (pi(2)(g686))
      )
     )
     (reduce.array (real.plus) (0.0)
      (real.times.array
       (pi(4)(g686))
       (pi(4)(g686))
      )
     )
    )
   )
  )
```

Figure 6.12: The Evaluated and Array of *R'*

```
generate([n],
  λ[g1000]·
   real.plus(
    element(g436,[g1000]),
      real.times(
       element(pi(3,g686),[g1000]),
       real.quotient(
        reduce([n],
         λ[g1003]·
          real.times(
           element(g436,[g1003]),
           element(g436,[g1003]))
         real.plus,0.0),
        reduce([n],
         λ[g1094]·
          real.times(
           element(pi(2,g686),[g1094]),
           element(pi(2,g686),[g1094]))
         real.plus,0.0)
        )
       )
      ):real
    )
```

$\longrightarrow$

```
(real.plus.array
    (g436)
    (real.times.array
     (pi(3)(g686))
     (real.quotient
      (reduce.array (real.plus) (0.0)
       (real.times.array (g436) (g436) )
      )
      (reduce.array (real.plus) (0.0)
       (real.times.array
        (pi(2)(g686))
        (pi(2)(g686))
       )
      )
     )
    )
   )
```

Figure 6.13: The Evaluated and Array forms of *P'*

```
generate([n],
 λ[g998]·
 real.plus(
  reduce([n],
  λ[g990]·
   real.times(
    element(g458,[g998,g990]),
    element(g436,[g990])),
   real.plus,0.0),
  real.times(
  element(pi(4,g686),[g998]),
  real.quotient(
   reduce([n],
   λ[g1004]·
    real.times(
    element(g436,[g1004]),
    element(g436,[g1004])),
   real.plus,0.0),
  reduce([n],
   λ[g1095]·
    real.times(
     element(pi(2,g686),[g1095]),
     element(pi(2,g686),[g1095])),
    real.plus,0.0)
   )
  )
  ):real
 ):real array
```

$\longrightarrow$

```
(real.plus.array
    (matrix.reducerows (real.plus) (0.0)
     (real.times.array
      (g458)
      (matrix.expand.rows (n) (g436) )
     )
    )
    (real.times.array
     (pi(4)(g686))
     (real.quotient
      (reduce.array (real.plus) (0.0)
       (real.times.array (g436) (g436) )
      )
      (reduce.array (real.plus) (0.0)
       (real.times.array
        (pi(2)(g686))
        (pi(2)(g686))
       )
      )
     )
    )
   )
```

Figure 6.14: The Evaluated and Array forms of *Q*'

6.2.5 Common Sub-expression Eliminated Form of CG

The Unfolding and Static Evaluation sub-derivations remove most of the structure that is present in the SML specification, on the assumption that such structure is present primarily for human understanding. One result of removing this structure is that the program contains several expressions that are computed more than once. The Common Sub-expression Elimination sub-derivation removes multiple occurrences of each (non-trivial) expression by λ-abstracting the expression (i.e. by binding the expression to a name, and replacing all other occurrences of the expression with that name).

SML $\longrightarrow$ DAP
$\equiv$ SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ Array Form $\longrightarrow$ Common Sub-expression Elimination . . . .

The form of CG after Common Sub-expression elimination is essentially the same as the Array Form, except that some additional λ-bindings have been introduced. The bindings and the uses of the names introduced by the bindings have been underlined in figure 6.15. (NB: for brevity, the figure shows only the iteration function.)

- `g1296` is a real value, the inner-product involving the second component, `r`, of `g1284` (the argument of the iteration function). This binding corresponds to that for `rr` in the SML specification.

- `g1295` is a real value, the quotient of the previous binding and the inner-product involving the fourth component, `q`, of `g1284`. This binding corresponds to that for `alpha` in the SML specification.

- `g1283` is a real vector corresponding to the binding for `r′`.

```
lambda g1284 @
 lambda g1296:real @
  lambda g1295: real @
   lambda g1283:real array @
    lambda g1292:real @
     tuple
      (real.plus.array
       (pi (1) (g1284))
       (real.times.array
        (pi (3) (g1284))
        (g1295)
       )
      )
      (g1283)
      (real.plus.array
       (g1283)
       (real.times.array
        (pi (3) (g1284))
        ( g1292)
       )
      )
      (real.plus.array
       (matrix.reduce.rows (real.plus) (0.0)
        (real.times.array
         (g1287)
         (multiarray.expand.rows (n) (g1283))
        )
       )
       (real.times.array
        (pi (4) (g1284))
        ( g1292)
       )
      )
    end
```

```
        (real.quotient
         (reduce.array (real.plus) (0.0)
          (real.times.array
           (g1283)
           (g1283)
          )
         )
         (g1296)
        )
       end
       (real.difference.array
        (pi (2) (g1284))
        (real.times.array
         (matrix.reducecols (real.plus) (0.0)
          (real.times.array
           (g1287)
           (multiarray.expand.cols (n)
            (pi (4) (g1284))
           )
          )
         )
         (g1295)
        )
       )
      end
      (real.quotient
       (g1296)
       (reduce.array (real.plus) (0.0)
        (real.times.array
         (pi (4) (g1284))
         (pi (4) (g1284))
        )
       )
      )
     end
     (reduce.array (real.plus) (0.0)
      (real.times.array
       (pi (2) (g1284))
       (pi (2) (g1284))
      )
     )
    end
```

Figure 6.15: The iteration function of CG after Common Sub-expression Elimination

- `g1292` is a real value, the quotient of the inner-product involving the previous binding and the first inner-product. This binding corresponds to that for `beta` in the SML specification.

6.2.6 DAP

Finally, the optimized, functional Array Form is converted into DAP specific form, expressed in imperative Fortran Plus Enhanced.

SML $\longrightarrow$ DAP
$\equiv$ SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ Array Form $\longrightarrow$ Common Sub-expression Elimination $\longrightarrow$ FPE.

The Fortran Plus Enhanced implementation of CG is shown in figure 6.16; the SML specification is also shown.

- The set of vectors manipulated by the algorithm is realized as four arrays `x(*n)`, etc. The `*`s in the declarations indicate that these arrays are to be processed by the processor grid. The computation of the vectors for the next approximation is performed as destructive updates on these arrays, so there are no separate variables corresponding to `x′`, etc.
- The iteration required by the algorithm (expressed using `iterate`) is implemented as a `GOTO` loop beginning at the line with label 1 and ending at the line with label 15.

```
type cgstate = reeal vector*real vector*...;
fun cgiters(a:real matrix,b:real vector):cgstate
 = let
  (* Terminating condition.*)
  fun isok((x,r,p,q):cgstate):bool
   = innerproduct(r,r)<epsilon;

  (* One iteration.*)
  fun cgiter((x,r,p,q):cgstate):cgstate
   = let
    val rr:real = innerproduct(r, r);
    val alpha:real = rr/innerproduct(q, q);
    val x':real vector = x+p*alpha;
    val atq:real vector = transpose(a)*q;
    val r':real vector = r-atq*alpha;
    val beta:real = innerproduct(r',r')/rr;
    val p':real vector = r'+p*beta;
    val q':real vector = a*r'+q*beta
   in
    cgstate(x',r',p',q')
   end
 in
  iterate(cgiter,cgstate(x0,r0,p0,q0),isok)
 end
```

$\longrightarrow$

```
   real*8 A(*n,*n)
   real*8 x(*n)
   real*8 r(*n)
   real*8 p(*n)
   real*8 q(*n)
   integer cnt
   real*8 b(*n)
   real*8 r1(*n)
   real*8 beta
   real*8 rr

   :
   :
1  continue
2  rr = sum(r*r)

3  if (sqrt(rr).lt. 1.0E-14) then
4  goto 15
5  else
6  alpha = rr/sum(q*q)
7  r1 = r-sumr(A*matc(q,n))*alpha
8  beta = sum(r1*r1)/rr
9  q = sumc(A*matr(r1,n))+q*beta
10 x = x+p*b
11 r = r1
12 p = r1+p*beta
13 goto 1

14 endif
15 continue
   :
   :
```

Figure 6.16: DAP Fortran implementation of Conjugate Gradient

- Line 2 computes the inner-product of `r` with itself. This value is the measure of the accuracy of the approximation.
- If the approximation is sufficiently accurate, the loop is exited by the `GOTO` statement on line 4.
- Otherwise, the next set of values (x′, r′, p′ and q′) is computed by lines 6 to 12
- Note in particular that lines 7 and 9 compute the two matrix-vector products:

  $$\begin{array}{lcl} \texttt{transpose(A)*q} & \longrightarrow & \texttt{sumr(A*matc(q,n))} \\ \texttt{A*r}' & \longrightarrow & \texttt{sumc(A*matr(r1,n))}\ . \end{array}$$

  In the first product, the matrix A is transposed, but no explicit transpose operation occurs in the implementation; rather, row and column operations in the implementation of normal matrix-vector multiplication (i.e. without the transposition) are interchanged. This is the cause of the slight difference in form between the implementations of the two products.

Examination of the implementation shows it to be quite efficient: it certainly makes effective use of the DAP hardware, with all vector and matrix operations being performed in fully data-parallel manner. The only unsatisfactory aspect of the implementation is the unnecessary use of the variable `r1`: the assignment to `r1` in line 30 could be replaced with an assignment to `r`, obviating the need to assign `r` later. Efficiency could be improved by eliminating two vector assignments and one vector variable. The improvement would not be major (as vector assignments are very cheap on the DAP), but is desirable nonetheless; transformations do exists for optimizing the reuse of storage, but that topic is beyond the scope of this thesis.

Some of the operations in the SML specification (such as vector addition and scaling) are almost identical to their counterparts in the FPE implementation; such operations would have been easy to implement manually. What are more interesting are the matrix-vector products: the FPE counterparts of these are somewhat esoteric. Even so, the implementation of CG on the DAP would not be considered difficult.

That does not mean that all the effort in constructing specifications and deriving the FPE implementation has been in vain. The same derivation can be applied to the POT algorithm (discussed in the next section), the implementation of which for the DAP is *not* simple. Further, the next chapter considers minor modifications to the derivation that permit sequential and CRAY implementations to be produced. Then, the the derivation is extended to tailor implementations to use sparse data types, for sequential, CRAY and DAP systems. The sparse implementations are definitely not straightforward.

6.3 POT

To illustrate further the effectiveness of the DAP derivation, I here discuss the implementation of the POT algorithm. For brevity, I present only the specification and the Fortran Plus Enhanced implementation (omitting the intermediate forms): the specification and implementation are generally the only forms that would be of interest to a programmer using a derivation, and the implementation would be of interest only in the sense that it is to be presented to the compilation system; it is not to be studied. I also compare the execution performance of the derived implementation with that of a hand-crafted implementation.

[97]

The POT algorithm for the computation of the eigensystem of a real symmetric matrix A, of order N, consists essentially of the following iterative cycle. Given an $N * N$ matrix R of orthonormal approximations to the eigenvectors of A:

1. Form a transformed matrix B, of order N, using the sequence of algebraic operations:

$$V := A * R, B := R^T * V$$

2. Construct a transformation matrix T, of order N, whose columns form an approximation to the set of right eigenvectors of the symmetric matrix B. This construction is described in [8].
3. Compute a new set of eigenvector approximations R, which is represented in matrix form by the algebraic matrix product

$$R := V * T.$$

4. Orthonormalise the set of vectors R.

The cycle ceases whenever the off-diagonal components of B are zero to a predetermined precision. The diagonal of B then yields the required eigenvalues and the columns of the final matrix R yield the required eigenvectors.

Figure 6.17: Mathematical definition of POT

The POT algorithm, figure 6.17 is used to calculate eigenvalues and eigenvectors of real, symmetric matrices. An eigenvector/eigenvalue pair of a matrix A satisfy the *characteristic equation* $Ar = \lambda r$ where r is the eigenvector and λ the corresponding eigenvalue. In general, if A is of order N, there are N such pairs, so the characteristic equation for the full eigensystem (the collective term for the eigenvalues and eigenvectors) can be written

$$AR = R\Lambda$$

where R is a matrix of order N and Λ is a diagonal matrix also of order N; the columns of R are the eigenvectors of A and the diagonal elements of Λ are the corresponding eigenvalues.

The mathematical definition of the POT algorithm is repeated in figure 6.17 for ease of reference. The transformation matrix used in step (2) is defined in figure 6.18. The orthonormalization referred to in step (4) is discussed in appendix C.

[98]

The function $transform$ generates from its real symmetric argument, B_k, a non-singular orthogonal transformation matrix T_k, whose elements are given as follows:

$$\begin{aligned} t(i,j) &= 2b(i,j)/[d(i,j) + sign(d(i,j))\sqrt{(d(i,j)^2 + 4b(i,j)^2)}] && i > j \\ t(i,i) &= 1 \\ t(j,i) &= -t(i,j) && i < j \end{aligned}$$

where

$$d(i,j) = b(i,j) - b(i,i)$$

and where, for convenience, the subscript k has been ignored.

Figure 6.18: Mathematical definition of the transformation matrix

6.3.1 SML Specification of POT

Figure 6.19 shows the main functions of the SML specification of POT.

- The function `POT` computes (an approximation to) the eigensystem:
  - The matrix whose eigensystem is to be computed is argument `A`.
  - The initial approximation is the unit matrix of the same order as `A`.
- The computation of the eigensystem is an iterative process, so `POT` uses the library function `iterate`:
  - Each iteration applies the function `POTstep` to `A` and to the current approximation of the eigenvectors.
  - The current approximation to the eigenvectors is argument `R`.
  - The iteration terminates when the function `IsOk` applied to the current approximation returns true.
- Each iteration forms an improved approximation to the eigenvectors by forming a (triple) matrix product with a transformed matrix (the application of `transform`[1]), and then orthonormalizing (using the function `orthonormalize`, the definition of which is not included in this figure).
- The functions `transform` and `calculate` compute the transformation matrix, as defined in figure 6.18).
- The function `IsOk` computes the mean of the squares of the off-diagonal elements of the current eigenvalue approximation and compares the mean with a constant (`TOL`) which determines the required accuracy of the solution.

[1]The function `transform` defined here is unrelated to TAMPR transformations!

```
fun POT(A:real matrix):(real matrix * real matrix)
  = iterate(POTstep, (A, UnitMatrix(shape(A))), IsOk)

fun POTstep(A:real matrix, R:real matrix)
    :(real matrix * real matrix)
  = let
      val V:real matrix = A*R
      val B:real matrix = transpose(R)*V
      val T:real matrix = transform(B)
      val R':real matrix = V*T
    in
      (A, orthonormalize(R'))
    end

fun transform(B:real matrix):real matrix
  = let
      fun calculate(i:int,j:int):real
        = let
            val d:real = B@[j,j]-B@[i,i]
          in
            2*B@[i,j]/(d+sign(d)*sqrt(sqr(d)+4*sqr(B@[i,j])))
          end
      in
        generate(shape(B),
          fn(i:int,j:int) =>
            if (i>j) then calculate(i,j)
            else if (i=j) then 1.0
            else ~calculate(j,i))
      end

fun IsOk(A:real matrix,R:real matrix):bool
 = let
      val V:real matrix = A*R
      val B:real matrix = transpose(R)*V
      val diag:real vector = diagonal(B)
      val N = size(A,0)
     in
       (sum(B*B)-sum(diag*diag))/(N*(N-1))<TOL
     end
```

Figure 6.19: Functional specification of POT

The SML specification of POT is a simple rewriting of the mathematical definition to conform with the specification language.

6.3.2 DAP Form of POT

The main section of the Fortran Plus Enhanced implementation of POT is shown in figure 6.20.

- The iteration of `POTiters` has been implemented as a `GOTO` loop beginning at line 100 and ending at line 200.
- The loop terminated at line 110 computes the product of matrices A and R. This product is stored as it is used twice: in the computation of B ($R^T * A * R$) and the eigenvector matrix approximation R ($A * R * transform(B)$). (The storing of the value was arranged by the Common Sub-expression Elimination sub-derivation.)
- The computation of B (the current approximation to the eigenvalues) is completed at line 125.
- If B is sufficiently close to being diagonal (the root-mean-square of the off-diagonal elements is sufficiently close to zero), the loop is exited by the `GOTO 200` statement.
- The following lines, up to line 126, form the transformation matrix. The definition of $transform$ explicitly distinguishes elements in the lower triangle from elements in the upper triangle; its implementation on a SIMD architecture thus requires the computation of two matrices (one for lower, one for upper triangle) which are ‘merged’. However, because the

```
      integer n
      real*8 tol
      parameter(tol=1e-15)
      parameter(n=??)

      real*8 m(*n,*n)
      real*8 A(*n,*n)
      real*8 R(*n,*n)
      real*8 tranR(*n,*n)
      real*8 AR(*n,*n)
      integer step
      real*8 B(*n,*n)
      real*8 g611(*n,*n)
      real*8 diagB(*n)
      real*8 g598(*n)
      logical mask(*n,*n)
      logical g650(*n,*n)
      real*8 D(*n,*n)
      integer signD(*n,*n)
      logical g595(*n,*n)

      :
      :
100   continue
C        AR := A*R
      tranR=tran(R)
      AR=0
      do 110 step=1,n
      AR=AR+matr(A(,step),n)*
    .   matc(tranR(,step),n)+
110   continue

C        Compute B
      B=AR
      g611=0
      do 120 step=1,n
      g611=g611+matc(tranS(,step),n)*
    .   matr(b(,step),n)
120   continue
125   B=g611
```

```
C        Test for convergence
      diagB=B(patunitdiag(n),)
      g598=diagB
      g598=g958*g958
      if ((sum(B*B)-sum(g598))/
    .   (n*(n-1)).lt.tol) goto 200

C        R := Transform(B)
      g595=patlowertri(n)
    .   .and. .not. patunitdiag(n)
      mask=patlowertri(n)
      g650=patunitdiag(n)
      R(mask.and.g650)=1
      mask(g650)=.false.
      g650=g595
      D=matr(diagB,n)-matc(diagB,n)
      signD=1
      signD(D.lt.0)=-1
      R(mask.and.g650)=(-2*B)/
    .   (D+signD*sqrt(D*D+4*(B*B)))
126   R(.not.patlowertri(n))=-tran(S)

C        update R
      D=0
      do 130 step=1,n
      D=D+matc(AR(step,),n)*
    .   matr(S(,step),n)+
130   continue
      S=R
      :
      :
      goto 100
200   continue
```

Figure 6.20: Fortran Plus Enhanced implementation of POT
N.B.: the comments were inserted manually

transformation matrix is anti-symmetric, only one of these expressions need be computed; the other expression can be formed by transposing and negating (as in line 126).

Some of the manipulation of masks in this part of the computation is unnecessary: no effort has been made to optimize mask expressions as they are so cheap on the DAP. (The grid of single-bit processing elements can manipulate the single-bit representation used for booleans very efficiently.)

- The eigenvector approximation matrix R is then updated by the loop terminating on line 130. The orthonormalization of R is not shown. (It is discussed in appendix C.)

The FPE implementation of POT is considerably different from its SML specification: the computation of the matrix-matrix products, and of the transformation matrix would seem quite strange to one not familiar with the DAP.

6.3.3 Evaluation of the POT Implementation

The derived implementation of POT can be evaluated by comparing its performance with that of an independently developed (hand-crafted) implementation.

In figure 6.21 the execution time (for the computation of one approximation[2]) for the implementation of POT derived by automatic program transformation is shown together with times for two hand-crafted versions - the first is written in Fortran-Plus and the other written in Fortran Plus Enhanced. These hand-crafted versions have been analysed in [97]. A hand-crafted Fortran-Plus version of POT is between 12% and 13% faster than a hand-crafted Fortran-Plus Enhanced version. As reported in [26] the code produced by the Fortran-Plus Enhanced compiler for frequently occurring linear algebra operations (e.g. matrix product) is very efficient but it is less so on more specialized operations (e.g. Gram-Schmidt orthogonalization, discussed in appendix C).

The hand-crafted and automatically derived Fortran Plus Enhanced versions have execution times that are almost identical. For the large matrix examples the derived implementation is marginally slower than the time for the hand crafted version (between 0.1% and 0.6%). This discrepancy arises from a minor optimization made possible by the particular way the hand-crafted version produces the transformed matrix. It would be possible to write a transformation that could perform this specific optimization, but it would probably be too special-purpose to be of general use.

Matrix Size	*Time per iteration (sec)*		
	Hand Crafted Fortran Plus	Hand Crafted Fortran Plus Enhanced	Automatically Derived Fortran Plus Enhanced
64	1.2	1.35	1.35
128	8.23	9.30	9.31
256	60.92	69.86	70.30

Figure 6.21: Hand-crafted POT versus derived POT

The virtual parity in execution times of the derived and hand-crafted versions of POT is evidence that the DAP derivation is indeed effective even for quite complex algorithms. It should be noted that, while the section of the implementation shown in figure 6.20 is quite complex, the implementation of orthogonalization (discussed in the next section) is rather more intricate. As orthogonalization is equally as important to the total execution time (orthogonalization is responsible for about 50% of the execution time) it can be concluded that the DAP derivation is just as effective at implementing orthogonalization.

6.4 Summary

In the previous chapter, I illustrated the derivation of an implementation of a simple algorithm (matrix-vector multiplication). Since the derivation process is entirely automatic, it should — in theory — be a simple matter to apply the derivation to more complex algorithms, and obtain implementations. In practice, there are several reasons why derivations *may* fail to produce efficient implementations of algorithms more complex than MVM (though the derivations presented in this work do *not* fail):

- The transformations or the structure of the derivation may have been too tightly tailored for a particular algorithm; a more complex algorithm may introduce operations that did not arise in the simple example.

[2]The time to compute each approximation is constant.

- Transformations may recognize particular, rather complex operations such as matrix-vector or matrix-matrix multiplication, and optimize the implementation of these operations. In more complex algorithms, such operations may be combined with other operations (for example, in the conjugate gradient algorithm, matrix-vector multiplication is combined with matrix transposition and vector scaling and addition); the expressions resulting from such combinations may not be recognized by the transformations.[3]

  That is to say, transformations based upon special cases are usually more restrictive than transformations based upon general rules.

The purpose of this chapter was to illustrate that the derivations discussed in this work are *not* limited to simple operations — that they are general purpose.

The problems discussed above are avoided in the derivations presented in this thesis by concentrating on the development of sequences of programming models (intermediate forms), which by their nature, are independent of particular algorithms; and by designing sub-derivations to convert from one model to the next using general identities based on the formal properties of the models.[4]

The production of DAP implementations from SML specifications of Conjugate Gradient and POT illustrates several aspects of derivations:

- A sequence of intermediate forms has been identified that provides a graceful transition from SML specification to efficient Fortran Plus Enhanced implementation for the DAT array processor.
- Intermediate forms of non-trivial algorithms can be quite complex; too complex for a person to manually refine.
- FPE implementations can often be quite arcane, but that is not a major concern when implementations are automatically derived.
- The virtual parity in execution performance of the the derived implementation of POT with an independent, hand-crafted implementation supports one of the underlying tenets of this thesis: that efficient implementations can be automatically derived from simple, abstract specifications.

[3]Such limitations seem to bedevil advanced compilers: for example, the CRAY compiler will recognize matrix-vector multiplication and convert it into an application of a highly optimized routine coded directly in machine language. But it will recognize matrix-vector multiplication only in certain forms. Thus, the effectiveness of the conversion performed by a compiler is limited to those forms the compiler developer anticipated and thought worth the effort of encoding.

[4]Of course, there are occasions when it is expedient to use a transformation that is designed to optimize a particular algorithm, but generally it is preferable if the principles underlying the transformation are developed into more generally applicable transformations (that, say, optimize a class of algorithms, rather than a single algorithm).

Chapter 7

Extending the Derivation for Other Architectures

In the discussion on functional specifications, I emphasized that one motive for constructing a specification independent of any particular computational model was to facilitate implementing the specification of a range of models. In this chapter, I modify the SML-to-DAP derivation to produce implementations for sequential and vector computational models, using Conjugate Gradient as an example.

The discussion is as follows:

- Modifying the structure of the derivation for sequential implementations, with Fortran77 as the implementation language.
- Those stages in the derivation specific to a sequential implementation.
- A brief description of the CRAY X-MP vector processor.
- Modifying the structure of the derivation for a vector architecture, the CRAY X-MP.
- The CRAY implementation of CG.
- An evaluation of the CRAY implementation.

7.1 Sequential Implementations

In this section, I discuss modifying the SML-to-DAP derivation into an SML-to-Fortran77 derivation.

The derivation to produce DAP implementations is:

$$\boxed{\begin{array}{l} \text{SML} \longrightarrow \text{DAP} \\ \quad \equiv \text{SML} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \longrightarrow \text{Static Evaluation} \\ \qquad \longrightarrow \text{Array Form} \longrightarrow \text{Common Sub-expressions} \longrightarrow \text{FPE.} \end{array}}$$

That is, the initial SML form is converted into a standard functional form with simple syntax, and then optimized by having definitions unfolded and expressions simplified. A specification is then tailored for the implementation architecture by recasting it into an explicitly data-parallel form. The data-parallel form is optimized by ensuring expensive computations are not repeated, and then finally converted into the chosen implementation language (Fortran Plus Enhanced).

Of the stages of the derivation, only the Array and FPE stages are directly dependent upon the DAP being the implementation architecture: converting to λ-calculus, unfolding, evaluating and eliminating common sub-expressions can, and probably should, be applied regardless of the implementation architecture.

Thus, the initial stages of an SML-to-Fortran77 derivation are already available, to convert an SML specification into the λ-calculus (and to simplify that form). The final stage of the derivation, to convert λ-calculus into Fortran77, is also already available: it was discussed in section 4.5.1 in the context of the Lisp-to-Fortran77 derivation. Thus, the SML-to-Fortran77 derivation is as follows:

$$\boxed{\begin{array}{l} \text{SML} \longrightarrow \text{Fortran77} \\ \quad \equiv \text{SML} \longrightarrow \lambda\text{-Calculus} \longrightarrow \text{Unfolding} \longrightarrow \text{Static Evaluation} \\ \qquad \longrightarrow \text{Common Sub-expressions} \longrightarrow \text{Fortran77.} \end{array}}$$

That is, after converting to the simple Statically Evaluated form, a specification is restructured to ensure expensive computations are not repeated, and then converted into Fortran77.

To illustrate this derivation, I shall discuss the implementation of Conjugate Gradient. The derivation from SML specification to Statically Evaluated form was discussed in the previous chapter, so here I discuss only the application of the Common Sub-expression Elimination and Fortran77 sub-derivations.

7.1.1 CG after Common Sub-expression Elimination

The previous chapter discussed the initial stages of the derivation — conversion to λ-calculus form, unfolding and Static Evaluation; the Statically Evaluated form of CG is shown in figure 6.5.

After Static Evaluation, the Common Sub-expression sub-derivation is applied, to produce the expression in figure 7.1.

The important aspects of the figure are the λ-abstractions at the beginning; uses of the names introduced by the bindings are underlined in the figure.

- `g936` is the inner-product of the second component of `g898` (the 4-tuple of real vectors that is the argument of the iteration function) with itself: this binding corresponds to the binding for `rr` in the SML specification.

```
λg898·
 λg936:real·
 λg935:real·
  λg960:real array·
   λg932:real·
    tuple
     (generate ([n])
      (λ[g868]·
       real.plus
        (element (pi (1) (g898)) ([g868]))
        (real.times
         (element (pi (3) (g898)) ([g868]))
         (g935)
        )
       end
      ):real ma
     )
     (g960)
     (generate ([n])
      (λ[g884]·
       real.plus
        (element (g960) ([g884]))
        (real.times
         (element (pi (3) (g898)) ([g884]))
         (g932)
        )
       end
      ):real ma
     )
     (generate ([n])
      (λ[g897]·
       real.plus
        (reduce ([n])
         (λ[g888:int]·
          real.times
           (element (g906) ([g897,g888]))
           (element (g960) ([g888]))
          end
         )
         (real.plus) (0.0)
        )
        (real.times
         (element (pi (4) (g898)) ([g897]))
         (g932)
        )
       end
      ):real ma
     )
    end
```

```
      (real.quotient
       (reduce ([n])
        (λ[g882]·
         real.times
          (element (g960) ([g882]))
          (element (g960) ([g882]))
         end
        )
        (real.plus) (0.0:real)
       )
       (g936)
      )
    end
     (generate ([n])
      (λ[g958]·
       real.difference
        (element (pi (2) (g898)) ([g958]))
        (real.times
         (reduce ([n])
          (λ[g885]·
           real.times
            (element (g906) ([g885,g958]))
            (element (pi (4) (g898)) ([g885]))
           end
          )
          (real.plus) (0.0)
         )
         (g935))
       end
      ):real ma
     )
   end
    (real.quotient
     (g936:real)
     (reduce ([n])
      (λ[g891]·
       real.times
        (element (pi (4) (g898)) ([g891]))
        (element (pi (4) (g898)) ([g891]))
       end
      )
      (real.plus) (0.0)
     )
    )
   end
    (reduce ([n])
     (λ[g890]·
      real.times
       (element (pi (2) (g898)) ([g890]))
       (element (pi (2) (g898)) ([g890]))
      end
     )
     (real.plus) (0.0)
    )
  end
```

Figure 7.1: The iteration function of CG after Common Sub-expression Elimination

- `g935` is the quotient of the previous binding with the inner-product of the fourth component of `g898` with itself; this binding corresponds to the binding for `alpha` in the SML specification.

- `g960` is a real vector; this binding corresponds to the binding for `r′`.

- `g932` is the quotient of the inner-product of the vector in the previous binding, with itself; this binding corresponds to that for `beta`.

The form produced by Common Sub-expression Elimination is computationally efficient: expensive computations are performed once only.

7.1.2 Fortran77 Implementation of CG

The Fortran77 implementation of Conjugate Gradient is shown in figure 7.2.

```
   real A(n,n)
   real b(n)
   real x(n)
   real r(n)
   real p(n)
   real q(n)
   integer j
   integer i
   real r1(n)
   real rr
   real g940
   real beta
   real alpha

   :
   :
1  continue
2  rr = 0.0
3  do 20 i = 1, n, 1
4  rr = rr+r(i)*r(i)
5  continue

6  if (sqrt(rr).lt.1E-14) then
7  goto 44
8  else

9  alpha = 0.0
10 do 12 j = 1, n, 1
11 alpha = alpha+q(j)*q(j)
12 continue
13 alpha = rr/alpha

14 do 20 i = 1, n, 1
15 r1(i)= 0.0
16 do 18 j = 1, n, 1
17 beta = beta+A(j,i)*q(j)
18 continue
19 r1(i)= r(i)-beta*alpha
20 continue
21 beta = 0.0
22 do 24 i = 1, n, 1
23 beta = beta+r1(i)*r1(i)
24 continue
25 beta = beta/rr

26 do 28 i = 1, n, 1
27 x(i)= x(i)+p(i)*alpha
28 continue

29 do 31 i = 1, n, 1
30 p(i)= r1(i)+p(i)*beta
31 continue

32 do 34 i=1,n
33 r(i) = r1(i)
34 continue

35 do 41 i = 1, n, 1
36 g940 = 0.0
37 do 39 j = 1, n, 1
38 g940 = g940+A(i,j)*r1(j)
39 continue
40 q(i) = g940+q(i)*beta
41 continue

42 goto 1
43 endif

44 continue
   :
   :
```

Figure 7.2: Fortran77 implementation of CG

The implementation operates as follows:

- 1 is the start of the loop that implements the iteration required by CG; line 44 is the end of the loop.
- 2-5 compute `rr`, the inner product of `r` with itself; this value is the measure of accuracy of the current approximation.
- 6,7 test if the approximation is sufficiently accurate, and if so, exit the loop.
- 9-13 compute the value of `alpha`, the quotient of the inner-product of `q` with itself and rr.
- 14-20 compute the next value for `r`, stored in `r1`:
  - 15-18 compute one element of the product of the transpose of `A` and `q` (using the variable `beta` to temporarily store the value).
  - 19 subtracts that value (scaled by alpha) from `r`.
- 21-25 compute `beta`, the quotient of the self inner-product of the new value for `r` (which is stored in `r1`) and `rr`.
- 26-28 subtract `p` scaled by `beta` from `x` to give the new value for `x`.
- 29-31 add `p` scaled by `beta` to `r1` to give the new value for `p`.

- 32-34 copy `r1` into `r`.

- 35-41 compute the new value for `q`:

  - 36-39 compute one value of the product of `A` and the new value of `r`, storing the value in `g940`.

  - 40 adds that value to `q` (scaled by `beta`) to give one element of the new value for `q`.

- 42 returns to the start of the loop.

In the work discussed in this thesis, I am primarily concerned with implementations for array and vector architectures, so I shall not discuss the sequential implementation in detail; a few points should be noted though:

- The most important aspect of the implementation is that updating of the array variables representing the four vectors x, r, p and q manipulated by the algorithm is performed with only one additional array variable `r1`: the in-place updating of `x`, `p` and `q` gives better performance than storing the updated values in additional variables and then copying the value to `x`, `p` and `q`. One failing of the implementation is that `r`$'$ is not updated in place, but discussion of why `r1` was introduced is beyond the scope of this thesis.

- The computations of `r`$'$ and `q`$'$ (beginning at lines 14 and 34) are extremely compact: each computation merges a matrix-vector product with the scaling and addition of a vector. (Such merging is produced by the Unfolding and Static Evaluation stages of the derivation.) Theoretically, these compact forms should be efficient: each avoids the construction of a full vector for the matrix-vector product, forming each element of the product only as it is needed (the element's value being stored in `beta` or `g940`); the merging of the computation of the product with the scaling and addition of the vector reduces the number of loops required, and so should reduce overheads associated with managing the loops.

  However, the efficiency of a Fortran77 program is always difficult to judge *a priori*, due to the manipulation performed by most compilers. Thus, if I intended make extensive use of the sequential implementation, I would probably introduce further stages in the derivation to tailor the implementation to the appropriate compiler.

- The loops beginning at lines 26 and 29 could be merged, further reducing overheads for loops.

7.1.3 Summary of Sequential Sub-derivation

The modification of the SML-to-DAP derivation into an SML-to-Fortran77 derivation is intended as:

- an illustration of the principles that functional specifications can be constructed in a manner that facilitates implementation on more than one architecture, and that most of the stages of a derivation are independent of the implementation architecture (and so can be shared by several derivations);

- a reference point for the CRAY implementation (discussed in the next section), which also is expressed in Fortran77.

The sequential implementation produced is not intended to be highly efficient: it is certainly quite efficient, but would require further tailoring to individual implementation systems (particularly to the Fotran77 compilers for those systems) if it is to be used as a practical implementation.

7.2 CRAY Implementation of CG

In this section, I consider a derivation to produce implementations suited to CRAY vector processors from SML specifications. I assume a general familiarity with the CRAY computer systems, so I provide only a very basic overview.

7.2.1 The Target Environment

The computer system used for this section is a CRAY X-MP. The particular X-MP used has three processors, each of which has multiple vector units (which provide efficient manipulation of vectors). However, only one of the three processors was used for the work reported here: that is, this work used the X-MP as a vector processor, rather than as a multi-processor system. In addition, the CRAY compiler provides the option to disable vector operations, so the X-MP can be used as an advanced sequential processor.

The language used to program the X-MP is Fortran77. The compilation system is responsible for *vectorizing* programs, i.e. for interpreting the inherently sequential instruction sequence of an F77 program in such a way as to make effective use of the vector units. Vectorization is primarily based upon `DO` loops whose bodies are such that the iterations can be performed simultaneously (though a vector processor pipelines operations rather than performing them simultaneously). For example, the n assignments in the loop

```
  DO 1 i=1,n
  A(i) = B(i)+C(i)
1 CONTINUE
```

where `A`, `B` and `C` are arrays, can be pipelined so that the entire loop executes in approximately n clock cycles, rather than the $n.t$ clock cycles required by sequential execution (where t is the number of cycles required for one addition and assignment).

Other classes of `DO` loops can also be executed efficiently, using special features of the processor units; for example, the class of *reduction loops*, which perform reductions:

```
  s = 0.0
  DO 1 i=1,n
  s = s+A(i)
1 CONTINUE
```

The CRAY compiler also recognizes certain forms of loops as applications of particular operations for which it has efficient routines, hand-crafted in assembly language. For example, the loop

```
  DO 1 i=1,n
  V(i) = 0.0
  DO 1 j=1,n
  V(i) = V(i)+A(i,j)*U(j)
1 CONTINUE
```

is recognized as matrix-vector multiplication, and is replaced by an invocation of a CRAY-supplied routine, called `SGEMVX` ("`S`tandard `GE`neral `M`atrix-`V`ector", and "`X`" for multiplication).

The vectorizing compilers of major vendors such as CRAY are quite effective at vectorizing loops that commonly occur in programs used by numerical, scientific and engineering programs. However, even relatively simple changes to a program can be sufficient to inhibit vectorization or other optimizations. For example: if the computations of the product of a matrix `A` and a vector `U` is to be composed with the addition of a vector `W`, it could be expressed as

```
  DO 2 i=1,n
  V(i) = W(i)
  DO 1 j=1,n
  V(i) = V(i)+A(i,j)*U(j)
1 CONTINUE
2 CONTINUE
```

or as

```
  DO 2 i=1,n
  V(i) = 0.0
  DO 1 j=1,n
  V(i) = V(i)+A(i,j)*U(j)
1 CONTINUE
  V(i) = V(i)+W(i)
2 CONTINUE
```

.

The CRAY compiler (version 5) converts the form on the left into a combination of vector addition and `SGEMVX`, resulting in much quicker execution than for the form on the right, which it only vectorizes.

Thus, the precise form of loops can have a considerable impact upon the execution performance.

7.2.2 CRAY Derivation

Since the CRAY is programmed using Fortran77, the CRAY derivation is essentially the same as the sequential derivation, though the CRAY derivation has an additional stage that tailors the program for the CRAY:

SML $\longrightarrow$ CRAY F77
$\equiv$ SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ $\underline{\text{CRAY}}$ $\longrightarrow$ Common Sub-expressions $\longrightarrow$ Fortran77.

That is, a specification is converted into a simple functional form and optimized. It is then tailored for the CRAY, further optimized and finally converted into imperative form (expressed in Fortran77).

The important aspect of the derivation is the tailoring for the CRAY. Ideally, this tailoring should be based upon a well-defined programming model, similar in nature to that developed for the DAP array processor. However, the CRAY hardware is in many ways rather more complex than the DAP hardware: each vector processor has multiple vector units that can perform operations (such as memory fetch or store, addition and multiplication) independently; some of the vector units can perform reduction operations as well as elementwise operations; operations can be combined (or *tail-gated*) to eliminate memory stores and fetches by keeping data in vector registers.

But perhaps more important is that the programming language (Fortran77) provides such a confused interface to the hardware: the operations that can be performed by the vector units, and the ways in which these operations can be effectively combined, are *not* delineated by the programming model supported by Fortran77: Fortran77 is an abstraction of a sequential architecture, not a vector architecture.

In addition, what is important is not the programming model supported by the programming language, but the model supported by that language *as interpreted by the compiler*. The analysis and manipulation performed by the CRAY compiler to vectorize and optimize a program is necessarily quite complex, but inevitably incomplete: thus, it can be difficult to judge how the compiler has interpreted a program, and ill-advised to assume that the compiler has obtained the best possible performance from a program.

The situation with the CRAY compiler is very different from that with the DAP compiler. Fortran Plus Enhanced does provide an abstraction of the DAP hardware, but it is an abstraction that is well matched to the hardware. Consequently, the compiler need perform a much simpler analysis of a program to obtain effective use of the hardware. As a result, the programming model presented to the programmer is mostly a simplification of the underlying machine; the CRAY programming environment provides the programmer with a model that is almost entirely alien to the machine.

The consequences for the CRAY derivation of the above considerations are:

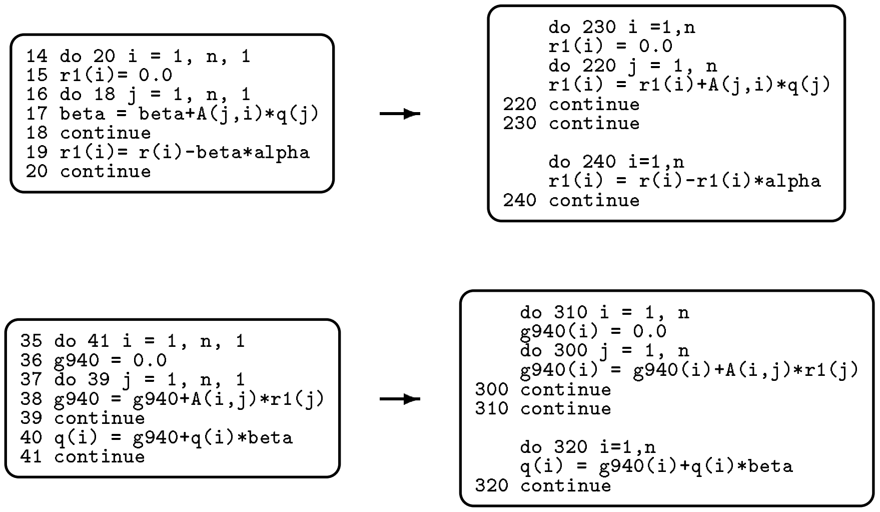
- The Statically Evaluated form produces generations and reductions that, expressed as `DO` loops in F77, are quite well suited to the compiler: the generating functions tend to be quite complex expressions which the compiler can analyze to extract work for the multiple vector units.
- However, the analysis performed by the compiler will sometimes fail if an expression is ‘too complex’ (for example, if it contains conditional expressions that are ‘too deeply nested’).
- In addition, some important optimizations performed by the compiler require loops to conform to quite rigid templates.

Thus, a straightforward translation of generations and reductions into loops provides reasonable efficiency, but considerable improvement can be obtained by tailoring loops to ensure that expressions are not ‘too complex’ and that certain operations appear in forms the compiler will recognize.

It is this tailoring to the CRAY compiler that is performed by the CRAY stage of the derivation. As might be expected by the above discussion, the tailoring is rather *ad hoc*: shortcomings of the compiler are identified and expressions recast to avoid the shortcomings; worthwhile optimizations performed by compiler are identified and expressions recast to become eligible for optimization.

I would prefer the CRAY stage to be more systematic, but that seems to be precluded by the programming environment. However, it seems that CRAY (and many other vendors) are adopting data-parallel versions of Fortran such as Fortran90 and High Performance Fortran as their primary programming language; it may be hoped that the adoption of such standards would facilitate the development of a useful programming model. Indeed, it is likely that such a model would be similar to the Array Form used for the DAP (since many of the data-parallel features of FPE are shared by both Fortran90 and High Performance Fortran): then a derivation for a CRAY system could be constructed from the DAP derivation by replacing the final Array-to-DAP sub-derivation with an Array-to-Fortran90 or Array-to-HPF sub-derivation.

7.2.3 CRAY Form of CG



```
14 do 20 i = 1, n, 1
15 r1(i)= 0.0
16 do 18 j = 1, n, 1
17 beta = beta+A(j,i)*q(j)
18 continue
19 r1(i)= r(i)-beta*alpha
20 continue
```

→

```
    do 230 i =1,n
    r1(i) = 0.0
    do 220 j = 1, n
    r1(i) = r1(i)+A(j,i)*q(j)
220 continue
230 continue

    do 240 i=1,n
    r1(i) = r(i)-r1(i)*alpha
240 continue
```

```
35 do 41 i = 1, n, 1
36 g940 = 0.0
37 do 39 j = 1, n, 1
38 g940 = g940+A(i,j)*r1(j)
39 continue
40 q(i) = g940+q(i)*beta
41 continue
```

→

```
    do 310 i = 1, n
    g940(i) = 0.0
    do 300 j = 1, n
    g940(i) = g940(i)+A(i,j)*r1(j)
300 continue
310 continue

    do 320 i=1,n
    q(i) = g940(i)+q(i)*beta
320 continue
```

Figure 7.3: Modified versions of matrix-vector products for CRAY implementation of CG

The CRAY form of CG is almost identical to the Sequential form (figure 7.2), with the exception that the matrix-vector products have been isolated by the CRAY stage of the derivation to allow them to be recognized by the compiler[1]: see figure 7.3.

[1] Alternatively, the transformations could introduce the CRAY routines directly.

7.2.4 Evaluation of CRAY Implementation

The CRAY implementation can be evaluated in two ways:

- comparison of the performance of the implementation with and without compiler vectorization enabled provides an indication of the suitability of the implementation for vectorization by the compiler;
- comparison of performance of two implementations, one derived using the CRAY stage, the other without the CRAY stage, provides an indication of the importance of that stage.

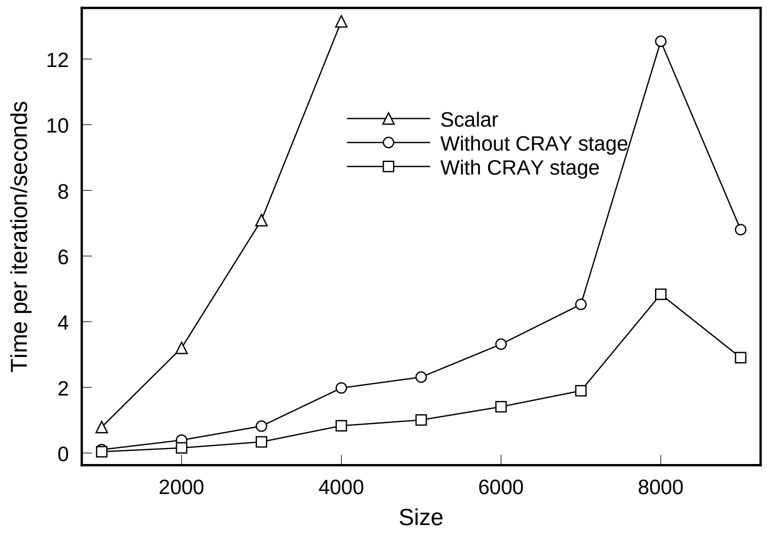


Figure 7.4: Comparison of execution times on CRAY X-MP of versions of CG

Figure 7.4 illustrates the execution performance of CG under the above listed conditions:

scalar: compiler vectorization disabled — the CRAY stage of the derivation makes little difference to the execution performance in scalar mode;

with: compiler vectorization enabled for an implementation produced using the CRAY stage;

without: compiler vectorization enabled for an implementation produced *not* using the CRAY stage.

For the vector versions, 9000 was the largest size of matrix that could be stored in memory; I have no idea what causes the huge increase in execution time at size 8000. For the scalar version, 4000 was the largest size of matrix that could be processed within the CPU time limit imposed on a single program; larger matrices could have been processed by submitting the program to a batch queue, but the timings shown are sufficient to indicate that the scalar version is much slower than the vector versions.

The CRAY documentation suggests that programs well suited to vectorization should exhibit speedups in the range 8 to 10. The speedup of the *with* version compared with the *scalar* version is about 8, so it can be concluded that the implementation is suitable for vectorization.

The *without* version is 2 to 3 times slower than the *with* version, indicating the importance of the CRAY stage of the derivation.

7.2.5 Summary of CRAY Sub-derivation

The lack of an elegant programming model for the CRAY precludes the development of a sub-derivation to perform comprehensive tailoring of a program for the CRAY. However, a sub-derivation can assist the compiler in vectorizing and optimizing a program, with considerable benefits to the execution performance.

7.3 Summary

Developing functional specification in a machine-independent manner facilitates the derivation of implementations for multiple architectures.

The structuring of derivations into sub-derivations and corresponding intermediate forms allows much of the SML-to-DAP derivation to be used unchanged in both an sequential and a CRAY derivation. Additional sub-derivations are introduced to convert a specification in functional form into Fortran77, and to tailor an implementation for execution by the CRAY. The CRAY tailoring is subtle, but can have considerable effect on execution performance.

Chapter 8

Sparse Matrices

In previous chapters, I have discussed derivations to produce implementations of SML specifications on sequential, array and vector computational architectures. In this chapter, I discuss an additional sub-derivation that optimizes implementations to make effective use of compressed storage schemes for matrices that are known to be sparse (i.e. to have mostly elements that are zero).

This Sparse sub-derivation can be combined with each of the derivations discussed earlier:

- information is provided in a specification to indicate the nature of the sparsity (e.g. that the non-zeros occupy a certain region of a matrix);
- the specification is converted into λ-calculus form and simplified;
- the Sparse sub-derivation is applied to optimize the specification;
- then the sequential, array or vector sub-derivation is applied to produce an optimized implementation for the appropriate architecture.

The discussion is as follows:

- A discussion of sparsity.
- The introduction of a sub-derivation for a particular form of sparsity: *tridiagonal*.
- The details of the tridiagonal sub-derivation.
- An evaluation of the implementations produced for sequential, array and vector architectures.
- An evaluation of the implementations produced for sequential and vector architectures for a second form of sparsity: *row-wise*.

8.1 Sparsity

In this section, I discuss sparsity in general and explain some of the terms that will be used throughout this chapter.

In this thesis, the only sparse data structure that I consider is a sparse matrix of real elements. A sparse matrix is thus a matrix in which most of the elements are zero. It is not necessary to define 'most' as a precise proportion of the elements: the use of sparse programming techniques is motivated by pragmatism, so 'most' means a sufficient number that it is worthwhile using special programming techniques to reduce storage requirements and/or execution time.

Sparse matrices arise in many 'real life' problems (particularly in engineering and modelling) where often only a few per cent of the elements of the matrices are non-zero. The use of sparse programming techniques may be the only way to obtain practical implementations of algorithms manipulating such matrices, as standard programming techniques usually take too long to execute or require more storage than is feasible to provide.

There are many types of sparsity, characterized by the locations of the non-zero elements. The types can be divided into two groups: structured and unstructured. In structured sparsity, the non-zero elements occur in patterns in the matrix: their locations can be specified parametrically. In unstructured sparsity, the non-zeros do not occur in patterns: their locations must be individually listed.[1]

Figure 8.1 shows an example of a structured and an unstructured matrix. The structured matrix is a *tridiagonal* matrix: the non-zeros occur only on the leading diagonal, one sub-diagonal and one super-diagonal (that is, the region for which the row index differs from the column index by not more than one).

$$\begin{bmatrix} (1,1) & (1,2) & 0 & 0 & 0 \\ (2,1) & (2,2) & (2,3) & 0 & 0 \\ 0 & (3,2) & (3,3) & (3,4) & 0 \\ 0 & 0 & (4,3) & (4,4) & (4,5) \\ 0 & 0 & 0 & (5,4) & (5,5) \end{bmatrix}$$

(a) A tridiagonal matrix

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 2 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \end{bmatrix}$$

(b) An unstructured matrix

Figure 8.1: Sparse matrices

8.1.1 Storing Sparse Matrices

The storage requirements of a program that manipulates a sparse matrix can be reduced by storing only the non-zero values in some compact form; for example, see figure 8.2.

- The term *sparse matrix* refers to the matrix shown on the left, with all the zeros in place; a *sparse index* locates a position in this matrix.

[1]These terms are also somewhat imprecise: the locations of the non-zeros in an unstructured matrix could be considered to form a pattern, if only one is willing to consider complex enough patterns. Such imprecision is of little importance though, as it is usually resolved by pragmatism: if employing 'structured' techniques results in better execution performance, then use them.

- The term *primary store* refers to the matrix on the right, from which the zeros have been excised; in general, the primary store need not be a matrix: it may be a vector, or a list, or a look-up table, for example. A *primary index* locates a position in the primary store.

$$\underset{\text{The } \textit{sparse matrix}}{\begin{bmatrix} (1,1) & (1,2) & 0 & 0 & 0 \\ (2,1) & (2,2) & (2,3) & 0 & 0 \\ 0 & (3,2) & (3,3) & (3,4) & 0 \\ 0 & 0 & (4,3) & (4,4) & (4,5) \\ 0 & 0 & 0 & (5,4) & (5,5) \end{bmatrix}} \longrightarrow \underset{\text{The } \textit{primary store}}{\begin{bmatrix} - & (1,1) & (1,2) \\ (2,1) & (2,2) & (2,3) \\ (3,2) & (3,3) & (3,4) \\ (4,3) & (4,4) & (4,5) \\ (5,4) & (5,5) & - \end{bmatrix}}$$

Figure 8.2: Storage for a tridiagonal matrix

Some means is required to map between sparse indices and primary indices. For a structured matrix, the mapping can be implemented as a computation. For example, as illustrated above, a tridiagonal matrix of order n can be stored as an $n \times 3$ matrix, with the first column of the primary store corresponding to the sub-diagonal of the sparse matrix, the second column to the diagonal, and the third column to the super-diagonal. The mapping from sparse index $[i, j]$ to primary index $[i', j']$ is

$$[i, j] \rightarrow [i, i - j + 2]$$

and the inverse is

$$[i', j'] \rightarrow [i', i' + j' - 2].$$

Note that the mapping is *not* 1-1: not every sparse index must correspond to a valid primary index (only the sparse indices of the non-zeros), and not every primary index must correspond to a valid sparse index (i.e. there may be locations in the primary store which do not contain any of the non-zeros of the sparse matrix). For example, in the the primary store of a tridiagonal matrix of order n, the locations $[1, 1]$ and $[n, 3]$ do not correspond to any position in the sparse matrix: they are *invalid*. In this work, such invalid locations are assigned the value zero; this allows them to be included in some computations (such as summation of the elements) as though they where valid positions.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 2 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \end{bmatrix} \longrightarrow \begin{array}{rl} \text{Primary store:} & \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix} \\ \text{Secondary store:} & \begin{bmatrix} i: & 1 & 1 & 2 & 3 & 3 & 4 & 4 & 5 & 5 \\ j: & 2 & 5 & 1 & 2 & 3 & 4 & 5 & 1 & 3 \end{bmatrix} \end{array}$$

Figure 8.3: An unstructured matrix and its primary and secondary stores

For an unstructured matrix, the mapping between sparse and primary indices is implemented as a look-up table: conceptually, the sparse index for each element in the primary store is recorded in a *secondary store*. See figure 8.3, in which a sparse index is stored as a row index (top row of the secondary store) and a column index (bottom row).

The element at location i' in the primary store contains the (non-zero) value of that element in the sparse matrix whose index is stored at location i' in the secondary store. This relationship can be represented as

$$A_p[i'] \equiv A[A_s[i']]$$

where A_p and A_s are the primary and secondary stores for matrix A. (It should be remembered that the primary and secondary stores need not be arrays, so indexing indicated in the figures should be interpreted in a generic manner; in this thesis, though, the stores are always arrays.)

There is no direct method for performing the inverse mapping: the only way to find that primary index which corresponds to a given sparse index is to search the secondary store for the sparse index. It is usually preferable to avoid the need for the inverse mapping, as searching is generally an expensive operation (and can be difficult to parallelize for some architectures).

For some unstructured matrices, it is sufficient to arrange the primary store so that the rows, say, correspond to the rows of the sparse matrix. Then the secondary store need contain only the column index of each sparse index [66]; see figure 8.4. Note that some zeros in the sparse matrix can be stored in the primary store as though they were non-zeros, in order to obtain the same number of stored elements in each row.

This is the type of storage used for unstructured matrices in this work: it is an efficient storage scheme when the number of non-zeros averaged over the rows is not much less than the maximum number of non-zeros in any one row. This form of sparsity will be referred to here as *row-wise* sparsity.

$$\begin{array}{ccccc} \text{Sparse matrix} & & \text{Primary store} & & \text{Secondary store} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 2 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \end{bmatrix} & \longrightarrow & \begin{bmatrix} 1 & 2 \\ 3 & 0 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} & & \begin{bmatrix} 2 & 5 \\ 1 & 2 \\ 2 & 3 \\ 4 & 5 \\ 1 & 3 \end{bmatrix} \end{array}$$

Figure 8.4: Storage for a row-wise sparse matrix

8.1.2 Reducing Execution Time

In addition to reducing the storage requirements of an implementation, the use of sparse programming techniques can reduce the execution time by allowing operations to be restricted to only the non-zero elements.

For example, in forming the product of a matrix A and a vector V, the i^{th} component of the product vector is given by multiplying the i^{th} row of A element-by-element with V and summing the elements of the result. Now if A is sparse, then only the non-zero elements in a given row need be considered in the multiplication/summation (as the zero elements all contribute nothing to the sum). So for, say, an order n tridiagonal matrix, matrix-vector multiplication requires only $3n-2$ multiplications and $2n-2$ additions, rather than the n^2 multiplications and $n(n-1)$ additions required when the matrix is dense.

8.2 Overview of Sparse Derivations

In this section, I describe the method used to incorporate sparse programming techniques into derived implementations.

Implementing an algorithm in such a way as to make effective use of sparse programming techniques is often treated as a task distinct from the task of implementing the algorithm in ‘dense’ form (i.e. without using sparse programming techniques). This is understandable, as a sparse implementation is often radically different from a dense implementation.

However, the approach I adopt in this thesis is to treat the use of sparse programming techniques as merely yet another implementation technique, in the same manner as parallelization or the implementation of abstract data types. Thus, the same specification is used for both sparse and dense implementations; just as the same specification is used regardless of whether the target architecture implements a sequential, vector or array model.

The derivation for a sparse implementation is almost the same as that for the dense implementation: the only difference is that the derivation for the sparse implementation includes an additional stage to incorporate sparse programming techniques.

For example, the derivation to produce a dense, sequential implementation is

SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ Common Sub-expressions $\longrightarrow$ Fortran77

while the derivation to produce a *sparse*, sequential implementation is

SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ $\underline{\text{Sparse}}$ $\longrightarrow$ Common Sub-expressions $\longrightarrow$ Fortran77.

That is, an SML specification is converted into a simple, functional form; is optimized by unfolding definitions and simplifying expressions; is then modified to use sparse programming techniques; and is then optimized and implementation in Fortran77.

Similarly, for sparse vector and array implementations:

SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ $\underline{\text{Sparse}}$ $\longrightarrow$ CRAY $\longrightarrow$ Common Sub-expressions $\longrightarrow$ Fortran77

SML $\longrightarrow$ λ-Calculus $\longrightarrow$ Unfolding $\longrightarrow$ Static Evaluation
$\longrightarrow$ $\underline{\text{Sparse}}$ $\longrightarrow$ Array $\longrightarrow$ Common Sub-expressions $\longrightarrow$ Fortran Plus Enhanced.

The term "sparse sub-derivation" is here a generic term: there are many forms of sparsity (e.g. tridiagonal and row-wise), each of which requires a somewhat different sub-derivation (though fortunately, the sub-derivations are quite similar): the appropriate sub-derivation is inserted into the derivation at the indicated point.

The positioning of the sub-derivation in the derivation is not arbitrary:

- Unfolding and Static Evaluation produce a simple form of a specification, facilitating the conversion to sparse programming techniques.

- The form of the sparse programming techniques is independent of the target architecture, so the sparse sub-derivation should come before the architecture specific stages of the derivation.

The above discussion omits one detail: which matrices in an algorithm are sparse, and what is the type of the sparsity, are properties of the data sets the algorithm is processing — as such, they cannot be automatically incorporated into an implementation by transformation: they must be specified at some point by the programmer. Thus, some means must be provided for the programmer to introduce information specifying that such-and-such a matrix is tridiagonal, or whatever; transformations can then use this information to optimize an implementation.

How such information is provided is mainly a matter of convenience: perhaps the most convenient method would be to include assertions in a specification. For example, if a matrix is input into the 'driver' function of a specification, then the driver function might also include an assertion that the matrix is tridiagonal and an assertion that is is row-wise sparse:

```
function init(n:int)
  =
    ⋮
    val A:real matrix = read_matrix([n,n]);
    assert: tridiagonal(A,n)
    assert: rowwise(A,[n,w])
    ⋮
```

These assertions indicate that A can be tridiagonal (of order n), or row-wise sparse (with n rows, each with w non-zeros). Each sparse sub-derivation would then use the appropriate assertion: for example, the tridiagonal sub-derivation would take A as being tridiagonal and ignore the `rowwise` assertion. The standard, dense derivation would ignore all such assertions.

Effective methods for incorporating information such as sparsity into a specification requires further research and is outside the scope of this thesis; in this thesis, I consider how such information can be used once it has been supplied.

8.3 The Tridiagonal Sub-derivation

In this section, I discuss the method used to produce tridiagonal implementations.

The starting point I assume is a specification containing information that a particular matrix is tridiagonal, and that the sparsity is of a particular type. This specification is processed by the initial stages of the derivation to produce a Statically Evaluated form (preserving the information about sparsity).

The task of the tridiagonal sub-derivation is to convert operations on the tridiagonal matrix into equivalent operations on the primary store.

To illustrate the method used to achieve this conversion, I will trace the conversion of matrix-vector multiplication, assuming that the matrix is tridiagonal.

8.3.1 Worked Example: Matrix-Vector Multiplication

Consider the (dense) expression for matrix-vector multiplication:

```
generate([n],λ[i]·
  reduce([n],λ[j]·times(element(A,[i,j]),element(V,[j]),+,0.0)
```

and assume that the matrix `A` is known to be tridiagonal, of order n. Since only the non-zero elements of `A` (that is, the tridiagonal elements) are to be stored, then the expression `element(A,[i,j])` may be thought of as representing a composite operation, as follows:

- test whether the indices i, j refer to a stored element;
- if so, return the stored value;
- otherwise, return zero.

That is, the expression `element(A,[i,j])` is equivalent to the conditional expression

```
if ([i,j] ∈ tridiagonal(n)) then element(A,[i,j]) else 0.0
```

where `tridiagonal(n)` represents the set of indices in the tridiagonal region.

Thus, the expression for matrix-vector multiplication can be written

```
generate([n], λ[i]·reduce([n],λ[j]·times(
  if ([i,j] ∈ tridiagonal(n)) then element(A,[i,j]) else 0.0,
  element(V,[j])),
  +, 0.0)).
```

This form has three advantages over the original form:

- It contains an explicit conditional expression that distinguishes the zeros from the non-zeros; this conditional expression can be manipulated by transformations.
- The zero elements are designated explicitly by the number zero (rather than by an application of `element`); again, this explicitness permits manipulation by transformations.
- All remaining applications of `element` are known to refer to stored values.

The explicitness of the the new form permits the expression to be simplified:

- The application of `times` can be propagated into the limbs of the conditional expression.

```
==> generate([n], λ[i]·reduce([n],λ[j]·
  if ([i,j] ∈ tridiagonal(n))
  then times(element(A,[i,j]),element(V,[j]))
  else times(0.0,element(V,[j])),
  +, 0.0))
```

- The resulting multiplication by zero can be simplified

```
==> generate([n], λ[i]·reduce([n],λ[j]·
  if ([i,j] ∈ tridiagonal(n))
  then times(element(A,[i,j]),element(V,[j]))
  else 0.0,
  +, 0.0))
```

- In the reduction, `i` is fixed, so the predicate `[i,j] ∈ tridiagonal(n)` is satisfied for, at most, three values of `j`:$\{i-1, i, i+1\}$. The reduction can thus be further simplified to give

```
==> generate([n],
  λ[i]·if (i=1) then 0.0 else element(A,[i,i-1])*element(V,[i-1])
  + element(A,[i,i)*element(V,[i])
  + if (i=n) then 0.0 else element(A,[i,i+1])*element(V,[i+1]))
```

  in which the conditional expressions check for the first and last rows of `A` (which have only two non-zeros rather than three).

In the expression produced by simplification, the matrix `A` is still a sparse matrix: it has n^2 elements, most of which are zeros; what has changed is that all applications of `element` refer to non-zeros.

Next, the non-zero region is mapped onto the primary store (an $n{\times}3$ matrix, denoted as `Ap`). The mapping from sparse index to primary index is

$$[i,j]{\rightarrow}[i,j-i+2]$$

which applied to the above expression gives (after some arithmetic simplification)

```
generate([n],
  λ[i]·if (i=1) then 0.0 else element(Ap,[i,1])*element(V[i-1])
  + element(Ap,[i,2])*element(V,[i])
  + if (i=n) then 0.0 else element(Ap,[i,3])*element(V[i+1]).
)
```

The above expression is the optimized form of matrix-vector multiplication: each element in the product vector requires two additions and, at most, three multiplications. The presence of the conditional expression to test for the first and last elements would reduce efficiency if preserved in an implementation. It is removed by later stages of the derivation.[2]

8.3.2 Method

The above example was intended to illustrate that the derivation of the sparse form has three main steps:

- **Introduction of conditional expressions** to explicitly test for zero/non-zero values.
- **Simplification** of the resulting expression.
- **Mapping** the non-zeros onto the primary store.

These steps are detailed below.

Introduction of Conditional Expressions

An application of `element` to a matrix which is marked as tridiagonal is transformed into a conditional expression which tests if the index refers to a non-zero value.

```
.sd.
  element(A:tridiagonal(n),[i,j])
  ==>
  if ([i,j] ∈ tridiagonal(n))
  then element(A:tridiagonal(n),[i,j])
  else 0.0
.sc.
```

[2]How it is removed depends on the target architecture; that is why removal is not attempted by the sparse sub-derivation. For example, for sequential and vector architectures, the conditional expression can be removed by 'loop peeling', whereby the first and last elements are implemented separately from the other elements, allowing the main part of the generation to be implemented as a simple loop with no conditional expression.

Simplification

Once the explicit tests for zero/non-zero elements have been introduced, transformations are applied to simplify expressions. Some of this simplification is similar to that performed by the Static Evaluation transformations (and indeed, the transformations used to perform the simplification are the same); of particular relevance are the evaluations involving zero, such as `.sd. x+0 ==> x .sc.`.

Other aspects of the simplification are specific to the sparse sub-derivations. In particular, an important simplification is the removal from reductions of the conditional expression introduced by the previous section. This removal is achieved by splitting reductions into a pair of reductions: one reduction over the region where the conditional is true; the other over the false region.

There are several types of splitting:

Tridiagonal region is sub-set of reduction region

If a reduction is performed over a shape $[n, n]$, and if the reduction's generating function tests for membership of the tridiagonal region, then the reduction is split into a reduction over the tridiagonal region, and a reduction over the rest of the shape.

```
.sd.
  reduce([n,n],
    λ[i]·if ([i] ∈ tridiagonal(n) then T else F,
    λx,y·r,init)
  ==>
  λx,y·r (reduce(tridiagonal(n),λ[i]·T,λx,y·r,init),
    reduce([n,n]-tridiagonal(n),λ[i]·T,λx,y·r,init))
  where S-S′ indicates set difference.
.sc.
```

Vector reductions

If a reduction is over one dimension, and the reduction's generating function tests for membership of the tridiagonal region, then those non-zeros that contribute to the reduction may form a sub-set of the tridiagonal region that can readily be separated from the rest of the region, such as the non-zeros in a given row or column.

For example:

```
.sd.
  reduce([n],
    λ[j]·if ([i,j] ∈ tridiagonal(n)) then T else F,
    λx,y·r,init)
  ==>
  λx,y·
  (
  λx,y·r
    (if (i=1) then 0.0 else λj·T (i-1))
    (λx,y·r (λj·T (i))
      (if (i=n) then 0.0 else λj·T (i+1))
    )
  )
  (reduce([n]-{i-1,i,i+1},λj·F,λx,y·r,init))
  where i is independent of j.
.sc.
```

Here, the three elements in row i that lie in the tridiagonal region have been separated.

A similar transformation can be applied for separating out a column of the tridiagonal region.

Two other forms of transformations are important for the sparse sub-derivations:

Reduction over zeros

The application of the previous transformations often results in reductions in which the generating function always returns zero. If the reducing function is addition or, less usually, multiplication, then the result is, of course, zero.

```
.sd.
  reduce(S,λi,j·0,real.plus,0)
  ==>
  0
.sc.

.sd.
  reduce(S,λi,j·0,real.times,0)
  ==>
  0
.sc.
```

Propagation of conditionals

As illustrated in the worked example, the propagation of function applications into conditional expressions can allow simplifications to takes place, when the predicate of the conditional is a test for membership of the tridiagonal region. Such propagation is performed by the transformation

```
.sd.
  f(if (i ∈ tridiagonal(n)) then T else F)
  ==>
  if (i ∈ tridiagonal(n)) then f(T) else f(F)
.sc.
```

(and equivalent transformations for functions of more than one argument).

Mapping onto Primary Store

After the simplification stage, operations on the sparse matrix are converted into operations on the primary store. This conversion is characterized by the mappings:

shapes: $tridiagonal(n)\rightarrow[n,3]$;

indices: $[i,j]\rightarrow[i,j-i+2]$ and the inverse $[i,j]\rightarrow[i,i+j-2]$ (for non-zero elements).

All array operations are expressed in terms of the three functions `element`, `generate` and `reduce`, so it is applications of these functions that must be converted.

`element` Each application of `element` is by now guaranteed to refer to a non-zero, so the conversion of `element` is a straightforward application of the above identities.

```
.sd.
  element(A:tridiagonal(n),[i,j])
  ==>
  element(A:[n,3],[i, j-i+2]).
.sc.
```

Some additional transformations can be applied to simplify the arithmetic expression produced by this transformation.

`generate` The conversion of `generate` primarily requires the application of the shape mapping to the shape and the *inverse* index mapping to the generating function. The conversion is somewhat more complex, though, due to the invalid locations ($[1,1]$ and $[n,3]$) in the primary store. It is assumed that these locations contain the value zero, so the generating function must return zero for those locations.

```
.sd.
  generate(tridiagonal(n),λi,j·g)
  ==>
  generate([n,3],
    λi,j′·if ([i,j]=[1,1] ∨ [i,j]=[n,3])
      then 0.0
      else λj·g (i+j′-2))
.sc.
```

Further transformations can be applied to remove the explicit test for the invalid locations when it is known that the generating function would return zero for those locations.

`reduce` The conversion of reductions is essentially the same as the conversion of generations, except that the generating function must return the identity element of the reducing function (which may or may not be zero) for the invalid locations. This ensures that including the invalid locations in a reduction does not alter the value of the reduction.

```
.sd.
  reduce(tridiagonal(n),λi,j·g,r,init)
  ==>
  reduce([n,3],λ[i,j′]·
    if ([i,j]=[1,1] ∨ [i,j]=[n,3]) then r0 else λj·g (j′),
    r,init)
  where r0 is an identity element of the reducing function r.
.sc.
```

Further transformations can be applied to remove the explicit test for the invalid locations when it is known that the generating function returns the identity element for those locations.

8.3.3 Imperative Implementations of Tridiagonal Form of Conjugate Gradient

Having derived the tridiagonal form of an algorithm, the later stages of the Fortran77, CRAY and DAP derivations are applied to produce implementations. Since these stages of the derivation are unchanged from the dense version (chapters 6 and 7), I shall simply present the implementations produced for Conjugate Gradient. Further, the only parts of CG that change for the tridiagonal version are two the matrix-vector products, in the computation of `r′` and the computation of `q′`. In the following sections, I present the implementations of these products for sequential, CRAY and DAP architectures, and discuss the efficiency of the CG implementations.

Tridiagonal Sequential Implementation of CG

Figure 8.5 shows the sequential implementations of the two matrix-vector products performed by Conjugate Gradient.

- Both of the implementations exhibit the effects of 'loop peeling': the first and last elements of `r1` and `q` are computed separately to avoid the need for a conditional expression in the main loop that computes the other elements.
- In the computation of `r′`, the matrix-vector product involves the transpose of `A`, but no explicit transposition is performed in the implementation; rather, the indices are modified: the elements of `A` that contribute to the value of the i^{th} element of the result are, in the standard and transposed versions respectively

```
val atq:real vector = transpose(a)*q;
val r':real vector = r-atq*alpha;
```

$\longrightarrow$

```
      r1(n) = r(n)-(A(n-1,3)*q(n-1)+A(n,2)*q(n))*alpha
      r1(1) = r(1)-(A(1, 2)*q(1)+A(2,1)*q(2))*alpha
      DO 1 i = 2,n-1
      r1(i) = r(i)-(A(i-1,3)*q(i-1)+
     .   A(i,2)*q(i)+A(i+1,1)*q(i+1))*alpha
1     CONTINUE
```

```
val q':real vector = a*r'+q*beta
```

$\longrightarrow$

```
      q(n) = A(n,1)*r1(n-1)+A(n,2)*r1(n)+q(n)*beta
      q(1) = A(1,2)*r1(1)+A(1,3)*r1(2)+q(1)*beta
      DO 1 i = 2, n-1
      q(i) = A(i,1)*r1(i-1)+A(i,2)*r1(i)
     .   +A(i,3)*r1(i+1)+q(i)*beta
1     CONTINUE
```

Figure 8.5: Sequential implementation of tridiagonal versions of matrix-vector products for CG

Standard		Transposed
`A(i,1)`	$\rightarrow$	`A(i-1,3)`
`A(i,2)`	$\rightarrow$	`A(i,2)`
`A(i,3)`	$\rightarrow$	`A(i+1,1)`

That the substitution of the right terms for the left terms is equivalent to performing matrix transposition is not, I believe, entirely obvious; yet it follows by the application of simple rules. Such results are often what present the greatest possibility for error when constructing an implementation manually.

Examination of the implementations shows that the sparse sub-derivation is effective in implementing these operations: the minimum number of computation required for the operations are performed with few overheads. Comparison of the execution times for the dense and tridiagonal versions is not very informative, since the number of computations has been reduced from $O(n^2)$ to $O(n)$, so clearly the tridiagonal version will execute much more quickly.

Such comparison does though indicate the effectiveness of the sub-derivation and the importance of sparse programming techniques. Figure 8.6 compares the execution times for dense and tridiagonal implementations on a sequential workstation and on the CRAY operating in scalar (sequential) mode. It should be noted that the derived tridiagonal version also allowed much larger problem sizes to be tackled: for the CRAY, the increase in problem size was approximately a thousandfold. The most obvious feature of these graphs is that the tridiagonal version's execution time does not noticeably increase!

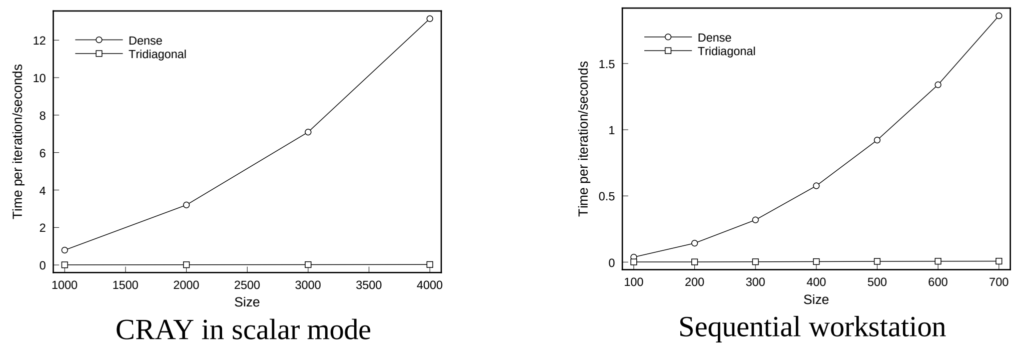


Figure 8.6: Comparison of sequential execution times for dense and tridiagonal versions of CG

Tridiagonal CRAY Implementation of CG

The tridiagonal CRAY implementation of CG is the same as the sequential implementation: the CRAY sub-derivation has no effect upon the tridiagonal form. There is little further to be gained by comparing the dense and tridiagonal execution times: the graph looks much like those in figure 8.6. However, two useful comparisons can be made:

- The matrix vector products include terms such as `A(i-1,3)*q(i-1)` and `A(i,1)*r1(i-1)`. It is possible that the index expressions inhibit vectorization; a comparison of execution times in scalar and vector modes may thus be informative.
- The CRAY software library contains implementations of the BLAS routines (Basic Linear Algebra Subroutines). One such routine performs the assignment

  $$\boxed{\mathtt{y:=}\alpha\mathtt{Ax+}\beta\mathtt{y}}$$

  where `A` is a sparse matrix with an arbitrary number of contiguous sub- and super-diagonals, `x` and `y` are dense vectors and α and β are scalars. This routine can be used to perform the computation of r′ and q′:

  ```
  val atq:real vector = transpose(a)*q;
  val r′:real vector = r-atq*alpha;
  ```

  ```
  val q′:real vector = a*r′+q*beta
  ```

  A comparison of the derived implementation and implementation using BLAS routine provides an indication of the absolute efficiency, though it must be remembered that the BLAS routine is capable of using banded matrices with any number of super- and sub-diagonal, not just tridiagonal matrices.

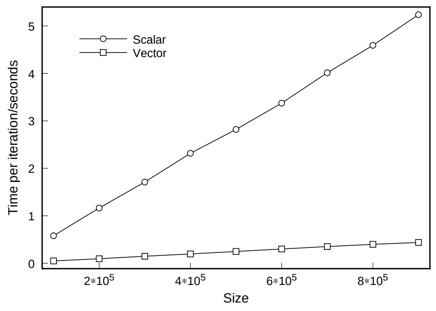


Figure 8.7: Execution times of derived tridiagonal implementation of CG in scalar and vector modes

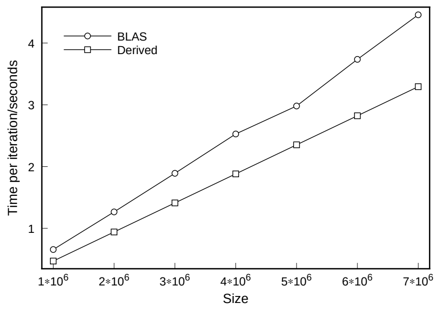


Figure 8.8: Execution times of derived tridiagonal implementation and implementation using BLAS routine

Figure 8.7 compares the performance of the derived tridiagonal implementation of CG in scalar and vector modes. The speedup for vector mode is 11-12: this is better than average speedup (8-10), indicating that the derived implementation is indeed well suited to vectorization.

Figure 8.8 compares the performance of the derived implementation against that of the BLAS-based implementation. The execution time of the derived implementation is between 20% to 30% less than that of the BLAS-based implementation, indicating that the derived routine is indeed quite efficient.

Tridiagonal DAP Implementation of Conjugate Gradient

As with the sequential and CRAY implementations of CG, the important aspects of the tridiagonal version are the two matrix-vector products, shown in figure 8.9.

- An expression of the form `A( ,c)` indicates the c^{th} column of matrix `A`.

```
val atq:real vector = transpose(a)*q;
val r':real vector = r-atq*alpha;
```

$\longrightarrow$

```
r1 = r-(shrp(A( ,3)*q)
  +A( ,2)*q+shlp(A( ,1)*q))*alpha
```

```
val q':real vector = a*r'+q*beta
```

$\longrightarrow$

```
q = A( ,1)*shrp(b)
  +A( ,2)*b+A( ,3)*shlp(b)+q*beta
```

Figure 8.9: DAP implementations of tridiagonal versions of matrix-vector products required by CG

- The functions `shrp` and `shlp` shift the elements of a vector to the right or left respectively, introducing a zero at one end.
- The tridiagonal (functional) form of matrix-vector multiplication was presented earlier; it is:

```
generate([n],
  λi·if (i=1) then 0.0 else element(Ap,[i,1])*element(V[i-1])
  + element(Ap,[i,2])*element(V,[i])
  + if (i=n) then 0.0 else element(Ap,[i,3])*element(V[i+1]).
)
```

  This form contains conditional expressions for the first and last elements of the product vector, to account for the first and last row contributing one less term to the sums than the other rows. The DAP implementations of matrix-vector multiplication does not contain conditional expressions: the vector shifting in the DAP expression has the effect of introducing a zero at the locations specified by the conditional expressions, so the conditionals are redundant.

- The computation of `r′` requires a product with `A` transposed. The DAP implementation does not perform an explicit transposition; rather the shifting is modified:

Standard		Transposed
`A( ,1)*shrp(b)`	$\longrightarrow$	`shrp(A( ,3)*q)`
`A( ,2)*b`	$\longrightarrow$	`A( ,2)*q`
`A( ,3)*shlp(b)`	$\longrightarrow$	`shlp(A( ,1)*q)`

  As with the sequential implementation, is is not obvious that such modification of the shifting is equivalent to transposing the matrix.

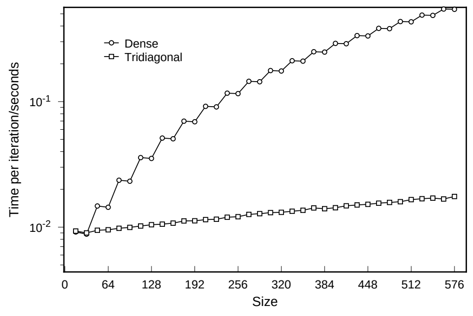


Figure 8.10: Execution times for tridiagonal and dense DAP implementations of CG

The DAP implementations of the matrix-vector products are clearly seen to be efficient. As for the sequential implementations though, a comparison of execution performance for tridiagonal and dense implementations, as shown in figure 8.10, illustrates the effectiveness of the tridiagonal sub-derivation and the importance of using sparse programming techniques (note that the time scale is logarithmic).

8.3.4 Summary of Tridiagonal Sub-derivation

The tridiagonal sub-derivation is effective at optimizing standard definitions of operations to make efficient use of the compact storage scheme used for tridiagonal matrices. In addition, the optimized form is suitable for processing by further sub-derivations which produce efficient sequential, CRAY and DAP implementations.

8.4 Row-wise Sparse Matrices

In this section, I discuss the implementations of Conjugate Gradient for a row-wise sparse matrix, obtained by modifying the tridiagonal sub-derivation.

To recap: a row-wise sparse matrix is a matrix in which the number of elements to be stored is the same in each row; that number is the *width* of the sparse matrix. (The stored elements should mostly be non-zeros, but some may be zeros.)

A row-wise sparse matrix A with n rows and of width w is stored as an $n{\times}w$ primary store A_p and an $n{\times}w$ secondary store A_s. A location $[i, j']$ in the primary store corresponds to the location $[i, A_s[i, j']]$ in the sparse matrix: i.e.

$$A_p[i, j'] \equiv A[A_s[i, j']] \quad .$$

See figure 8.11.

$$\begin{array}{cccc}
\text{Sparse matrix} & & \text{Primary store} & \text{Secondary store} \\
\begin{bmatrix} 0 & 1 & 0 & 0 & 2 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \end{bmatrix} & \longrightarrow &
\begin{bmatrix} 1 & 2 \\ 3 & 0 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} &
\begin{bmatrix} 2 & 5 \\ 1 & 2 \\ 2 & 3 \\ 4 & 5 \\ 1 & 3 \end{bmatrix}
\end{array}$$

Figure 8.11: Storing a row-wise sparse matrix

8.4.1 The Row-wise Sub-derivation

Details of the row-wise sub-derivation, including a worked example, are presented in appendix B.5.

8.4.2 Imperative Implementations of Row-wise Sparse Form of CG

Figure 8.12 shows the sequential implementation of the row-wise sparse versions of computations of CG involving matrix-vector products.

- The $n{\times}n$ sparse matrix `A` is represented as a primary store `Ap` and secondary store `As`, each of size $n{\times}w$, where w is the maximum number of non-zeros in any row.

```
val atq:real vector = transpose(a)*q;   →       do 230 i = 1,n,1
                                                 atq(j) = 0.0
                                             230 continue

                                                 do 250 i = 1,n,1
                                                 do 240 k = 1,w,1
                                                 j = As(i,k)
                                                 atq(j) = atq(j)+Ap(i,k)*q(i)
                                             240 continue
                                             250 continue
```

```
val q′:real vector = a*r′+q*beta   →       do 340 i = 1,n,1
                                            g63 = 0.0
                                            do 330 k = 1,w,1
                                            g63 = g63+Ap(i,k)*r1(As(i, k))
                                        330 continue
                                            q(i) = g63+q(i)*beta
                                        340 continue
```

Figure 8.12: The row-wise sparse versions of the matrix-vector product computations of CG

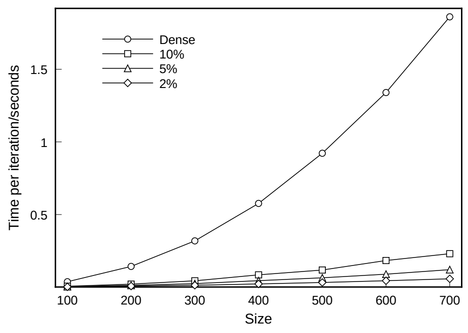
- The transposition of `A` required for the computation of `atq` is not explicitly performed in the implementation; rather, comparing with a non-transposed product, the indirect indexing is shifted from the operand vector `V` to the product vector `P`:

  Standard Transposed

  ```
  P(i) = P(i)+Ap(i,k)*V(As(i,k))   →   P(As(i,k)) = P(As(i,k))+Ap(i,k)*V(i)
  ```

  As with the implementations for the tridiagonal form, it is not obvious that this is how transposition should incorporated into the product.

The primary factor contributing to efficiency in the row-wise sparse implementations of CG is the restriction of the summations to only the non-zero region; this restriction was achieved by the sparse sub-derivation. However, the indirect array indexing introduced to achieve this restriction (in expressions such as `r1(As(i, k))`) introduces overheads that may offset the gain in efficiency.



(a)

Percentage non-zeros	Number of operations: dense/sparse	Execution times: dense/sparse
2	50	25
5	20	13
10	10	7

(b)

Figure 8.13: Comparison of sequential execution times of dense and row-wise sparse versions of CG

Figure 8.13(a) compares the execution times of the dense implementation of CG against execution times of the row-wise version, for row-wise sparse matrices containing 2%, 5% and 10% non-zeros in each row. (The non-zeros were randomly scattered along each row, though it was ensured that all the diagonal elements were non-zero, as this is required for convergence of CG.)

Figure 8.13(b) summarizes the results: on average, execution of the row-wise sparse version (for the various proportions of non-zeros) was 25, 13 and 7 times faster than the dense version; the corresponding reductions in the number of required computations are approximately 50, 20 and 10. These results suggest that the indirect indexing does indeed introduce considerable overheads (proportionately more the fewer the elements), but that the row-wise sparse form still provides a useful reduction in execution times; and, of course, the row-wise sparse version permits much larger problem sizes to be tackled, due to the reduced storage requirements.

8.4.3 CRAY Implementation of Row-wise Sparse Version of CG

The CRAY row-wise sparse implementation of CG is the same as the sequential row-wise sparse implementation (figure 8.12): the CRAY sub-derivation has no effect upon the sparse form. As before, the CRAY implementation can be evaluated by comparing its execution performance in scalar and vector mode, and by comparing its performance with the dense version of CG.

As with the sequential implementation, the primary concern for efficiency is to what extent the reduction in operations obtained by restricting summations to the non-zeros is offset by the overheads introduced by the indirect array indexing. The latter is likely to have a greater, detrimental effect than in the sequential implementation since it will inhibit vectorization. However, the X-MP processors do have 'scatter-gather' units designed specifically to implement such indirect addressing, so the effect on vectorization may be ameliorated somewhat.

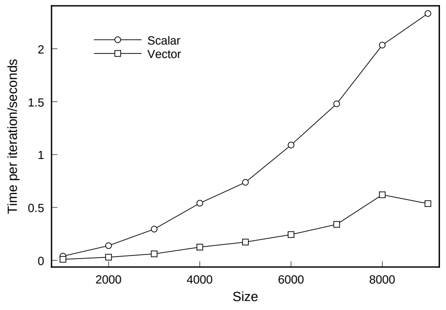
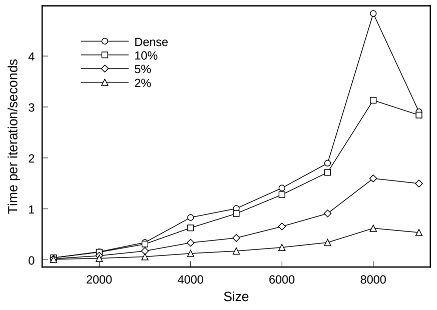


Figure 8.14: CRAY execution times of row-wise version of CG in scalar and vector modes (2% non-zeros)

Figure 8.14 compares the scalar and vector performance of the row-wise version of CG. The average speedup of the vector version over the scalar version is about 4, suggesting only moderately effective use of the processor vector units (good use of the vector units normally results in a speedup of between 8 and 10).



(a)

Percentage non-zeros	Number of operations: dense/sparse	Execution times: dense/sparse
2	50	5.8
5	20	2.2
10	10	1.15

(b)

Figure 8.15: Comparison of CRAY execution times of dense and row-wise sparse versions of CG

Figure 8.15(a) compares the execution performance of the dense and row-wise sparse versions of CG; figure 8.15(b) summarizes the results. For a matrix with 2% non-zeros, the number of computations performed by the sparse version is reduced by a factor of 50 compared with the dense version, but the execution time is reduced by a factor of only 6. For a matrix with 10% non-zeros, the execution time of the sparse version is almost the same as the dense version.

These results suggest that the indirect indexing does indeed have a considerable detrimental effect on performance; however, the row-wise sparse version still provides a useful reduction in execution time for matrices with a low proportion of non-zeros, and it permits larger problem sized to be tackled due to the reduced storage requirements.

8.4.4 DAP

The indirect array indexing required for the implementation of row-wise sparsity is virtually impossible to implement efficiently on the DAP array processor (the hardware does not suppose the communication required for the indirect indexing), so I do not consider an implementation for the DAP.

8.4.5 Summary of Row-wise Sparse Sub-derivation

The storage scheme chosen for row-wise sparse matrices is considerably more difficult to use efficiently than the tridiagonal scheme, due to the requirement for an index lookup table. However, the row-wise sparse sub-derivation does effectively tailor implementations to the chosen scheme.

8.5 Summary

The types of sparsity considered in this work (tridiagonal and row-wise) are only two examples of many possible types of sparsity: the extension to other types of sparsity should be a simple matter; indeed, the tridiagonal sub-derivation has already been provisionally extended to permit banded matrices with any number of sub- and super-diagonal to be used.

The sparse sub-derivations discussed in this chapter are effective at automatically tailoring programs to use the chosen storage schemes for at least a restricted, though widely used, class of algorithm. Efficient, imperative implementations for sequential, vector and array architectures can be derived using the same sub-derivations as for the dense form.

Further research is required to explore the limits of what can be achieved for sparse data structures by automatic tailoring such as discussed in this chapter. In particular, algorithms which *compute* sparse data structures may present greater difficulties than algorithms in which the sparse data structures are input data (as was assumed for this thesis). Also, specifying the nature of the sparsity may become a problem when more than one type of sparsity is present simultaneously (e.g. one matrix is tridiagonal, another row-wise sparse): this situation does not seem to arise often though.

Chapter 9

Related Work

The work discussed in this thesis touches on several fields of research: specification of numerical algorithms; parallel programming in both imperative and functional contexts, and particularly parallel programming for numerical algorithms; program transformation and derivation. In this chapter, I discuss the relationship between my work and these research areas.

9.1 Parallel Programming

Imperative languages are the languages predominantly used to program the high-performance computer systems considered in this thesis. In this section, I briefly discuss some of the issues more closely related to the work reported in this thesis. The two important issues are the vectorization/parallelization of sequential Fortran programs and the array-based dialects of Fortran that have been recently developed.

In addition, extensive research has been carried out on using functional languages to program parallel computer systems; in the hope that parallel systems would permit functional programs to compete on performance with imperative programs and in the hope that the greater expressiveness and simpler semantics of functional languages would ameliorate the difficulties encountered in programming such systems.

9.2 Parallelizing Compilers

The majority of programs that are executed by parallel computers are expressed in Fortran77. Fortran77 is inherently sequential, so compilers for parallel systems must perform extensive analysis of programs to obtain parallel execution, primarily by executing multiple iterations of `DO` loops simultaneously.

For example, the loop

```
  DO 1 i=1,n
  V(i) = 0
1 CONTINUE
```

has meaning: "assign 0 to V(1); then assign 0 to V(2); etc.". In this example, it is obvious that all the assignments can be performed simultaneously.

9.2.1 Dependence Analysis

In general, the conversion of a loop from sequential to parallel execution is complex. The basic tool in the conversion is the loop's *dependence graph*, which delineates the effect one statement in a loop has on other statements, or the effect one loop iteration has on another loop iteration. For example, in the loop

```
  DO 1 i=1,N
  A(i) = B(i)
  C(i) = A(i+1)
1 CONTINUE
```

the assignment to `A(i)` for some i cannot be performed until the value of `A(i)` has been used in the assignment to `C(i-1)`: the second statement is said to be dependent on the first.

Dependencies such as this can be represented as arcs in a graph, with the statements as nodes of the graph: if the node for a statement is linked to the node for a second statement, the second depends on the first. In general, the construction of the dependence graph for a loop is intractable; two of the reasons for this intractability are:

- To decide if there exists a dependence between a statement which uses the values of elements of an array and a statement that assigns elements of that array, it is necessary to determine the relationship between the indices for the elements used and the indices for the elements assigned (if the two are disjoint, there is no dependence between the statements). But array indices can be arbitrary expressions, whose values are unknown until program execution.

- A Fortran `DO` loop can be entered and exited at arbitrary points (using `GOTO` statements). It may not be possible to decide for which values of the loop index a particular statement is executed.

Worst case approximations may thus be required when constructing the dependence graph; for example, it may be assumed that a statement which assigns elements of an array affects all the elements of that array.

Given a dependence graph, many problems occurring in converting a loop from sequential to parallel execution may be expressed as well known graph problems. For example, if a loop's graph is acyclic, then it is always possible to execute all iterations of the loop simultaneously (though it may be necessary to reorder the statements in the loop).

An important class of problems in parallelizing loops is the removal of dependencies. For example, the loop

```
  DO 1 i=1,N
  t = A(i)
  B(i) = t
1 CONTINUE
```

contains a dependency induced by all iterations using the same scalar variable `t` to store a value temporarily. The dependency can be broken by replacing `t` with an array:

```
  DO 1 i=1,N
  T(i) = A(i)
  B(i) = T(i)
1 CONTINUE
```

This second form of the loop can then be executed in parallel.

Much of the analysis for converting loops into parallel form is the same whether the computer system is a vector processor, a shared memory multi-processor or a distributed memory multi-processor. Where compilers for these architectures differ is primarily in their treatment of nested loops: for vector systems, the innermost loop is the one executed in parallel, with outer loops executed sequentially; for multi-processor systems, the outermost loop is executed in parallel (as that provides the greatest granularity of computation). Compilers may change the order of nested loops if the inner or outer loop cannot be executed in parallel, or if another loop has better characteristics (for example, it is generally better to parallelize a loop with a large number of iterations).

9.2.2 Code Generation

A compiler computes an (approximate) dependence graph and perhaps manipulates a program to minimize dependances. The dependence graph may contain information that all the iterations of some loop can be executed in parallel; the compiler uses this information in generating instructions for the target system.

- For a vector architecture, with finite length vector units, the iterations of the loop will be divided into sections to match the length of the vector units. Each section is pipelined for efficient execution.

- For shared memory multi-processors, the iterations will be distributed over the processors. The compiler may arrange the distribution, or the operating system may arrange the distribution when the program is executed. In addition, the compiler must generally insert synchronization commands to ensure that no processor proceeds beyond a loop until all the processors have completed executing the loop.

- For distributed memory multi-processors, the iterations will again be distributed either by the compiler or the operating system. The compiler must also insert communication commands to obtain for each processor any required data that is stored on another processor.

  Since communication is normally several orders of magnitude slower than other processor operations, it is vital that communication be kept to a minimum. The amount of communication will depend on the way data is distributed over the processors, so the distribution can have a considerable effect on the performance. Unfortunately, automatically designing an optimum distribution is an intractable problem, so it is normally left to the user to design the distribution. The compiler can then insert code to perform the necessary communication.

9.2.3 Survey

Vectorizing and parallelizing Fortran77 (and other sequential languages) have been the subject of intense research for a couple of decades, and continue to be so now. For example, in 1974 Lamport [68] discussed manipulating loops to obtain better parallel execution; while Dowling [31] discussed extensions to that work in 1990; and in1987 Allen and Kennedy [2] discussed using parallelization and vectorization techniques to convert a Fortran77 program into a Fortran90 ("Fortran8x" as it was then known) program, rather than simply compiling the program for execution. Extensive details of vectorization/parallelization can be found in the books by Zima and Chapman [100] and by Wolfe [101].

This research area is not directly related to this thesis, in that the results of the research are not employed in the derivations presented here. However, the research is important for this thesis since Fortran77 is currently the only feasible language for programming many vector and parallel computer systems (and that has consequences for derivations). In addition, the intractability of many of the problems that arise in vectorization/parallelization is a major factor motivating research into alternative approaches to programming high-performance computer systems; the work reported here may be viewed as one alternative.

9.3 Array-based Fortran Dialects

Imperative languages have been extended to support parallel programming. The extensions range from subroutine libraries that are little more than interfaces to operating system routines, to entirely new languages such as ADA, which where designed with parallel execution in mind.

Of particular relevance to this thesis are the array extensions to Fortran provided by languages such as Fortran90 [79], Connection Machine Fortran [94], Fortran Plus Enhanced [1], Fortran-D [34] and Vienna Fortran [10].

These extensions provide, to some extent, an array abstract data type: many common operations such as the elementwise addition of two arrays are provided as pure functions (denoted using the usual + operator). Vienna Fortran is distinguished from the others in its advanced support for *data templates*, which permit the programmer to define the distribution of data on distributed memory systems.

Recently, many of the features of these array-based Fortran dialects have been coalesced into a single language called High Performance Fortran [59]. The language definition is still under review and there are no widely available HPF compilers available yet.

In some ways, the array extensions to Fortran may be viewed as an attempt to introduce into Fortran some of the features of functional languages (though it is unlikely that the language developers consider it in that manner): expressions permit array operations to be denoted in a high-level, machine-independent manner that allows operations to be succinctly combined and that facilitates analysis.

It is thus natural to ask if the wide-spread use of array-based Fortran would render irrelevant the work reported in this thesis, since programmers would be presented with array operations that are almost the same as those provided by the array ADT used for this thesis. Of course, I would answer in the negative:

- The array-based Fortran dialects fall short of providing complete array ADTs, since, as is common with Fortran dialects, the language designers have restricted themselves to the simpler operations which can be implemented efficiently: there is no standard matrix-product function for example and no means to apply a programmer defined function in an elementwise manner.
- Some of the Fortran dialects do provide module mechanisms for hiding implementation details, but efficiency considerations will probably force programmers to continue using subroutines as their main (if any) decomposition mechanism. What the derivational approach offers over Fortran in any form is a separation of the issues of specifying and algorithm and implementing the algorithm.
- The expression-based array operations are likely to impose just as high overheads on Fortran implementations as on functional implementations. The developers of compilers for the Fortran dialects will have to address issues such as destructively updating arrays, but they will have to address the issue in the context of an already complex compilation system. The derivational approach allows implementation issues to be separated and addressed more easily.

Thus, the primary relevance of the array-based dialects of Fortran for this thesis will probably arise from their use as programming models replacing the ill-defined model provided by Fortran77. (It should be simpler to derive HPF implementations than Fortran77 implementations.)

9.4 Functional Languages

Functional languages have been used in combination with parallel architectures in several ways.

Firstly, the semantics of functional programming languages generally permit great freedom in the order in which operations are performed; a compiler can use this freedom to introduce parallel execution. A general, widely used method for parallel execution is *parallel graph reduction* (discussed below).

Other methods include interpreting higher-order functions as templates for parallelism (often called *skeletons*): the functional arguments to the higher-order function represent computations that are to be performed while the high-order function itself represents a (potentially parallel) control structure for performing the computations (see below).

In addition, operations such as mappings and reductions on abstract data types such as lists and arrays can be interpreted for parallel execution.

Secondly, functional programming languages can be extended with operations that grant explicit control of processes to the programmer: typically the programmer specifies that a new process is to be created to evaluate some expression. For example, a programmer might construct the expression

```
F(future(f(x)),future(g(x)),h(x))
```

which specifies that a process should be created to evaluate each of the applications of `f` and `g`. The process that spawns the new processes proceeds with the evaluation of the application of `h`. If the spawned processes have not yet completed, the initial process will block until they do complete and provide the values of the expressions they evaluated. The initial process then evaluates the application of `F`.

Explicit process control is not directly relevant to the derivations discussed here, but it is likely to be relevant when the derivations are extended to multi-processor systems.

In addition, several computer systems have been developed specifically to support parallel execution of functional programming languages. The importance of special hardware is that it offers the possibility in a radical shift in the relative performance of functional and imperative languages that would reduce the need for the construction of an imperative implementation of a functional specification. However, the current view seems to be that special hardware is too expensive for practical use (and that improving the performance of functional languages on standard hardware through advanced compilation/transformation techniques is more promising).

9.4.1 Parallel Graph Reduction

A functional program may be viewed as a graph, with operations such as function application and λ-binding represented as nodes and data represented as leaves. Execution of the program can then be performed as manipulation of the graph, called *graph reduction*.

For example, execution of the program

```
λx·if (eq(x,1)) then 1 else 2 (0)
```

can be represented as shown in figure 9.1.

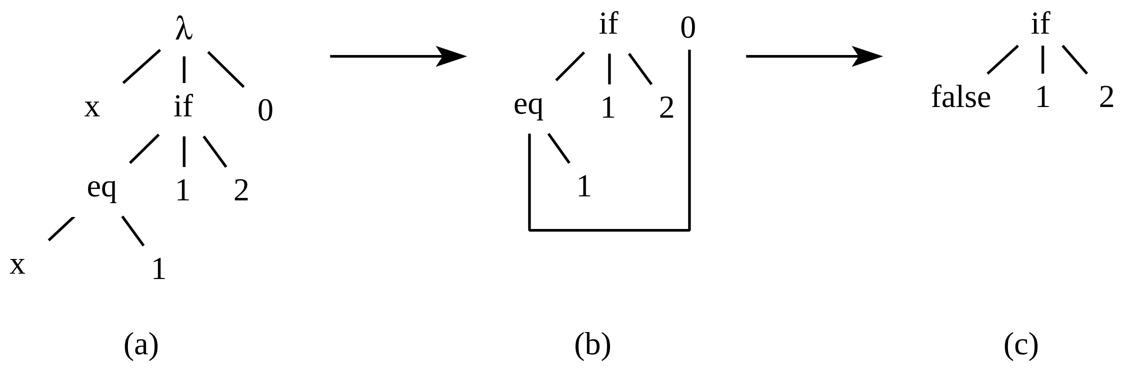


Figure 9.1: Graph reduction of λ `x·[if (x=1) then 1 else 2] (0)`

One of the advantages of implementing functional languages using graph reduction is that graph reduction is readily interpreted for parallel execution: any sections of a graph which do not share sub-graphs can be reduced simultaneously. Typically, parallel reduction is performed by initiating a process on the root node of the graph; when this process encounters an opportunity for parallel execution (such as a λ-expression which introduces several bindings), it creates one or more new processes to perform parts of the reduction, while it too performs part of the reduction.

Some of the problems encountered in parallel graph reduction are:

- As a process reduces some part of the graph, it may require the value of a section of the graph that is being reduced by another process. In such circumstances, the former process is suspended until the latter process completes its reduction.
- Functional programs often contain too much potential parallelism: if a new process were created to realize each opportunity for parallel execution, typical computer systems would become overloaded, with the combined demand of all the processes (for memory, say) outstripping the available resources.
- On a distributed memory system, a section of a graph stored on one processor may reference data stored on another processor. This clearly requires the operating system/language system to support communication, but it also complicates automatic storage management (using 'garbage collection') since a given section of storage cannot be deallocated until all references (including references from other processors) to it have been cleared.

Parallel graph reduction has been implemented on many multi-processor systems, including some systems constructed specifically to support graph reduction.

Graph reduction is important for my work for several reasons:

- It is probably the most popular approach for implementing functional programs and is thus the obvious competitor against the derivational approach advocated here. For the type of algorithms discussed here, the derivational approach generally provides better execution performance; but it must be remembered that graph reduction is entirely general (implementing any valid program), whereas the derivations reported here are less ambitious.
- Many of the optimization techniques (such as unfolding functions and converting tail recursive functions to iterative form) are shared by graph reducers and derivations.

9.4.2 Parallelism through Skeletons

Higher-order functions are used to capture common schemes of computation. Often, the schemes are suitable for interpreting for process parallelism.

For example, a 'divide-and-conquer' scheme to compute some function can be viewed as proceeding as follows: a complex argument to the function is recursively decomposed into simpler parts until each of the parts corresponds to some base case for which the result of the function is known; the function is applied to each of the base cases to produce a set of partial results; the partial results are then recursively combined (in the inverse order to the decomposition of the argument) to produce the final result.

A higher-order function can be defined to capture this scheme of computation. The high-order function may take four arguments:

- a function that tests if a value should be decomposed;
- a function to perform the decomposition if required;
- a function to produce results for the base cases;
- and a function to combine partial results.

Such a higher-order function can be interpreted for process parallelism: a single process is to perform the computation specified by the higher-order function;

- if the argument is not a base case, the processes decomposes the argument into a set of simpler values and creates a set of processes to perform the recursive application of the scheme to each simpler value; the initial process waits for each spawned process to return a partial result, combines the partial results and then terminates, returning the combined result;

- if the argument is a base case, the process terminates, returning the value corresponding to the base case.

Thus, the decomposition of complex values gives rise to a tree of processes. The leaves of the tree are processes performing base-case computations. The recombination of partial results corresponds to the recollapse of the tree into a single processes.

Skeletons are important for my work as they support one of the underlying tenets of this thesis: that the superior expressiveness of functional languages can be effectively harnessed for parallel programming. In addition, my use of `generate` and `reduce` for data parallel programming could be viewed as a particular example of the use of skeletons.

9.4.3 Survey

Architectures designed specifically to support parallel execution of functional languages have been investigated by several research teams. Systems based on graph reduction include the MaRS machine [27], the Dutch machine [53], the ν-STG machine [60], the GRIP machine [87, 86] and the Flagship machine [65] (which is an extension of the ALICE system [28, 50].) In addition, the SYNAPSE machine [73] was designed to support parallel execution of Lisp programs.

Other research has investigated the execution of functional programs on standard multi-processor systems: Peyton Jones details parallel graph reduction in his book [86]; Goldberg [40] discusses the implementation of the ALFL language on a distributed memory system and a shared memory system; Loogen *et al.* discuss graph reduction on a transputer system in [70]; the PARCEL project (Harrison *et al.*) [46, 51] investigated the parallel execution of Lisp programs; Larus and Hilfinger use the CURARE transformation system to optimize Lisp for parallel execution [69].

Jouret [62] discusses the implementation of a functional language on SIMD architectures. He defines a set of functions that have operational semantics well-suited to SIMD architectures, similar in concept to the Array form defined in this thesis. His work differs from that reported here in that his task is to implement a program containing applications of the data-parallel functions, whereas a derivation must first convert expressions into applications of Array form functions.

Interpreting high-order functions and ADT operations for parallel execution has been the subject of intensive research recently. Higher-order functions have been discussed by Harrison [47], Bratvold [24], Maaßen [71] (who also discusses parallel data structures) and Darlington (*et al.*) [29]. Parallel data-structures have been discussed by Hains and Mullin [43], Merral and Padget [78], Hill [55, 56] and Axford and Joy [3].

Extending functional languages to include process control has been investigated by several people. Gabriel *et al.* report on the QLisp language in [36, 41]; McGeharty and Krall discuss FutureLisp in [75, 76]; Halstead *et al.* discuss MultiLisp in [44] and its extension (Mul-T) in [67]; Giacalone *et al.* discuss the Facile language in [38].

A recent review of parallel functional programming by Hammond appeared in [45].

9.5 Transformation and Derivation

There is a large volume of literature on program transformations and derivations. For the purposes of this discussion, I shall divide the literature into several categories, though the categories are certainly not orthogonal.

Program Improvement: the use of transformations to recast a given program into a more efficient form.

Transformations can be used to formally recast a program into an equivalent but more efficient form. The initial and final form are generally expressed using the same syntax. An early example is Burstall's and Darlington's fold/unfold transformations [25] which improve the efficiency of systems of recursive equations. Improving the efficient of a recursive function has been discussed further by Boiten [16] and by Harrison and Khoshnevisan [48], while Kaser *et al.* discuss the effect of unfolding on systems of mutually recursive functions in [64]. Pettorossi and Burstall provide some examples of transformational improvement in [85].

Bird and Meertens [13, 77, 14, 12] have discussed transformations (or mathematical rules) based on formal properties of programs (their discussion is motivated partly by a desire to improve program efficiency, partly by a desire to provide a formal basis for program development). The so-called Bird-Merteens formalism has been further discussed by Malcolm in [72], by Backhouse in [6] and by Skillicorn in [92].

Wadler discusses program improvement by eliminating intermediate lists (*deforestation*) in [96]; this is discussed further by Peyton Jones *et al.* in [39]. Removal of intermediate data structures in also discussed by McCroksy [74] and Hennessy (*et al.*) [89] but they do not use a transformational approach.

Deforestation is closely related to Static Evaluation (discussed in this thesis). Both Wadler and Peyton Jones concentrate mainly on list-based data structures, while my work is primarily concerned with arrays; Peyton Jones does consider arrays, but only in terms of their representation as lists, so that after deforestation, array operations are recursive functions; this is undesirable as it inhibits parallelization. In contrast, after Static Evaluation, array operations are still expressed using the basic array functions.

In McCrosky's work, array operations such as transpose are viewed as transformations on array indices ($[i, j] \rightarrow [j, i]$); he removes intermediate arrays by composing such transformations. Hennessy takes a very different approach from mine; his is based on manipulating a data-flow graph of a program (as part of a compilation process).

Derivation: the use of transformations to obtain an implementation from some higher-level form.

The use of transformations to obtain a more concrete version of a high-level form has been discussed by Boyle (*et al.*) in [23, 20, 22, 21]. Roman and Wilcox discuss the tailoring of implementations to particular architectures in [88] while Skillicorn discusses the use of computational cost to direct tailoring [91]. In [35] Fradet and Métayer discusses the use of transformations to compile functional programs.

Morgan (*et al.*) [81] and Harrison (*et al.*) [49] have discussed the use of formal rules to implement (or realize) abstract data types. The realization of ADTs is an important part of the derivational process, but in my work it is performed mainly by function unfolding together with the implementation of primitive functions such as `generate`. Their work may be useful though in a formal analysis of aspects of derivations such as the conversion of a sparse matrix into a compact store.

Theory: the transformational/derivational method itself.

The CIP group, consisting of Bauer, Möller, Partsch and Pepper, has been influential in the investigation of the formal aspects of program transformation: [8, 7, 9]. Partsch has also done further work as part of the STOP project [83, 15]. Others working in this area include Pepper [84], Singh [90] and Zhu [103].

That research has been concerned mainly with the methodological aspects of transformations: formal theory and analysis of the transformation process, provision of software tools, etc. The work reported in this thesis is concerned more with the practical application of program derivation to enhance the productivity of programmers.

Systems: transformation tools.

Partsch gives an overview of various transformations systems in [82]; while that survey is somewhat old (1983), much of it is still relevant. Feather discusses the ZAP system in [32, 33]; Darlington, Harrison, *et al.* discuss the transformation system that forms part of the ALICE project in [30]; Yang and Choo discuss the Meta-Crystal system in [102]; Smith discusses the KIDS system in [93]; Hildum and Cohen discuss a system (which, it seems, has no name!) in [54]; the PROSPECTRA system is discussed in [63].

Several authors report that a major issue still to be addressed in transformation systems is the control of the derivation process; i.e. the specification of *strategies* to achieve some goal. The approach advocated by Boyle is to define a sequence of *canonical forms* that achieves a goal (the conversion from some initial form to some final form): consideration of strategy is then reduced to ensuring the transformations convert one canonical form into the next. (Informally, the intermediate forms discussed in this thesis may be viewed as canonical forms, though they probably require formalization before being granted that status.)

The use of normal forms was discussed at least as early as1970 by Boyle, e.g. [17] but the topic seems to be enjoying renewed interest; for example Hoare in [58].

Imperative: the use of transformations for imperative program development.

Functional languages generally provide more fertile ground for transformations than do imperative language, due the simpler semantics of functional languages. However, some researchers have been labouring to apply transformational techniques in attempts to formalize the development of imperative programs, both sequential and parallel.

In [4] Back develops transformations for a simple imperative language, with semantics defined using Hoare's/Dijkstra's precondition calculus. Back and Sere [5], use a similar basis for developing parallel programs from sequential programs, as does Gribomont [42]. Von Wright [95] uses Back's semantics to develop a framework for formally refining a data type. Morgan and Gardiner [81] also tackle this problem.

In addition, much of the literature for vectorization/parallelization of Fortran is discussed in terms of transformations (though it is debatable as to whether the term as used in that field means the same as the term used in this thesis).

Imperative transformation is perhaps most relevant to my work as an example of how not to approach the formal construction of parallel software. It is my opinion that it is vastly preferable to choose some starting point more amenable to manipulation than sequential, imperative programs (even if the programming language is minimalist).

Chapter 10

Conclusions

In this chapter, I summarize and assess the work presented in this thesis, and suggest areas for further research:

- **Summary**
- **Assessment**
- **Further Work**

10.1 Summary of Thesis

Numerical mathematical algorithms are the dominant consumers of the computational resources available to the research community. For the foreseeable future, their efficient implementation is likely to be an important area of work for computer scientists.

Numerical algorithms typically have elegant mathematical definitions (or at least have the potential for elegant definition) that facilitate understanding and analysis. However, the construction of efficient implementations for high-performance computer systems is a difficult task, as efficient implementations are typically dominated by architectural considerations rather than algorithmic considerations.

The consequences of the radically different natures of definitions and (efficient) implementations are that an implementation:

- is difficult to construct;
- is difficult to adjust to correspond to an evolving algorithm;
- is difficult to adjust to correspond to evolving implementation techniques;
- is difficult to transfer to other computational architectures;
- may be incorrect when first constructed;
- can be difficult to correct;
- is difficult to verify informally;
- is practically impossible to verify formally.

Advances in programming languages may ameliorate some of these problems, but as computer systems and algorithms become more complex, many of the problems are likely to worsen.

The method used in this thesis to overcome the above problems is to automate the implementation process: rather than devote his efforts to developing a single implementation, a programmer develops implementation methods, encoded as transformations and derivations.

- The starting point for automated implementation of an algorithm is a formal definition of the algorithm. A definition should be high-level, easily understood, easily analysed and easily manipulated; it should be a clear statement of the algorithm, probably close to the mathematical definition. Human consideration of the algorithm should normally be constrained to the high-level definition.
- By reducing the implementation process to the automated application of transformations, the programmer is freed from the often messy details of implementations: the minutiae of implementations are managed by the transformation system.

  Automation also provides a simple solution to the problem of keeping an implementation (or a set of implementations) in step with an evolving algorithm: change the high-level definition (which should be a simple task if the specification is designed correctly) and re-apply a derivation.

  Similarly, if an implementation technique changes, change the stage of the derivation corresponding to that technique, or add further stages to the derivation, and re-apply.
- Most transformations are simple enough to be self-evidently correct, and formal analysis of correctness may be possible for more complex transformations. If an implementation is

produced from a definition by the application of correct transformations, the implementation is guaranteed to be correct, by construction.

10.1.1 A Family of Derivations

An important aspect highlighted by the derivational approach is that the implementation process can be structured: there is a sequence (or perhaps several sequences) of intermediate forms that lead from a high-level definition to an efficient implementation. Intermediate forms can correspond to optimization processes on an algorithm definition, to tailoring an implementation for a particular computational architecture or to optimizing an implementation to use special programming techniques when input data is known to have certain properties (such as a matrix being sparse).

Further, many of the intermediate forms are shared by derivations for multiple computational architectures (and for multiple definition languages). This sharing leads to the concept of a family of derivations, as illustrated in figure 10.1. This structure represented in this figure is one of the most important results to emerge from this work.

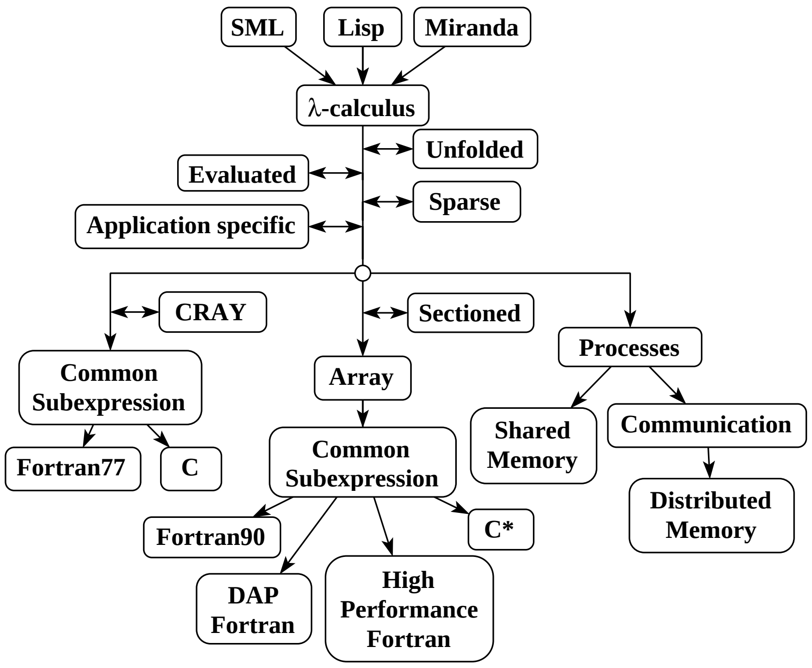


Figure 10.1: A family of derivations

At the top of the tree are possible languages for defining algorithms, functional programming languages. Functional languages permit an algorithm to be defined in a manner similar to mathematical definitions, but provide an entirely formal notation that is often lacking in mathematical definitions. The only aspect of functional languages which may pose a problem for numerical algorithms is the lack of a good array data type; this is overcome by defining my own array ADT.

Functional programming languages often provide features that simplify their use for human programmers; such features range from the purely syntactic (such as infix operators) to more fundamental features such as structuring mechanisms. For various reasons, such features generally complicate automated manipulation of programs, so a functional specification is converted into a simple notation, the λ-calculus form. The rest of the implementation process is entirely independent of the precise specification language.

A programmer should use decomposition techniques to reduce the complexity of a specification. Such structure induced by decomposition often introduces unnecessary overheads into an implementation, so the structure is stripped from a specification by unfolding definitions and evaluating certain forms of expressions. A specification is then in a simple, concise form.

If required, a specification can be tailored to use special programming techniques that take advantage of, say, sparse matrices to reduce execution time and storage. Problem domain information may also be applied to optimize a specification.(i)

Up to this point, the implementation process has been independent of the system that is to execute an implementation; but now the process begins to tailor a specification towards particular architectures.

- A specification can be converted into whole-array form, containing explicitly data-parallel operations suited to execution by an array processor. The specification can then be optimized by combining multiple occurrences of computationally expensive expressions.

  Then an implementation tailored for a specific system, such as the AMT DAP, can be produced. Other possible implementations could be for the emerging array-based Fortran dialects such as Fortran90 or High Performance Fortran.

- An implementation for sequential system can be produced, using standard Fortran or C as the implementation language.

- Multi-processor implementations could be produced by introducing process and communication. (Such implementations have not been considered in this thesis, but have been considered by Boyle.)

The derivations were illustrated using two significant algorithms: POT, for the computation of an approximate eigensystem of a matrix, and Conjugate Gradient, for the computation of the approximate solution to a set of simultaneous equations. The discussion included: each major stage of the derivation of sequential, CRAY and DAP implementations of CG; the tailoring of the sequential, CRAY and DAP implementations to use compact stores for tridiagonal and row-wise sparse matrices; the DAP implementation of POT.

10.2 Assessment

The importance of the work reported here is:

- It adds further support for the tenet that efficient implementations can indeed be automatically derived from high-level, algorithmic specifications. Boyle and Harmer provided a demonstration of feasibility by deriving an implementation of a cellular automata algorithm for the CRAY that out-performed a hand-crafted implementation [22]. The work reported here extends the range of computational architectures.

- It also supports the structuring of the implementation process into sub-derivations by identifying intermediate forms.

- In addition, the tailoring for sparse matrices has extended the scope of derivation in an important new direction. The construction of a sparse implementation is usually undertaken as an entirely separate (and quite complicated) task from the construction of a standard implementation. Viewing sparse programming techniques are merely another implementation decision, to be incorporated automatically if so desired, is a considerable shift in perspective.

- The sub-derivations and transformations developed for this thesis are a strong initial attempt at realizing the ideas discussed in this thesis. The transformations are not perfect, and I am already aware of possible improvement and extensions, but I believe the basic methods on which they are based are sound.

The work reported in this thesis can be assessed in several ways: the suitability of the specification language; the effectiveness of individual sub-derivations; and the effectiveness of derivations as a whole.

10.2.1 The Specification Language

The specification language used in this thesis (a subset of the SML functional programming language) seems well-suited to specifying numerical algorithms (probably algorithms in general, but numerical algorithms are what I have most experience of). I have not been aware of any difficulties in specifying the computational aspects of algorithms arising from the choice of language, and have not received any objections when discussing specifications before audiences.

However, what needs to be addressed is the provision of non-computational information, such as the shapes of arrays and properties of data structures (such as sparsity). It may be most convenient to extend type information to provide such information, but that would probably rule out a statically typed language, and I would prefer not to introduce run-time support for dynamic typing. (Of course, it may be that dynamic typing could be removed by transformations!). This is a subject for further research.

The Array ADT also seems well-suited to the majority of numerical algorithms. Specifically, the `generate` and `reduce` functions seem able to express most array computations succinctly, while still possessing simple semantics. The manipulation performed on array (Static Evaluation, conversion to Array form, introduction of sparse programming techniques) is quite extensive, but reasonably straightforward.

I believe that `generate`, as regards analysis and manipulation, is superior as the basic array constructor than the mapping function that seems to be the usual choice when implementing arrays. The choice of `map` presumably stems from the experience with lists, but lists have an inherently sequential processing order which is inappropriate for arrays.

Where the Array ADT may require extension is for operations in which the elements of an array are manipulated in a particular order. Currently, such ordered operations are defined as recursive functions, in which each iteration of the function manipulates sub-sets of the elements; such definitions may be difficult to parallelize. A more useful form of definition may require some general mechanism for defining an order over sub-sets of the elements of arrays. This is a subject for further research.

10.2.2 The Sub-derivations

SML-to-λ-calculus The conversion from SML to λ-calculus is mostly syntactic, and so is relatively straightforward. The more difficult aspects are removing structures and raising local functions to the top level. The sub-derivation is effective in performing the conversion.

Unfolding The Unfolding sub-derivation was developed by Boyle and Harmer so I will not discuss it in detail. The effectiveness of the sub-derivation was illustrated in this thesis by its reduction of each of the example algorithms into a single function definition! In general, the only factor limiting the effectiveness of the sub-derivation is the use of recursive functions.

Static Evaluation The sub-derivation is very effective at simplifying the structure of specifications. Indeed, one problem to be address is whether it should be as effective as it is: there may be occasions when it would be better to retain structure (where the structure is important for computational reasons, and where the structure cannot easily be recovered once removed). Identification of such occasions is likely to be difficult and is a subject for further research.

Array Form I believe the basic principle for conversion to Array form (the propagation rules) is sound. The transformations do though need to be extended to include further special cases, for expressions such as cyclic shifts of vectors and optimized versions of conditional array expressions. In addition, there are occasions when I believe the degree of parallelism could profitably be increased (to use matrix operations rather than vector operations); further work is required to identify such occasions and establish if they are capable of being handled systematically.

Sparse The sparse sub-derivations are general in the sense that they are based upon identities between data types. However, further optimizations (particularly for simplifying reductions which partly intersect the non-zero region) are required. In addition, I have not considered in great detail how the transformations will fare with algorithms which compute sparse matrices, rather than having sparse matrices as input data. The primary problem is that it may not be possible to always identify when a matrix expression will produce a sparse matrix. (e.g. is it possible to automatically determine the structure of the product of two sparse matrices?)

Common Sub-expression Elimination The CSE sub-derivation is based on general principles and is, I believe, effective at eliminating multiple occurrences of expressions that are considered computationally expensive. There are, however, several issues remaining to be addressed:

- Is the method used to identify expensive computations accurate, or does it require further refinement?
- The effectiveness of CSE is limited by its ability to determine whether two expressions are equivalent. Since this is a generally undecidable problem, CSE must always remain incomplete, but further work could usefully extend the types of expressions for which CSE *can* decide equivalence.
- Commoning is restricted to within a single function. This was not important for the algorithms considered here as unfolding reduced the algorithms to a single function, but it will probably be important for other algorithms.

CRAY Further work is required to establish some useful programming model for the CRAY Fortran77 compiler. Until such a model exists, the CRAY tailoring will always be incomplete.

Fortran77 The conversion to Fortran77 is based heavily on a sub-derivation developed by Boyle and Harmer, so I will not discuss it in detail. The Conjugate Gradient example considered in this thesis does however show that some refinement to the rules for variable reuse is required. (In addition, further work is required on reuse involving partial updating of arrays.)

Fortran Plus Enhanced Similar comments apply as for the Fortran77 sub-derivation.

10.2.3 Derivations

The preceding section discussed the effectiveness of each individual sub-derivation in performing its appointed task. A separate consideration is how effective are full derivations, constructed from the sub-derivations.

Perhaps the most important consideration in assessing a derivation is to remember that I do not aim for complete generality:

- Not all aspects of SML are supported; for example, the supported forms of pattern matching are rather restricted. (I do not wish to support all forms of pattern matching as I view it as merely a notational convenience, but the resolution of some patterns can involve non-trivial computation.)
- There is no support even for some forms that are, strictly speaking, permissible in the subset of SML that I use. For example, I assume that all polymorphic functions are unfolded, so that I do not need to support polymorphism in the Fortran implementation! This is a limitation on the current derivations, but I do not believe it is an important one for the types of algorithms I consider. (If it does turn out to be important, then the derivations can be extended to allow for it.)
- Although arrays can have any element type, the current sub-derivations assume the types are integer, real or boolean. Should an algorithm require arrays of arrays, then an additional stage should be added to recast nested arrays into a form suitable for the implementation language. (This may be viewed either as a limitation of the current derivation, or as an example of the flexibility of the derivational approach!)
- The current derivation for the DAP also assumes that arrays have no more than two dimensions (specifically, the sub-derivation attempts to parallelize all dimensions of an array, but Fortran Plus Enhanced restricts parallel arrays to two dimensions).

Given these restrictions, the derivations do seem to be effective: examination of implementations of Conjugate Gradient reveals only minor flaws, and comparison of POT with an independent, hand-crafted implementation shows virtual parity of execution performance.

One problem that has been encountered in assessing derivations is that often it is difficult to obtain sufficient information about algorithms, their implementation and the implementation process to permit a direct comparison: if I am to compare the execution performance of a derived implementation against that of an independent, manually constructed implementation, then it is important that both implementations are implementations of the same algorithm. Numerous publications report execution performance an implementation of an algorithm, but fail to provide a precise definition of the algorithm or implementation. For this reason, there is often little that can be done to assess an implementation than to examine it for flaws.

10.3 Further Work

Some area for future research were indicated in the previous section (overcoming limitations of the current derivation); of particular interest to me would be supporting arrays with more than two dimensions, as there are several possible approaches: e.g. ‘flattening’ a three dimensional array into two dimensions so that all the elements can be processed in parallel; choosing two of the dimensions for parallel processing and implementing the rest sequentially; if some of the dimensions are short, it may be feasible to reduce the dimensionality of the array by converting it into a set of arrays. (e.g. the FPE array `A(n,n,3)` would become the set of three arrays `A1(n,n)`, `A2(n,n)` and `A3(n,n)`.)

Also of practical use would be support for arrays which have arrays as elements.

Other potential sub-derivations could address the following problems:

- targeting other architectures, particularly multi-processors;
- targeting other implementation languages such as Fortran90;
- dividing arrays larger than the DAP's processor grid into sections the same size as the grid — it would be interesting to compare the performance of implementations sectioned by transformations with that of an implementation sectioned by the FPE compiler;
- optimizing implementations for other types of structured matrices (e.g. other types of sparsity, symmetric matrices);
- converting two-dimensional arrays into one-dimensional arrays — flattening arrays in this manner can permit optimizations by making explicit the computations a compiler introduces when indexing multi-dimensional arrays;
- dynamic allocation of arrays from a global array — this is an important implementation technique that is conceptually simple but technically difficult.

Other areas for further research include:

- developing a suitable programming model for the CRAY;
- formal proofs of correctness of transformations and completeness of derivations.

10.4 Concluding Remarks

At the start of this thesis, I said it was an open question as to what can be achieved through automatic program derivation, as regards producing efficient implementations for high-performance computer systems from high-level specifications.

It remains an open question!

However, I hope that the results presented here provide some encouragement that much useful work can be performed. I have been pleasantly surprised by some of the results: tasks such as converting to Array form and tailoring for sparse matrices have turned out to be vastly more elegant than first anticipated. That sparse tailoring can be automated even to the limited extent presented here was something of a revelation.

What was also a surprise was the emergence of the relationship between derivations, as illustrated in the tree of intermediate forms (figure 10.1). Though it is clearly impossible to tell if the tree will maintain the same basic structure that is has now, I expect that it may turn out to be an important conceptual guide for future work.

In conclusion, it would seem that structuring the implementation process and encoding programming techniques are both feasible and useful. Given the alternative — programmers repeatedly performing more or less the same work over and over — I believe the specification-derivation approach to programming will prove to be an important area of both research and practice.

10.5 Chapter Notes

i (page 161)

The precise point at which problem domain optimization should take place is problematic: optimizations will presumably be based upon certain functions that capture high-level operations in the problem domain, so optimization must take place before function unfolding strips away such functions. However, some unfolding should probably be performed to eliminate structure that is not essential to the optimizations (for example, simple name-value bindings) as removing such structure generally produces opportunities for optimization (especially if optimizations are based on combinations of problem domain functions). It may be that a layering of unfolding is required. This is a subject for future research.

Appendix A

The Intermediate Forms

The main body of the thesis discussed the intermediate forms produced by derivations. This appendix provides some additional technical details for some of the forms. A formal grammar is not used, as I wish to emphasize the ideas involved rather than their syntactic realizations.

The intermediate forms discussed are:

- **SML**
- **λ-Calculus**
- **Typed**
- **Unfolded**
- **Statically Evaluated**
- **Common Sub-Expressions**
- **Fortran77**
- **Array**
- **DAP Fortran**
- **Sparse**

A.1 SML

An SML specification can contain three categories of entity: definitions, expressions and types.

A.1.1 Definitions

A specification is a sequence of:-

- *function-definition* a function definition binds a function expression to a name;
- *value-definition***:** a value definition binds an expression to a name;
- *type-definition***:** a type definition introduces a name that can be used instead of a type expression;
- *structure-definition***:** a structure is one SML method for defining abstract data types;
- *open-statement***:** an open statement allows the types and functions defined in a structure to be accessed as though they were defined at the top level of the definition.

For example, the specification

```
structure S = struct
  type t = int
  fun maket(i:int):t = i
end
open S
val start:int = 3
fun double(x:int):int = 2*x
fun init():t = maket(double(start))
```

defines the following:-

- the structure `S` which is "opened" to allow its definitions to be referenced;
- the type `t`;
- the integer value `start`;
- the functions `maket`, `double` and `init`; the body of `init` is the expression to be calculated by the specification.

Function Definition

A *function-definition* has the form[1]

function-definition `::=` `fun` *simple-name formal-arguments*`:`*type* `=` *expression*.

for example `fun f(a:int)(b:real,c:real):real = e`.

Two forms of identifiers are used in specifications, which I call *simple-name* and *name*. A *simple-name* is the standard sequence of characters/digits; for example, `plus`. A *name* is a sequence of such *simple-names*, separated by periods ('.'); for example, `integers.plus`. A *name* is used to refer to

[1]The SML programming language normally requires names to be defined before they are used, except in special environments designed to allow mutually-recursive definitions. The transformations used in this thesis do not require define-before-use, so I do not consider the special SML environments.

a component of a structure; for example `integers.plus` refers to the function `plus` defined as part of the structure `integers`. All identifiers introduced by SML definitions must be *simple-name*s.

The *formal-arguments* occurring in a function definition have the form

formal-arguments `::=` *sequence of (sequence of simple-name:type)*.

The *type* following the arguments is the 'return' type of the function; that is, it is the type of an application of the function (in which all arguments are supplied).

Value Definition

A *value-definition* has the form

value-definition `::= val` *simple-name*`:`*type* `=` *expression*

for example `val espilon:real = 1E~14` defines the name `epsilon` to be equivalent to `1E~14`.

Type Definition

A type definition can be one of two forms:

type-definition `::=` *standard-type-definition* `|` *parameterized-type-definition*

A standard *type-definition* has the form:

standard-type-definition `::= type` *simple-name* `=` *type-expression*

while a parameterized *type-definition* has the form

parameterized-type-definition `::= type` *type-variable simple-name* `=` *type-expression*

where *type-variable* is defined as *'simple-name* (that is, a *simple-name* preceded by a quote mark).

Structure Definitions

A structure definition has the form

structure-definition `::=`
`structure` *simple-name* `= struct`
sequence of *function-*, *type-* and *value-definition*s and *open-statements*.
`end`

Note that a structure definition cannot contain structure definitions (i.e. nested structures are not allowed), although it should be simple to extend the relevant transformations to allow this.

The components defined in a structure definition may be referenced outside the structure using a *name*, as explained above. Inside the structure, a component may be referenced using a *simple-name*. For example, in the specification

```
structure S = struct
  val f :int= ...
  val g:int = f*2
end
val h:int = S.f*2
```

the two values `g` and `h` are the same.

Open Statements

An *open statement* has the form

`open` *simple-name*

An *open statement* permits all the definitions inside the opened structure to be referenced without using remote reference.

If a structure definition contains an open statement, the components of the structure so opened are considered to form part of the structure being defined. For example, given

```
structure S = struct
  fun f(x:int) = x
end

structure T = struct
  open S
  fun g(y) = y
end
```

then the structure `T` consists of both the functions `f` and `g`. Thus, the definition of `T` is equivalent to

```
structure T = struct
  fun f(x:int) = x
  fun g(y) = y
end
```

A.1.2 Expressions

For simplicity, in the following I ignore issues of precedence amongst operators and the presence of type information after an expression (essentially, any expression except conditional expressions and function expressions can be followed by a colon and a type).

An expression can have any of the following forms:-

name
The name should be a formal argument to a function or a name introduced in a function or value definition.

constant
A literal constant of type real, integer or boolean (the two boolean constants being `true` and `false`).

function-application `:==` *expression actual-arguments*
The *actual-arguments* have the form *sequence of (sequence of expression).* For example

`f(1,2)(3.0+4.0)(true).`

function-prototype `::= fn` *formal-arguments*=>*expression*
Function prototypes allow a function to be defined without having to name the definition; they are equivalent to λ-expressions. For example, `fn(x:int) => 2*x` is equivalent to $\lambda x : int \cdot 2 * x$.

conditional-expression `::= if (`*expression*`) then` *expression* `else` *expression*

prefix-operator-expression `::=` *prefix-operator expression*
The prefix operators are ~ (negation) and `#n` (tuple components), where `n` is `1`,`2`, ...

infix-operator-expression `:==` *expression infix-operator expression*
The infix operators are `+`,`-`,`*`,`/`,`::` and `@` (the final two operators being list prefixing and array indexing).

tuple-expression `::=` `(`*sequence of expression*`)`
For example: `(1,2,3)` is the 3-tuple with components `1`,`2` and `3`.

list-expression `::=` `[`*sequence of expression*`]`
This form of list expression is considered to be an abbreviation for the "cons" form. For example, `[1,2,3]` is equivalent to `1::(2::(3::nil))`.

local-expression `::=` `let` *sequence of function and value definitions* `in` *expression* `end`
The function and value definitions in a *local-expression* can only be referenced in the local expression. The value of a local expression is the value of the body expression (the expression following the `in`).

parenthetical-expression `::=` `(`*expression*`)`

A.1.3 Type Expressions

A type expression can have any of the following forms:-

basic-type `::= int | real | bool | list | array`

name
The *name* must be a *name* introduced by a type definition.

product-type `::=` *type*`*`*type*
A tuple expression has a product type; for example, the tuple `(1,2.0,true)` has type `int*real*bool`.

function-type `::=` *type*`->`*type*
Not surprisingly, functions have *function-type*s. For example, the function `fn(x:int)=>1.0` has type `int->real`.

parameterized-type `::=` *type-expression name*
Again, the *name* must be introduced in a type definition (`array` and `list` are taken to be predefined). For example, `int pair`.

parenthetical-type `::=` `(`*type*`)`

A.2 λ-Calculus

The λ-calculus form used for this thesis is the standard λ-calculus extended with named function, value and type definitions and a set of primitive functions (for integers, reals, booleans and arrays).

As with the SML form, there are three categories of entities in the λ-calculus form: definitions, expressions and types. These are discussed below: where a syntactic class is not elaborated, the class is the same as defined for the SML form.

A.2.1 Definitions

A specification in the λ-calculus form is a sequence of:

function-definition `::= fun` *name*`:`*type* `=` *expression*
Note the differences from the SML form of function definition: there are no formal arguments (on the left side of the definition) and the name introduced may be a *name*, not just a *simple-name*.

value-definition

type-definition

There are no structure definitions or open statements; all definitions occur at the top level of a specification.

A.2.2 Expressions

The various forms an expression may take in the λ-calculus form are listed below.

name

constant

function-application

λ-expression `::=` λ*formal-arguments*$\cdot$*expression*

conditional-expression `::= use` *expression* `if (`*expression*`) otherwise` *expression* `end`
This form of conditional expression is used in the λ-calculus form for purely technical reasons. (This form of conditional allows the entire conditional expression to have a type appended, after the `end`.)

list-expression
Lists are allowed as they are convenient for denoting shapes and indices.

Note in particular that the λ-calculus form does not permit operator, local or tuple expressions.

A.2.3 Type Expressions

Type expressions are as described for SML.

A.3 Array Form

The Array form is based on the λ-calculus form, but extends the set of primitive functions to include data-parallel functions. These functions are defined below.

Mappings

All elementwise functions are expressed using the `array.map` and `array.map2` functions:

```
array.map(A:α array,f:α->α):α array
 ≡ generate(shape(S),λi:index·f(A@i))

array.map2(A:α array,B:α array,f:α*α->α):α array
 ≡ generate(shape(A),λi:index·f(A@i,B@i))
```

where

- `A` and `B` have the same shape;
- α is `int`, `real` or `bool`;
- `f` is one of `plus`, `times`, `difference`, `quotient`, `negative`, `abs`, `lessp`, `greaterp`, `lesseqp` or `greatereqp` for ineger and real; `ln`, `exp` or `sqrt` for real only; `and`, `or` or `not` for boolean only; `eq` for integer, real and boolean.

Elementwise Reductions, including Partial Reductions

Elementwise reductions are expressed using the `array.fold` function:

```
array.fold(A:α array,r:α*α,init:α):α
 ≡ reduce(shape(A),λi:index·A@i,r,init)

array.fold.rows(A:α array,r:α*α,init:α):α array
 ≡ generate([size(A,0)],
    λ[i:int]·reduce([size(A,1)],λ[j:int]·A@[i,j],r,init))

array.fold.cols(A:α array,r:α*α,init:α):α array
 ≡ generate([size(A,1)],
    λ[j:int]·reduce([size(A,0)],λ[i:int]·A@[i,j],r,init))
```

where `r` is one of `plus`, `times`, `greaterp` or `lessp` for integer or real; `and`, `or` for boolean.

Conditionals

The general data parallel conditional function is `join`:

```
join(M:bool array,A:α array,B:α array):α array
 ≡ generate(shape(M),λi·if (M@i) then A@i else B@i)
```

where `M`, `A` and `B` have the same shape.

An optimized version ‘updates’ a single element of a vector:

```
vector.updateel(V:α array,i:int,e:α):α array
 ≡ generate(shape(V),λ[j:int]·if (j=i) then e else V@[i])
```

Extractions

The three extraction functions return a specified row or column of a matrix, or the diagonal.

```
row(A:α array,i:int):α array
  ≡ generate([size(A,1)],λ[j]·A@[i,j])
column(A:α array,j:int):α array
  ≡ generate([size(A,0)],λ[i]·A@[i,j])
diagonal(A:α array):α array
  ≡ generate([size(A,1)],λ[i]·A@[i,i])
```

Transpose

Matrix transpose is performed using the `transpose` function:

```
transpose(A:α array):α array
 ≡ generate([size(A,1),size(A,0)],λ[i,j]·A@[j,i])
```

Expansions

Scalar expansion and expansion of a vector into a matrix are defined as:

```
expand(S:shape,e:α):α array
 ≡ generate(S,λi:index·e)

expand.rows(n:int,V:α array):α array
 ≡ generate([n,size(V)],λ[i,j]·V@[j])

expand.cols(n:int,V:α array):α array
 ≡ generate([size(V),n],λ[i,j]·V@[i])
```

Shifts

Only planar shifts of vectors and matrices are considered, in which a null value is introduced along one edge. In the following, `null` is 0 for integer, 0.0 for real, and `false` for boolean.

```
vector.shift.left(V:α array):α array
 ≡ generate(shape(V),λ[i:int]·
   if (i=size(V)) then null else V@[i+1])

vector.shift.right(V:α array):α array
 ≡ generate(shape(V),λ[i:int]·
   if (i=1) then null else V@[i-1])
```

```
matrix.shift.west(A:α array):α array
 ≡ generate(shape(A),λ[i,j]·
   if (j=size(A,1)) then null else A@[i,j+1])

matrix.shift.east(A:α array):α array
 ≡ generate(shape(A),λ[i,j]·
   if (j=1) then null else A@[i,j-1])

matrix.shift.north(A:α array):α array
 ≡ generate(shape(A),λ[i,j]·
   if (i=size(A,0)) then null else A@[i+1,j])

matrix.shift.south(A:α array):α array
 ≡ generate(shape(A),λ[i,j]·
   if (i=1) then null else A@[i-1,j])
```

Appendix B

The Sub-derivations

In this chapter, I discuss in detail each of the sub-derivations developed for this thesis:

- **SML–to–λ-Calculus**
- **Static Evaluation**
- **Common Sub-expression Elimination**
- **Array**
- **Sparse**

The Unfolding sub-derivation and the sub-derivations to convert to Fortran77 or Fortran Plus Enhanced are based heavily on sub-derivations developed by Boyle and Harmer, so they are not discussed here.

B.1 SML–to–λ-calculus

Comparison of the SML form (section A.1) and λ-calculus form (section A.2) reveals the following differences:

- **Definitions**
  - No structures or open statements in the λ-calculus.
  - Names introduced in λ-calculus definitions can be *remote-name*s.
  - Function definitions have no formal arguments in the λ-calculus.
- **Expressions**
  The λ-calculus allows λ-abstractions but not function prototypes, prefix- or infix-operator expressions, or local expressions.

The task of the SML to λ-calculus derivation is to convert those types of expressions and definitions present in the SML form but absent in the λ-calculus into equivalents in the λ-calculus, as described below.

B.1.1 Definitions

Structures

Structures are used in specifications merely to enhance modularity, and can be eliminated by 'raising' all definitions occurring inside structures to the 'top-level'. The main difficulty in such raising is ensuring that all names are unique while maintaining all references to definitions. (That is, ensuring that a reference is to the same definition both before and after raising, even though the name used to refer to the definition has changed.)

If all references to a definiton use remote names, the definitions inside a structure can be raised by prefixing the structure name to the definition name. For example, the structure

```
structure S = struct
type t = int;
val ident = 1;
fun f = e;
end
```

is equivalent to the top-level definitions

```
type S.t = int;
val S.ident = 1;
fun S.f = e;
```

.

Since not all references to a definition use remote naming, two extensions are required for when (i) the reference occurs inside the same structure as the definition; (ii) the reference follows a statement that opens the structure.

The first case can be allowed for by converting all references inside a structure to remote references (by prefixing the structure name to each reference). The second case can be allowed for by replacing the open statement with a set of definitions (which duplicate the names occurring inside the structure) which explicitly refer to the definitions inside the structure using remote naming. For example, the structure definition and open statement

```
structure S = struct
fun f = e;
end;
open S;
```

is equivalent to

```
structure S = struct
fun f = e;
end;
fun f = S.f;
```

.

All references following the open statement now refer to the top level definition, which in turn refers to the definition inside the structure.[1]

Function Definitions

The λ-calculus form does not allow formal arguments on the left side of function definitions. Any such formal arguments in an SML function definition are moved to the right side of the definition as bound identifiers of a λ-abstraction. For example,

```
.sd.
  fun f(i,j)(r)= e ==> fun f = λi,j·λr·e
.sc.
```

B.1.2 Expressions

Applications of infix- and prefix-operators can be eliminated by converting to prefix function applications. For example, `a+b` can be converted into `plus(a,b)`. As most of the operators are overloaded, the names chosen for the prefix functions are generic (they do not imply any particular data type such as integer or real), for example, `op.plus` and `op.times`.

Function prototypes are converted into λ-abstractions; for example:

```
.sd.
  fn(i:int)=>e  ==>  λi:int·e
.sc.
```

A local expression which introduces a value binding is equivalent to a closed λ-expression:[2]

```
.sd.
  let val i:int = e in B end
  ==>
  λi:int·B (e)
.sc.
```

If a local expression introduces multiple bindings, the bindings are first nested then each is converted into a λ-expression.

If a local expression introduces a function definition, the function definition is separated as a new top-level function. For example

```
fun f
  = λx·
    let fun g = λy·G
    in F
    end
```

becomes

```
fun g = λy·G
fun f = λx·F.
```

If the body `G` of the local function depends upon the formal arguments of `f`, then additional formal arguments are added to `g`, and the appropriate values appended to each application of `g` in `f`.

[1] This approach can lead to a large increase in the size of a specification due to the number of definitions being formed. An alternative approach which would avoid the increase in size would be to trace references back through open statements and structures.

[2] The transformations for converting local expressions where written by Harmer.

B.2 Static Evaluation

The forms of static evaluation used for this thesis can be categorized as:

- **Algebraic identities of data types.**
- **β-reduction.**
- **Propagation of accessor functions.**
- **Conditional expressions.**

B.2.1 Algebraic Properties of Data Types

Most data types have a range of simple algebraic identities (often the defining axioms of the data type). For example, the integers have the identities

$$i + 0 \equiv i \quad \text{and} \quad i - i \equiv 0$$

and lists have

$$head(cons(a,l)) \equiv a$$

and

$$length(nil) \equiv 0, \quad length(cons(a,l)) \equiv 1 + length(l).$$

Such identities can be used as transformations to evaluate expressions. For example, the two integer identities give the transformations

```
.sd. integer.plus(e,0) ==> e .sc.
.sd. integer.difference(e,e) ==> 0 .sc.
```

A comprehensive list of transformations based on algebraic identities is given at the end of this section.

B.2.2 β-Reduction

Consider an expression such as

λ`x·integer.plus(x,y) (0)`.

While this expression performs a calculation that is equivalent to the addition of zero, the expression would not be evaluated by the algebra-based set of transformations as the addition of zero is not *manifest*: the addition involves `x` and `y`, not 0.

However, by β-reducing the λ-expression, the addition of zero is manifested, and then removed:

```
λi·integers.plus(x,y) (0)
==>
integers.plus(0,y)
==>
y.
```

Thus, to increase the applicability of the transformations based on algebraic properties, the Static Evaluation derivation β-reduces most λ-expressions. β-reduction itself may result in an *increase* in the operational complexity of an implementation, if the bound identifier occurs more than once in the body of the λ-expression. However, the effects of further Static Evaluation transformations normally outweigh any such increase, and a later sub-derivation can λ-abstract multiple occurrences of an expression.

There are two cases where β-reduction is not performed: the bound value contains an application of the function `iterate`, or an application of a program defined function. In such cases, the increase in operational complexity caused by β-reduction is likely to be enormous (since, after function unfolding, such expressions normally indicate major sections of the computation) and few opportunities exist for further evaluation of such expressions.

B.2.3 Propagation of Accessor Functions

Consider an expression such as

```
head(use cons(a,l) if (p) otherwise cons(b,l) end).
```

Just as with λ-expressions, the transformations based on algebraic properties of the list data type cannot simplify this expression because it does not contain a manifest occurrence of one of the the patterns. However, propagating the `head` function into the conditional gives

```
use head(cons(a,l)) if (p) otherwise head(cons(b,l)) end
```

which *can* then be simplified

```
use a if (p) otherwise b end.
```

The `head` function is an example of an *accessor* function, that is a function that accesses a component of a data structure. (Other examples are the functions `tail`, `element` and `pi`.) Transformations are applied to propagate accessor functions though conditional expressions.

B.2.4 Conditional Expressions

There are several forms of evaluation involving conditional expressions. The simplest are:

- The predicate is a literal `true` or `false`: the conditional is simplified into the appropriate limb.

```
.sd. use T if (true) otherwise F end ==> T .sc.
.sd. use T if (false) otherwise F end ==> F .sc.
```

- The limbs of the conditional are the same expression, so the conditional as a whole is equivalent to that expression:

```
.sd. use T if (p) otherwise T end ==> T .sc.
```

Contextual Evaluation

Probably the most important form of evaluation of conditional expressions is *contextual evaluation*. That is, for a conditional expression

```
use T if (p) then F end
```

the true limb `T` is evaluated in the context of the predicate `p` being true, and the false limb `F` in the context of `p` being false.

For example, in the expression

```
use plus(1,use 1 if (eq(n,0)) otherwise n end)
if (eq(n,0))
otherwise plus(2,use 1 if (eq(n,0)) otherwise n end)
end
```

the first of the inner occurrences of the expression `eq(n,0)` can be replaced with `true` while the second of the inner occurrences can be replaced with `false`:

```
use plus(1, use 1 if (true) otherwise n end)
if (eq(n,0))
otherwise plus(2, use 1 if (false) otherwise n end)
end
```

which can then be further evaluated.

It may seem unlikely that such expressions would occur frequently. However, we have found that function unfolding often results in just this sort of nested repetition of conditions. (Presumably the modularity of the original function definitions forces many functions to perform the same test on its arguments.)

Conditional evaluation is performed by the transformation

```
.sd.
  use T if (p) otherwise F end
  ==>
  use T[true/p] if (p) otherwise F[false/p] end
.sc.
```

This transformation is very simple, but also very effective. Additional transformations could be used — but are not used for the work reported here — to perform other types of contextual evaluation. For example, if the predicate is a conjunction, then each component of the conjunction is known to be true in the true limb:

```
.sd.
  use T if (and(p)(q)) otherwise F end
  ==>
  use T[true/p,true/q] if (p) otherwise F end
.sc.
```

B.2.5 Transformations Based on ADT Algebra

Here, for completeness, I briefly list the transformations that are based on the algebraic properties of ADTs. Most of these transformations are simple, so I do not discuss them.

Arrays

The `generate` function for arrays is defined by the following two identities:

$$shape(generate(S, f)) \equiv S$$

$$element(generate(S, \lambda i \cdot E), I) \equiv \lambda i \cdot E(I).$$

Transformations follow immediately from these identities.

```
.sc.
  shape(generate(S,λi·f)) ==> S
.sc.
.sd.
  element(generate(S,λi·f),i′) ==> λi·f (i′)
.sc.
```

Tuples

The basic identity for tuples was stated earlier:

$$pi(n, tuple(e_1, \ldots, e_n \ldots)) \equiv e_n.$$

The transformations to implement this identity are

```
.sd. pi(1,tuple(e ?)) ==> e .sc.
.sd. pi(i,tuple(e1 ?)) ==> pi(i′,tuple(?)) .sc.
```

where `i` is an literal integer constant and where `i′` is the integer constant one less than `i`.[3]

Lists

The specifications reported in this work do not use lists as a main data type. However, the array ADT uses lists to represent indices of arbitrary dimension. By applying transformations based on list identities, most of the index manipulation can be eliminated from a specification.

```
.sd. head(cons(e,l)) ==> e .sc.
.sd. tail(cons(e,l)) ==> l .sc.
```

The `nth` function is handled similarly to the tuple projection function.

```
.sd. isempty(nil) ==> true .sc.
.sd. isempty(cons(e,l)) ==> false .sc.
```

```
.sd. map(cons(e,l),λx·E) ==> cons(λx·E (e),map(l,λx·E)) .sc.
.sd. map(nil,λx·E) ==> nil .sc.
```

Scalars

Transformations for integers, reals and booleans follow from basic mathematical properties.

Integer and Real There is little difference between the set of transformations for integers and the set for reals, so I list them together here.

```
.sd. eq(i,i) ==> true .sc.
.sd. eq(i,j) ==> false .sc.
```
where `i`, `j` are literal integer or real constants

[3]The transformation system does not provide for arithmetic, so `i′` is found by applying the transformations
`.sd. 9==>8 .sc.`, `.sd. 8==>7 .sc.`, …, `.sd. 2==>1 .sc.`
(only one of these transformations is applied). These transformations can be extended arbitrarily for tuples with more than 9 components (if ever such a beast occurs).

The second of these transformations is correct because `i` and `j` must be different when it is applied (if they were the same, the first transformation would have been applied).

```
.sd. plus(e,0)        ==> e.sc..sd. plus(0,e)       ==> e           .sc.
.sd. difference(e,0) ==> e.sc..sd. difference(0,e) ==> negative(e) .sc.
.sd. difference(e,e) ==> 0.sc.
.sd. times(e,0)      ==> 0.sc..sd. times(0,e)      ==> 0           .sc.
.sd. times(e,1)      ==> e.sc..sd. times(1,e)      ==> e           .sc.
.sd. quotient(0,e)   ==> 0.sc.
.sd. quotient(e,e)   ==> 1.sc.
.sd. quotient(e,1)   ==> e.sc.
.sd. negative(0)     ==> 0.sc.
                 .sd. negative(negative(e)) ==> e .sc.
```

Boolean

```
.sd. and(true,e)  ==> e     .sc. .sd. and(e,true)  ==> e     .sc.
.sd. and(false,e) ==> false .sc. .sd. and(e,false) ==> false .sc.
.sd. or(true,e)   ==> true  .sc. .sd. or(e,true)   ==> true  .sc.
.sd. or(false,e)  ==> e     .sc. .sd. or(e,false)  ==> e     .sc.
.sd. not(false)   ==> true  .sc. .sd. not(true)    ==> false .sc.
```

B.3 Common Sub-Expression Elimination

Consider an expression such as `f(e,e)`, where `e` is some computationally expensive expression. A straightforward implementation of this expression (such as created by the final stages of derivations) would result in `e` being evaluated twice. A more efficient implementation would evaluate `e` once, assign the value to a variable, and refer to the variable twice. In functional terms, this more efficient implementation corresponds to the expression λv·[f(v,v)](e).

The λ-abstraction of multiple occurrences of an expression is known as *common sub-expression elimination*, or CSE. Many compilers for imperative languages perform CSE, but imperative languages present many difficulties (mainly side effects) that complicate CSE to such an extent that the effectiveness of compilers in this area must be questioned. Certainly, it is to be expected that CSE can be performed much more easily in a pure, referentially transparent context.

The technique used to perform CSE was discussed in section 5.9; here, I discuss details of the sub-derivation.

B.3.1 Method

The CSE sub-derivation has the following main steps:

- **Introduction of if-levels**
- **Identification of computationally inexpensive expressions**
- **Standardization of loop notation**
- **Common sub-expression elimination**
- **Clean up**

Introduction of If-Levels

All expressions have an if-level attached (as part of their type information). The if-levels are calculated by first setting them all to zero

```
.sd. e ==> e:0 .sc.
```

then ensuring that each expression in the limbs of a conditional has an if-level one more than than the conditional's if-level.

```
.sd.
  if (p) then T else F end:i
  ==>
  if (p)
  then T { e:i′ ==> e:(i+1) }
  else F {.sd. e:i′ ==> e:(i+1) .sc. }
  end:i
.sc.
```

Identification of Computationally Inexpensive Expressions

The method used to 'mark' inexpensive expressions is to remove the if-level; since the transformations that perform the abstraction explicitly check if-levels, any expression without an if-level will not be abstracted.

For example:

```
.sd. c:i ==> c .sc.
where c is a literal constant
.sd. x:i ==> x .sc.
where x is an identifier.
```

The other transformations are simple and are not listed.

Standardization of Loop Notation

Functions such as `generate` and `reduce`, which involve the evaluation of some expression for some set of values, are put into a common notation

```
.sd. generate(S,λi·g) ==> loop(generate,S,λi·g) .sc.
.sd. reduce(S,λi·g,r,init) ==> loop(reduce(r,init),S,λi·g) .sc.
```

which highlights the generating function and the shape it is to be applied over. This allows transformations, where appropriate, to consider only the shape and generating function, regardless of whether they form part of a generation or a reduction. This notational change is reversed later.

Common Sub-Expression Elimination

The transformation to perform CSE is

```
.sd.
  E { ? e:i ? e:i ? }:i
  ==>
  λv·E[v/e:i]:i (e:i)
.sc.
```

That is, if an expression contains at least two instances of some sub-expression (at the same if-level as the enclosing expression), the sub-expression is bound to a name and all instances of the sub-expression replaced with the name.

In the preceding discussion, I stated that abstraction was always performed at the lowest possible level; this is guaranteed by the TAMPR's standard application order.

The above transformation is applied together with the transformation to lower if-levels:

```
.sd.
  if (p) then T { ? e:i′ ? } else F { ? e:i′ ? } end:i
  ==>
  if (p) then T[e:i/e:i′] else F[e:i/e:i′] end:i
  where i′ is i+1
.sc.
```

That is, if both limbs of a conditional expression contain the same sub-expression (at an if-level one greater than that of the conditional expression itself), the if-level of all instances of the sub-expression is lowered by one.

Expressions which are denotationally equivalent, but syntactically different, are made syntactically identical by the following transformation.

```
.sd. E { ? e1 ? e2 ? } ==> E { ? e1 ? e1 ? } .sc.
where compare(e1,e2) = true.
```

The 'function' `compare` in this transformation is evaluated by a set of sub-transformations based on the structure of the expressions. For example:

```
.sd.
  compare(e,e) ==> true
.sc.
.sd.
  compare(if (p1) then T1 else F1,if (p2) then T2 else F2)
  ==>
  compare(p1, p2) ∧ compare(T1,T2) ∧ compare(F1,F2)
.sc.
.sd.
  compare(λx·b (e),E)
  ==>
  compare(b[e/x],E)
.sc.
.sd.
  compare(λx1·b1,λx2·b2)
  ==>
  compare(λx1·b1,λx1·b2[x1/x2])
.sc.
```

Sub-expressions shared by two generating functions are extracted as follows:

```
.sc.
  E
  {
    ?
    loop(f1,S,λi·g1 {? e1 ?})
    ?
    loop(f2,S,λj·g2 {? e2 ?})
    ?
  }
  ==>
  E
  {
    ?
    loop(f1,S,λi·g1 {? element(loop(generate,S,λk·e1[k/i]),[i]) ?})
    ?
    loop(f2,S,λj·g2 {? element(loop(generate,S,λk·e1[k/i]),[j]) ?})
    ?
  }
  where compare(e1,e2[i/j]) = true.
.sd.
```

The newly created arrays may then be abstracted.

Clean Up

The clean up step removes if-levels and restores generations and reductions to their original notation. The transformations are simple and are not listed.

B.4 Array

The method employed by the Array sub-derivation to convert from generations and reductions into applications of data-parallel functions was discussed in section 5.8; here, I list the more important transformations for this conversion.

Elementwise functions

If the outermost function in a generating function has an elementwise equivalent, then the generation becomes an application of `map` or `map2`:

```
.sd.
  generate(S,λi·f(e))
  ==>
  map(generate(S,λi·e),f)
.sc.
.sd.
  generate(S,λi·f(e1,e2))
  ==>
  map2(generate(S,λi·e1),generate(S,λi·e2),f)
.sc.
where f has an elementwise equivalent.
```

Generating function is independent of the generating index

If the generating function is independent of the generating index, the generation is equivalent to an application of the function `expand`.

```
.sd. generate(S,λi·e) ==> expand(S,e) .sc.
where e is independent of i.
```

For arrays with more than one dimension, the generating function may be independent of some components of the index. Such components can be separated out for expansion. The functions `expand.rows` and `expand.cols` perform two important examples of such expansions, expanding a vector row-wise or column-wise into a matrix.

Patterns

If the generating function is characteristic of a pattern, the generation is converted into that pattern. For example, the matrix pattern with true values only on the leading diagonal is characterized by the equation i=j (it is also characterized by the shape of the matrix being square):

```
.sd.
  generate(shape(e,e),λ[i,j]·integers.eq(i,j))
  ==>
  pattern.diagonal(S)
.sc.
.sd.
  generate(shape(e,e),λ[i,j]·integers.eq(j,i))
  ==>
  pattern.diagonal(S)
.sc.
```

Other common matrix patterns can be converted in a similar manner.

Row and column

The basic pattern for a row of a matrix is:

```
generate([n])(λj·element(e)([r,j]))
```
where `e` has width n and `r` is independent of `j`.

However, this pattern can be generalized somewhat by replacing `j` in the index with any function of `j`:

```
.sd.
  generate([n])(λj·element(e)([r,J]))
  ==>
  generate([n])(λj·element(row(e)(r))([J]))
.sc.
```
where `e` has dimensions $[m, n]$, `r` is independent of `j` and `J` is a function of `j`.

Diagonal

A vector generation in which the generating function accesses the diagonal elements of a square matrix corresponds to the extraction of the diagonal.

```
.sd.
  generate([n])(λ[i]·element(e,[i,i]))
  ==>
  diagonal(generate([n,n],λ[i,j]·element(e,[i,j])))
.sc.
```

Identity

The identity generation of an array of shape `S` is the generation over `S` in which the generating function returns element `i` of the array for index `i`:

```
.sd. generate(S,λi·element(e,i)) ==> e .sc.
```
where `e` has shape `S`.

Transpose

For `transpose`, the components of the shape are exchanged, and the body is an application of `element` with an index in which the components of the generating index are exchanged.

```
.sd.
  generate([m,n],λ[i,j]·element(e,[j,i]))
  ==>
  matrix.transpose(e)
  where e has shape [n,m]
.sc.
```

Shifts

There are two versions of array shifting in common use: cyclic and planar. In a cyclic shift, an element at an edge is wrapped around to the opposite edge; in a planar shift, an element at an edge is lost, and a null value (zero or false) inserted at the opposite edge. The current version of the derivation recognizes only planar shifts: cyclic shifts are not required for the algorithms used in this thesis, and have considerably more complex generating functions than planar shifts.

The characteristic generating function for shifts has one component of an index offset by a constant amount: here it is assumed the offset is always one, though it would be simple to extend to arbitrary constants.

For example:

```
.sd.
  generate(S,λi·element(e,[integers.difference(I,1),J]))
  ==>
  matrix.shiftnorth(generate(S,λi·element(e,[I,J])))
.sc.
```

λ-expressions

If the body of the generating function is a (closed) λ-expression in which the bound value is a scalar, the generation can be parallelized by replacing the bound value with an array of bound values and replacing instances of the bound identifier in the body with the appropriate element of the bound array:

```
.sd.
  generate(S,λi·λx·e (v))
  ==>
  λX·generate(S,λi·e [element(X,i)/x]) (generate(S,λi·v))
.sc.
```

As a special case, if the bound value is independent of the generating index, the closed λ-expression can be moved outside the generation without expansion:

```
.sd.
  generate(S,λi·λx·e (v))
  ==>
  λx·generate(S,λi·e) (v)
.sc.
where v is independent of i.
```

Conditionals

General conditional expressions are converted into applications of join by propagating the generation into the limbs and the predicate:

```
.sd.
  generate(S,λi·use T if (p) otherwise F end)
  ==>
  join(generate(S,λi·T),generate(S,λi·p),generate(S,λi·F))
.sc.
```

There are several optimizations to this general technique. If the predicate is independent of the generating index, then the conditional can be moved outside the generation:

```
.sd.
  generate(S,λi·use T if (p) otherwise F end)
  ==>
  use generate(S,λi·T) if (p) otherwise generate(S,λi·F) end
.sc.
```
where `p` is independent of `i`.

If the predicate corresponds to the update of a single element of an vector, the generation is converted into an application of an update function. Note that only one scalar value is calculated for the true limb (for the single element that is to be updated); this value is the expression for the true limb instantiated for the index of the element that is to be updated.

```
.sd.
  generate(S,λ[i]·use T if (integers.eq(i,I)) otherwise F end)
  ==>
  vector.update.element(generate(S,λ[i]·F),I,T[i/I])
.sc.
```
where `I` is independent of `i`.

A similar rule applies for a single element, row or column of a matrix. Extending this rule to higher dimensions would be cumbersome (and is not performed for this work) as the predicate must have the form

```
and(integers.eq(i1,I1),and(integers.eq(i2,I2),
  ... integers.eq(in,In) … ))
```

in which each component of the index is separately tested against an integer value, in any order.

B.4.1 Converting Reductions

The Array form defines some common, simple reductions (particularly: summation, conjunction and disjunction). If a reduction has a pattern that corresponds to one of these reductions, the reduction is converted into the Array form equivalent.

For example, the transformation for summation of a real vector is

```
.sd.
  reduce(S,λ[i]·element(U,[i]),real.plus,0.0)
  ==>
  sum(U)
.sc.
```
where `U` has shape `S`.

The most important extension to these conversions is when the generating function of the reduction is not simply an application of `element`; for example

```
reduce(S,λ[i]·real.times(element(U,[i]),element(V,[i])),real.plus,0.0)
```

in which the generating function is the product of the elements of two vectors. Such expressions are converted by extracting the generating function as a generation:

```
==> reduce(S,
  λ[i]·element(
    generate(S,λ[i′]·real.times(element(U,[i′]),element(V,[i′]))),
    [i]),
  real.plus,0.0).
```

The reduction can now be converted while the generation is processed by other transformations.

Partial Reductions

The Array form row and column versions of reductions which reduce a matrix along only one direction (e.g. to form a vector in which each element is the sum of the elements in the corresponding row of the matrix). Such partial reductions occur as generations in which the generating function is a reduction over one dimension.

For example:

```
.sd.
  generate([n],
    λi·reduce([n],λ[j]·element(A,[i,j]),real.plus,0.0))
  ==>
  sum.columns(generate([n,n],λ[i,j]·element(A,[i,j])))
.sc.
```

B.5 Row-wise Sparse

In this section, I discuss modifying the tridiagonal sub-derivation for row-wise sparse matrices. Sparse matrices in general, the tridiagonal sub-derivation and the results of applying the row-wise sub-derivation to the Conjugate Gradient algorithm were discussed in chapter 8.

B.5.1 Worked Example

As with the tridiagonal sub-derivation (8.3, I shall first trace the derivation of an optimized form of matrix-vector multiplication, then provide details of the transformations. Most of the derivation remains unchanged for row-wise sparsity: row-wise sparsity follows the same three steps of:

distinguishing the zeros from the non-zeros;

simplifying the resulting expression;

mapping onto primary store, which, for row-wise sparsity, involves the secondary store.

Distinguishing

This step is the same as for the tridiagonal example.

```
generate([n],λ[i]·reduce([n],λ[j]·times(
  if ([i,j] ∈ rowwise([n,n], w)) then element(A,[i,j]) else 0.0,
  element(V,[j])),
  +, 0.0)
```

Simplifying

Most of this step is same as for the tridiagonal example: in particular, all the transformations (based on the Static Evaluation transformation) for simplifying expression involving zero are the same.

Propagating the application of `times` through the conditional and simplifying the resulting multiplication by zero gives:

```
==> generate([n],λ[i]·reduce([n],λ[j]·
  if ([i,j] ∈ rowwise([n,n], w))
  then times(element(A,[i,j]), element(V,[j]))
  else 0.0,
  +, 0.0)).
```

At this point, the derivation diverges from that in the tridiagonal example. As in that example, the predicate in the reduction's generating function distinguishes row i of the non-zero region. However, in this example, the number of non-zeros on that row is unknown (it is specified as a parameter to the program), so the non-zeros cannot be explicitly delineated as they were in the tridiagonal example. Instead, the function `row` is used to indicate a row of the non-zero region, allowing the reduction to be split into a reduction over that row, and a reduction over zeros, the latter being then simplified.

```
==> generate([n],λ[i]·reduce(row(rowwise([n,n],w),i),
  λ[j]·times(element(A,[i,j]),element(V,[j])),
  +, 0.0)).
```

Mapping

This stage is similar in principle to the mapping stage of the tridiagonal sub-derivation, but of course differs in the mappings applied:

shapes: $rowwise([n,m],w) \rightarrow [n,w]$;

indices $[i,j] \rightarrow [i, search([i,j])]$ and the inverse $[i,j] \rightarrow [i, lookup([i,j])]$

where the function $search$ performs a search on the secondary store to locate the specified sparse index, and $lookup$ uses the secondary store the translate the specified primary index into a sparse column index (implemented as a simple indexing operation on the secondary store).

Applying these mappings to the simplified expression for matrix-vector multiplication gives:

```
==> generate([n],λ[i]·reduce([w],λ[j′]·
  λj·times(
    element(A:[n,w],[i,search(A,[i,j])]),
    element(V,[j])
   )
   (lookup(A,[i,j′])),
  +, 0.0)).
```

The use of `search` in this expression would incur heavy computational expenses in an implementation, and almost certainly prohibit the derivation of an efficient vector or array form. Fortunately, its use can be eliminated, first by β-reduction

```
==> generate([n],λ[i]·reduce([w],λ[j′]·
  times(
    element(A:[n,w],[i,search(A,[i,lookup(A,[i,j′])])]),
    element(V,[lookup(A,[i,j′])])),
  +,0.0))
```

then by using the identity

$$search([i, lookup([i, j])])\equiv j$$

to give

```
==> generate([n],λ[i]·reduce([w],λ[j′]·
  times(element(A:[n,w],[i,j′]),element(V,lookup([i,j′]))),
  +, 0.0))
```

This is the optimized form of matrix-vector multiplication: each element of the product vector is formed by matching the w elements of a row of the primary store, which are located at indices 1 to w, with the appropriate w elements dispersed through the vector V, performing the w multiplications and $w-1$ additions. This process is illustrated in figure B.1.

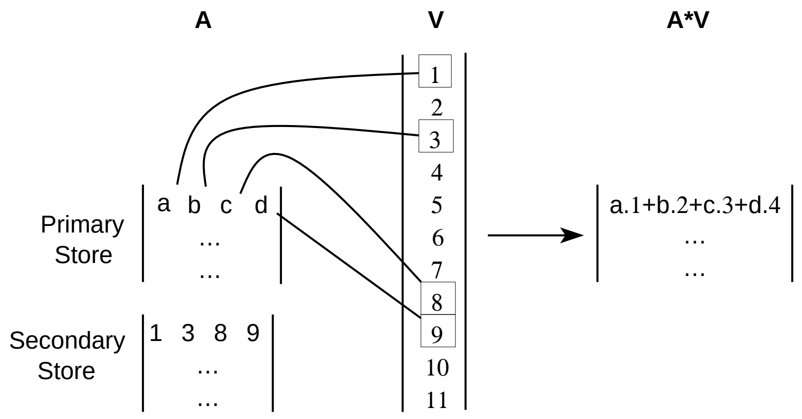


Figure B.1: Multiplication of a row-wise sparse matrix and a vector

B.5.2 Method

As I have already indicated, most of the transformations for the row-wise sub-derivation are the same as for the tridiagonal sub-derivation, so here I shall list only those transformations peculiar to the row-wise sub-derivation.

Splitting Reductions

The storage scheme for row-wise sparsity allows a row of the sparse matrix to be readily located in the primary store, so it is useful to split off a reduction over a row of non-zeros:

```
.sd.
  reduce([n],
    λ[j]·if ([i,j] ∈ tridiagonal(n)) then T else F,
    λx,y·r,init)
  ==>
  λx,y·r
    (reduce(row(tridiagonal(n)),λj·T,λx,y·r,r0),
     reduce([n]-row(tridiagonal(n)),λj·T,λx,y·r,init))
   where i is independent of j
   and where r0 is an identity value for r
.sc.
```

Mapping

As in the tridiagonal sub-derivation, there are three transformations for the basic array functions. The mappings on which the transformations are based are:

shapes: $rowwise([n,m],w) \rightarrow [n,w]$;

indices $[i,j] \rightarrow [i, search([i,j])]$ and the inverse $[i,j] \rightarrow [i, lookup([i,j])]$.

`element` The transformation for `element` is simply the application of the forward index mapping.

```
.sd.
  element(A:rowwise([m,n],w),[i,j])
  ==>
  element(A:[m,w],[i,search([i,j])])
.sc.
```

`generate` The transformation for `generate` applies the shape mapping to the generation's shape and the inverse index mapping to the generating function. Note that, unlike in the tridiagonal storage scheme, there are no invalid elements that need to be treated explicitly.

```
.sd.
  generate(rowwise([m,n],w),λ[i,j]·g)
  ==>
  generate([m,w],λ[i,j′]·λj·g (lookup(A,[i,j′])))
.sc.
```

`reduce` Two transformations are required for reduce: one for a reduction over a shape specified as `rowwise` (which applies the shape mapping to that shape); the other for a reduction over a row of such a shape (introduced by the transformation for splitting reductions). Both transformations apply the inverse index mapping to the generating function. Again, note that there are no special cases for invalid locations.

```
.sd.
  reduce(rowwise([n,n],w),λ[i,j]·g,r,init)
  ==>
  reduce([n,w],λ[i,j′]·λj·g (secondary(A, [i,j′])),r,init)
.sc.

.sc.
  generate(rowwise([n,n],w),λ[i,j]·g)
  ==>
  generate([n,w],λ[i,j′]·λj·g (secondary(A,[i,j′])))
.sc.
```

Additional transformations are required for the row-wise sub-derivation to simplify combinations of `search` and `lookup`. These functions are, almost, inverses:

$$search([i, lookup([i,j])]) \equiv j, \quad lookup([i, search([i,j])]) \equiv j.$$

Transformations based on these identities can eliminate occurrences of `search`, and so avoid the computational expense of searching the secondary store for an index. (The `lookup` function is implemented as a simple array indexing operation, so it is not as important to eliminate its uses.)

Appendix C

Gram-Schmidt Orthonormalization

The POT algorithm discussed in section 6.3 requires an orthonormalization function (the meaning of which is explained later): the particular orthonormalization function used in this work is (modified) Gram-Schmidt (GS). This function is responsible for about half the execution time of the POT algorithm, so its efficient implementation is important. In addition, the definition of GS, while conceptually simple, does require some attention to detail. GS thus represents a good test case for functional decomposition and derivational implementation.

In this appendix, I discuss the derivation of a Fortran Plus enhanced implementation of GS:

- **Definition of GS**
- **Functional specification**
- **FPE implementation**

C.1 Gram-Schmidt Orthonormalization

The SML specification of POT uses an orthonormalization function. Orthonormalization is a straightforward operation, but its definition requires some attention to detail: it is a good candidate to illustrate functional decomposition. The efficiency of implementation of orthonormalization is important: for example, orthonormalization consumes approximately half the execution time of POT. Orthonormalization is thus also a good test of the effectiveness of the DAP derivation.

The POT algorithm (amongst others) requires the formation of an orthonormal set of vectors from a given set of vectors. First, some definitions:

- Vectors U and V are said to be orthogonal if their inner-product is zero.
- A set of vectors is said to be orthogonal if each of the members is orthogonal to each of the other vectors.
- A vector is said to be normalized if its length is 1.
- A set of vectors is said to be orthonormal if it is orthogonal and if each of its members is normalized.

Often, a set of vectors is represented as a matrix whose columns are the vectors. Then, an orthonormalization function takes a matrix as argument and returns a matrix whose columns form an orthonormalized set of vectors. The orthonormalization function used for the DAP implementation of POT is Gram-Schmidt orthonormalization (GS). GS is based on the following:

- A vector is normalized by dividing by its magnitude (the square root of the sum of the squares of the components).
- That component of a vector U which is parallel to a vector V can be found by evaluating the inner-product (denoted (U, V)) of U and V: if V is normalized, the parallel component is $V * (U, V)$.
- If the component of U that is parallel to V is subtracted from U, the resulting vector has no component parallel to V: i.e. it is orthogonal to V.

Thus:

$$U \rightarrow U - V * (U, V)$$

The method employed by GS is to normalize one vector, and then make each of the remaining vectors orthogonal to that vector, and repeat until each of the vectors has been normalized. That is:

- Choose a vector that has not been normalized: normalize it.
- Subtract the appropriate component of the normalized vector from each of the other vectors (that have not yet been normalized).

For numerical stability, the order in which the vectors are normalized is important. In the POT algorithm, the eigenvectors are ordered according to the magnitude of the approximations to the eigenvalues.

C.1.1 SML Specification of GS

GS is an iterative process: it performs n iterations of an operation, where n is the size of the matrix to be orthonormalized. It can thus be specified using the `iterate` function.

```
fun GS(A:real matrix,order:int vector,n:int):real matrix
  = #1(iterate(GSstep,(A,n,constant(n,true),order,0),finished))
```

- The iteration function is `GSstep`, defined below. This function takes five arguments (combined as a 5-tuple):
  - The matrix processed so far.
  - The size of the matrix.
  - A boolean vector which records those columns that have not already been processed.
  - A vector of column indices which defines the order in which the columns are to be processed.
  - The number of columns which have so far been processed (used as an index into the vector of column indices to locate the column to be processed next).
- The terminating function is `finished`; this function simply tests if the number of columns processed is equal to the size of the matrix.

The `GSstep` function must:

- normalize the appropriate column of the matrix;
- orthogonalize the remaining columns with respect to that column;
- modify the mask vector to indicate that the column has been processed;
- increment the count of columns that have been processed.

The SML specification of `GSstep` is shown in figure C.1:

- `projection` computes the projection of one vector onto another.

- `make_ortho` performs the task of orthogonalizing, with respect to a given vector, all those columns of the matrix that have not already been normalized.

  The `generate.cols` function is a library function (defined in terms of `generate`) that has not yet been discussed: it generates a matrix in a 'column-wise' fashion. That is, the generating function takes a column index as argument, and returns a vector of elements (rather than a single element, as does the standard `generate`) which forms the designated column of the matrix. Consider the generating function:

  - If a column has already been normalized (as recorded in the `mask` vector), then it is left unaltered.
  - Otherwise, the projection of that column with the given vector is computed and the projection subtracted from the column.

- `normalize` normalizes a vector by dividing it by its magnitude.

```
fun projection(U:real vector,V:real vector):real vector
  = V*innerproduct(U,V);

fun make_ortho(A:real matrix,n:int,mask:bool vector,column:real vector)
    :real matrix
  = generate.cols(n,fn(i:int)=>
    if (not mask@[i])
    then column
    else col(A,i)-projection(col(A,i),column)
    );

fun normalize(U:real vector):real vector
  = U/sqrt(innerproduct(U,U);

fun normalize_column(A:real matrix,n:int,j:int):real matrix
  = generate.cols(n,fn(i:int)=>
    if (i=j) then normalize(col(A,j)) else col(A,i);

fun GSstep((A:real matrix,n:int,mask:bool vector,order:int vector,count:int))
  :real matrix*int*bool vector*int vector*int
  = let
    val count′:int = count+1;
    val j:int = order@[count′];
    val A′:real vector = normalize_column(A,n,j);
    val A″:real matrix = make_ortho(A′,n,mask′,j);
    mask′:bool vector = updateel(mask,j,false);
  in
    (A″,n,mask′,order,count′)
  end
```

Figure C.1: One step in GS

- `normalize_column` normalizes a specified column of a matrix.

- `GSstep` performs one step of the GS method.

    - If first increments the count of the columns processed.
    - It then retrieves the index of the next column to be processed from the `order` vector.
    - That column is normalized.
    - The mask of columns still to be processed is updated to exclude the current column (using the `updateel` library function).
    - The matrix is orthogonalized with respect to the normalized column.

Although GS is somewhat intricate in its details, functional decomposition allows it to be specified relatively painlessly.

C.1.2 DAP Implementation of GS

The function specification of orthonormalization makes liberal use of functional decomposition to simplify the specification. As is typical of specifications, the specification does not represent an efficient method for performing orthonormalization. The use of the functions `normalize_column` and `make_ortho` in particular represent potentially inefficient aspects of an implementation since they copy the entire matrix and since they construct the copy one column at a time.

Unfolding the functions and simplifying by Static Evaluation removes these inefficiencies: the bindings for A' and A'' are removed and column-wise matrix constructions are replaced with matrix generations, which may be interpreted as constructing all the elements simultaneously (rather than column by column). Converting to Array Form makes use of this interpretation to construct matrices in a data parallel manner, using data-parallel conditionals to mask off those columns that have already been normalized (and which should not be altered by the orthogonalizing that is performed for remaining columns).

The Fortran Plus Enhanced implementation of GS is shown in figure C.2.

```
   real*8 A(*n,*n)
   logical mask(*n)
   integer count
   real*8 column(*n)
   integer order(*n)
   real*8 project(*n,*n)

   .
   .
   .
1  count = 0
2  mask = true
3  continue

4  if (count.gt.n)then
5  goto 16

6  else
7  count = count+1
8  column = A( ,order(count))
9  column = column/sqrt(sum(column*column))
10 mask(order(count)) = .false.
11 project = matc(column,n)
12 project = matr(sumr(A*project),n)*project
13 A(matr(mask,n)) =  A-project
14 A( ,order(count)) = column
15 goto 3
16 continue
```

Figure C.2: Fortran Plus Enhanced implementation of GS

The function of each line is:

- 1,2 initialize the count of the number of columns processed so far and the mask of columns still to be processed.
- 3 is the start of the `GOTO` loop that implements the iteration of GS; line 16 is the end.
- 5 if all the columns have been processed, the loop is exited by the `GOTO 16` statement on line 5.
- 7 updates the count of the number of columns processed.
- 8,9 compute a vector that is the normalized version of the current column.

- 10 excludes the current column from the mask of columns still to be processed.

- 11-13 orthogonalize the rest of the matrix with respect to the normalized column:
  - Line 11 expands the normalized vector column-wise into a matrix
  - Line 12 uses that matrix to form simultaneously the inner-product of each column with the normalized vector (the expression `sumr(A*project)`), and then to compute simultaneously the projection of each column onto the normalized vector: `matr(...,n)*project`.
  - The projections are subtracted from the matrix in line 13; the mask in the assignment (`matr(mask,n)`) ensures only those columns that have still to be processed are affected.

- 14 copies the normalized vector into the appropriate column of the matrix.

- 15 returns execution to the start of the loop.

C.2 Summary

As with the FPE implementation of the main section of POT, the implementation of orthonormalization, particularly the computation of the projections of the columns onto the current column (line 12), would probably seem quite strange to one not experienced in programming the DAP. Yet it follows by the application of fairly simple rules to a straightforward specification.

As was argued in the evaluation of the derived implementation of POT (section 6.3.3), the virtual parity of the derived and hand-crafted implementations of POT indicates that the implementation of GS is quite efficient, and thus that the DAP derivation is effective.

Bibliography

[1] Active Memory Technology Corporation, Active Memory Technology Inc., 16802 Aston St, Suite 103, Irvine, California, 92714, USA. *Introduction to Fortran-Plus Enhanced*, November 1990.

[2] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] Tom Axford and Mike Joy. List processing primitives for parallel computation. *Computer Languages*, 12(1):1–17, January 1993.

[4] R.J.R. Back. A calculus of refinements for program derivation. *Acta Informatica*, 25:593–624, 1988.

[5] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13:133–180, 1990.

[6] Roland Backhouse. An exploration of the Bird-Merteens formalism. In *STOP Summer School on Constructive Algorithmics*, pages 1–61, September 1989.

[7] F. L. Bauer, H. Ehler, B. Horsch, B. Möller, O. Paukner, and P. Pepper. *The Munich Project CIP, Vol. 2: The Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[8] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by stepwise transformations — computer-aided intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, 1986.

[9] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper. Formal program construction by transformations — computer-aided, intuition guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[10] Siegfried Benker, Barbara M Chapman, and Hans Zima. Vienna Fortran-90. In Robert Voigt and Joel Saltz, editors, *Proceedings of the Scalable High-Performance Computing Conference*, pages 51–59. IEEE Computer Society Press, April 1992.

[11] Dimitri P. Bertsekas and Tsitsiklis John N. *Parallel and Distributed Computation*. Prentice-Hall, 1989.

[12] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[13] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, April 1989.

[14] R. S. Bird and John Hughes. The alpha-beta algorithm: An exercise in program transformation. *Information Processing Letters*, 24(1):53–57, January 1987.

[15] E. A. Boiten, H. A. Partsch, D. Tuijnman, and N. Völker. How to produce correct software — an introduction to formal specifications and program development by transformations. *The Computer Journal*, 35(6):547–554, 1992.

[16] Eerke A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18(2):139–179, April 1992.

[17] James M. Boyle. A transformational component for programming languages grammar. Technical Report ANL-7690, Argonne National Laboratory, July 1970.

[18] James M. Boyle. Program adaptation and program transformation. In R. Ebert, J. Lueger, and L. Goecke, editors, *Practice in Software Adaptation and Maintenance*, pages 3–20. North-Holland Publishing Co., 1980.

[19] James M. Boyle. Towards automatic synthesis of linear algebra programs. In M. A. Hennell and L. M. Delves, editors, *Production and Assessment of Numerical Software*, pages 223–245. Academic Press, 1980.

[20] James M. Boyle. Abstract programming and program transformation. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press/Addison-Wesley, 1989.

[21] James M. Boyle and Terence J. Harmer. Functional specifications for mathematical computations. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Construction of Programs from Specifications*, pages 205–224, 1991.

[22] James M. Boyle and Terence J. Harmer. A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, 2(1):81–126, 1992.

[23] James M. Boyle and Managur N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, 10(5):574–588, September 1984.

[24] Tore A. Bratvold. Determining useful parallelism in higher order functions. In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, September 1992.

[25] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[26] M. Clint, J. S. Weston, and C. W. Bleakney. Comparison of parallel Fortran environments on the AMT DAP 510 for a linear algebra application. *Concurrency: Practice and Experience*, 6(3):193–204, May 1994.

[27] A. Contessa, E. Cousin, C. Coustet, M. Cubero-Castan, G. Durrieu, B. Lecussan, M. Lemaître, and P. Ng. MaRS, a combinator graph reduction multiprocessor. In M. Rem E. Odijk and J. C. Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe, I*, volume 365 of *Lecture Notes in Computer Science*, pages 176–192. Springer-Verlag, June 1989.

[28] M. D. Cripps, J. Darlington, A. J. Field, P. G. Harrison, and M. J. Reeve. The design and implementation of ALICE: A parallel graph reduction machine. In S. S. Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*. IEEE Computer Society Press, 1987.

[29] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *PARLE*. Springer-Verlag, June 93.

[30] John Darlington, Peter Harrison, Hessam Khoshnevisan, Lee McLoughlin, Nigel Perry, Helen Pull, Mike Reeve, Keith Sephton, Lyndon While, and Sue Wright. A functional programming

environment supporting execution, partial execution and transformation. In E. Odijk, M. Rem, and J. C. Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe, I*, volume 365 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, June 1989.

[31] Michael L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, December 1990.

[32] M. Feather. A system for transformationally deriving programs. *ACM Transactions on Programming Languages and Systems*, 4(1):1–21, January 1982.

[33] Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[34] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical report, Rice University, Houston TX-77251, USA, January 1992. [Document source available by anonymous FTP from titan.rice.edu:public/HPFF].

[35] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[36] R. Gabriel and J. McCarthy. QLISP. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 63–89. Kleuwer Academic Pub., 1988.

[37] Marc Garbey and David Levine. Massively parallel computation of conservations laws. *Journal of Parallel Computing*, 16:293–304, 1990.

[38] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programmning*, 18(2):121–160, 1989.

[39] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, April 93.

[40] Benjamin Goldberg. Multiprocessor execution of functional programs. *International Journal of Parallel Programming*, 17(5):25, 1988.

[41] Ron Goldman and Richard P. Gabriel. Qlisp: Experience and new directions. In *Parallel Programming: Experience with Applications, Languages and Systems*, volume 23 of *SIGPLAN Notices*. ACM Press/Addison-Wesley, September 1988.

[42] E. Pascal Gribomont. Concurrency without toil: A systematic method for parallel program design. *Science of Computer Programming*, 21(1):1–56, August 1993.

[43] G. Hains and L. M. R. Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238–245, 1993.

[44] R. Halstead. Parallel computing using Multilisp. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*. Kluwer-Academic Publishers, 1987.

[45] Kevin Hammond. Parallel functional programming: An introduction. In *PASCO '94 First International Symposium on Parallel Symbolic Computation*. World Scientific Publishing Company, September 1994.

[46] Luddy Harrison and David A. Padua. PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp. In *International Conference on Supercomuting*, pages 527–538. ACM Press/Addison-Wesley, July 1988.

[47] P. G. Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

[48] P. G. Harrison and H. Khoshnevisan. Algebraic transformation techniques for functional languages. *The Computer Journal*, 31(3):229–242, 1988.

[49] P. G. Harrison and H. Khoshnevisan. The mechanical transformation of data types. *The Computer Journal*, 35(2):138–147, 1992.

[50] P. G. Harrison and M. Reeve. The parallel graph reductions machine ALICE. In J. H. Fasel and R. M. Keller, editors, *Graph Reduction: Proceedings of a Workshop, Santa Fe*, volume 279 of *Lecture Notes in Computer Science*, pages 181–202. Springer-Verlag, 1986.

[51] W. Ludwell Harrison, III and David A. Padua. Representing S-expressions for the efficient evaluation of lisp on parallel processors. In Kai Hwang, Steven M Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 703–710. IEEE Computer Society Press, August 1986.

[52] R. Hempel. The MPI standard for message passing. In Wolfgang Gentzsch and Uwe Harms, editors, *High Performance Computing and Networking, HPCN '94*, volume 797 of *Lecture Notes in Computer Science*, pages 247–252. Springer-Verlag, April 1994.

[53] L. O. Hertzberger and W. G. Vree. A coarse grain parallel architecture for functional languages. In M. Rem E. Odijk and J. C. Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe,I*, volume 365 of *Lecture Notes in Computer Science*, pages 269–285. Springer-Verlag, June 1989.

[54] David Hildum and Jacques Cohen. A language for specifying program transformations. *IEEE Transactions on Software Engineering*, 16(6):630–638, June 1990.

[55] Jonathan M.D. Hill. The AIM is laziness in a data-parallel language. In K. Hammond and J. T. O'Donnell, editors, *GLA*. Springer-Verlag, 93.

[56] Jonathan M.D. Hill and Keith Clarke. Parallel Haskell: The vectorisation monad. Technical Report 658, QMW CS, Dec 93.

[57] C. A. R. Hoare and *et al*. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987. Corrigenda in *30*, 9, p. 770.

[58] C. A. R. Hoare, J. F. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30(8):701–739, 1993.

[59] High performance Fortran. Scientific Computing, volume 2, issues 1,2. Entire issue devoted to HPF.

[60] S. Hwang and D. Rushall. The nu-STG machine: a parallelized Spineless Tagless Graph reduction machine in a distributed memory architecture. In *Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages*, 92.

[61] Alan Jennings. *Matrix Computations for Engineers and Scientists*. Wiley, first edition, 1977.

[62] Guido Karel Jouret. *Exploiting Data-Parallelism in Functional Languages*. PhD thesis, Department of Computing, Imperial College of Science Technology and Medicine, University of London, 1991.

[63] Einar W. Karlsen, Bernd Krieg-Brückner, and Owen Traynor. The PROSPECTRA system: A unified development framework. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors,

Algebraic Methodology and Software Technology (AMAST '91), pages 421–433. Springer-Verlag, May 1991.

[64] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2:151–164, 1993.

[65] John A. Keane. An overview of the Flagship system. *Journal of Functional Programming*, 4(1):19–45, January 1994.

[66] J. E. Key. Computer program for solution of large sparse unsymmetric systems. *Int. J. Numer. meth Eng.*, 6:497–509, 1973.

[67] D. A. Kranz, R. H. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. *ACM SIGPLAN Notices*, 24(7):81–90, 1989. SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, OR.

[68] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[69] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Parallel Programming: Experience with Applications, Languages and Systems*, volume 23 of *SIGPLAN Notices*, pages 100–110. ACM Press/Addison-Wesley, September 1988.

[70] Rita Loogen, Herbert Kuchen, and Klaus Indermark. Distributed implementation of programmed graph reduction. In M. Rem E. Odijk and J. C. Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe,I*, volume 365 of *Lecture Notes in Computer Science*, pages 136–157. Springer-Verlag, June 1989.

[71] Andreas Maaßen. Parallel programming with data structures and higher order functions. *Science of Computer Programming*, 18:1–38, 1992.

[72] Grant Malcolm. Data structures and program transformations. *Science of Computer Programming*, 14:255–279, October 1990.

[73] Shogo Matsui, Yoshinobu Kato, Shinsuke Teramura, Tomoyuki Tanaka, Nobuyuki Mohri, Atsushi Maeda, and Masakazu Nakanishi. SYNAPSE: A multi-microprocessor lisp machine with parallel garbage collector. In *Parallel Algorithms and Architectures*, volume 269 of *Lecture Notes in Computer Science*, pages 131–137. Springer-Verlag, May 1987.

[74] Carl McCrosky. Intermediate container removal. *Computer Languages*, 16(2):179–195, 1991.

[75] Patrick F. McGehearty and Edward J. Krall. Execution of common lisp programs in a parallel environment. *Parallel Computing and Computers for A.I.*, page 222, ???

[76] Patrick F. McGehearty and Edward J. Krall. Potentials for parallel execution of common lisp programs. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 696–702. IEEE Computer Society Press, August 1986.

[77] Lambert Meertens. *Constructing a Calculus of Programs*, volume 375 of *Lecture Notes in Computer Science*, pages 66–90. Springer-Verlag, June 1989.

[78] Simon Merrall and Julian Padget. Collections and garbage collection. In *proceedings of the International Workshop on Memory Management*, pages 473–489. LNCS 637, Sept 92.

[79] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford Science Publications. Oxford University Press, 1990.

[80] Jagdish J. Modi. *Parallel Algorithms and Matrix Computations*. Oxford University Press, 1988.

[81] Carroll Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[82] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[83] Helmut A. Partsch. *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[84] Peter Pepper. A simple calculus for program transformation. *Science of Computer Programming*, 9(3):221–262, December 1987.

[85] A. Pettorossi and R. M. Burstall. Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica*, 18:181–206, 1982.

[86] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1986.

[87] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In E. Odijk and J.-C. Syre, editors, *Parallel Architectures and Languages Europe, I*, volume 365 of *Lecture Notes in Computer Science*, page 193. Springer-Verlag, 1989.

[88] Gruia-Catalin Roman and C. Donald Wilcox. Architecture-directed refinement. *IEEE Transactions on Software Engineering*, 20(4):239–258, April 1994.

[89] Peter Schnorf, Mahadevan Ganapathi, and John L. Hennessy. Compile-time copy elimination. *Software — Practice and Experience*, 23(11):1175–1200, November 1993.

[90] Ambuj K. Singh. Program refinement in fair transition systems. *Acta Informatica*, 30:503–535, 1993.

[91] D. B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 20(3):205–221, June 1993.

[92] D.B. Skillicorn. Parallelism and the Bird-Meerteens formalism. FTP from qucis.queensu.ca, April 1992.

[93] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[94] Thinking Machines Corporation, Thinking Machines Corporation, Cambridge, MA, USA. *CM Fortran Reference Manual*, October 1991.

[95] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31:105–135, 1994.

[96] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 90.

[97] J. S. Weston and M. Clint. Two algorithms for the parallel computation of eigenvalues and eigenvectors of large symmetric matrices using the ICL DAP. *Parallel Computing*, 13:281–288, 1990.

[98] J. S. Weston, M. Clint, and C. W. Bleakney. The parallel computation of eigenvalues and eigenvectors of large hermitian matrices using the AMT DAP 510. *Concurrency: Practice and Experience*, 3(3):179–185, June 1991.

[99] A. Wilström. *Functional Programming using Standard ML*. Prentice Hall, 1987.

[100] Hans Wolfe Wolfe and Barbara Wolfe Wolfe. *Supercompilers for Parallel and Vector Computers*. Frontier Series. ACM Press/Addison-Wesley, 1990.

[101] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989. Based on author’s Ph.D. thesis.

[102] J. A. Yang and Y. Choo. Parallel-programming transformation using a metalanguage. *ACM SIGPLAN Notices*, 26(7):11–20, 1991. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, VA, April1991.

[103] Hong Zhu. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):69–112, January 1994.